Math. Struct. in Comp. Science (1998), *vol.* 8, *pp.* 681–710. Printed in the United Kingdom
© 1998 Cambridge University Press

Concurrent programming as proof net construction

G. PERRIER

LORIA, Université Nancy 2, Campus Scientifique -B.P. 239,
54506 Vandœuvre-les-Nancy Cedex, France
E-mail: perrier@loria.fr

Received 11 December 1996; revised 26 November 1997

We propose a concurrent process calculus, called *Calcul Parallèle Logique* (CPL), based on the paradigm of computation as proof net construction in linear logic. CPL uses a fragment of first-order intuitionistic linear logic where formulas represent processes and proof nets represent successful computations. In these computations, communication is expressed in an asynchronous way by means of axiom links. We define testing equivalences for processes, which are based on a concept of interface, and use the power of proof theory in linear logic.

1. Introduction

A few years ago, Miller (1993) took up the following challenge: 'Can we view a given process calculus as a logic?'. Shortly afterwards he added that this question is different from viewing 'logic as an auxiliary language to that of the process calculus'. He proposed to consider logical formulas as processes, logical connectives as process combinators and proofs as process computations.

For a computer scientist, this approach has the advantage of belonging to the framework of pure logic programming, where programs and specifications are merged in the same logical formalism. So, a unique object, a formula, can be considered from two angles:

— seen from a logical angle, it is the expression of a specification;
— seen from a computational angle, it is a process, *i.e.*, an abstraction of a concurrent program.

Logic is thus used in two ways: the language constitutes a framework for specification and the proof theory gives a framework for computation.

To address this question, Miller chose the framework of linear logic because it is a resource-sensitive logic in contrast to classical and intuitionistic logics, which are essentially static. In a classical or intuitionistic proof, as soon as a proposition is established, it remains true forever. In contrast, an essential feature of linear logic (LL) (Girard 1987; Girard 1989) is that proofs are viewed as actions that transform hypotheses into conclusions like chemical reactions. In a linear logic proof, formulas are managed like resources that can be determined *a priori* as consumable or unbounded. Thus, the notion of *state change* can be expressed in linear logic.

https://doi.org/10.1017/S0960129598002655 Published online by Cambridge University Press

Miller (1993) used this feature to propose a translation of the π-calculus (Milner *et al.* 1992) into linear logic. In this translation, processes are represented by formulas of linear logic and operations on processes by linear logic connectives, with the exception of an important case: Miller uses a proper axiom with non-logical constants to represent synchronous communication.

Before Miller's paper, Andreoli and Pareschi (Andreoli and Pareschi 1991) were the first to use linear logic to introduce the notion of state change in logic programming. In their language, Linear Objects (LO), goals are linear logic formulas that represent objects changing their internal states during the computation, which consists of a proof construction in linear logic. In LO, objects can be created concurrently and then communicate via a blackboard using an extra-logical mechanism.

Kobayashi and Yonezawa (1995) developed a calculus, ACL, which differs mainly from Miller's proposal in its representation of communication: they used linear connectives to express asynchronous communication in a purely logical framework. That is why ACL is a model of concurrent calculus in pure linear logic, whereas the translation of Milner's π-calculus by Miller is a theory in linear logic.

Independently, Lincoln and Saraswat (1992) proposed a similar model, but with the particular motivation of representing concurrent constraint programming.

All the work cited above uses the formalism of *linear sequent calculus*. However, this formalism has a major defect: it introduces artificial sequentiality between inferences in a proof, which means that we need to study the property of inference permutability to separate the necessary aspect of this sequentiality from the accidental aspect (Galmiche and Perrier 1994). This defect becomes very embarrassing when we want to represent concurrent computations by proofs: concurrency is hidden by the formalism and we must use inference permutability again to highlight it.

However, at the origin of this approach, Girard (1987) proposed a natural syntax for proofs in linear logic, the syntax of *proof nets*, that does not have the defects of the sequent calculus. Many papers have been devoted to model concurrency in the formalism of proof nets but all resort to the paradigm of *proofs as programs*, which is the basis of the relationship between logic and functional programming (Abramsky 1993; Bellin and Scott 1994). In this approach, processes are represented by proof nets, interactions between processes are performed by means of cuts and process computations correspond to cut elimination.

By proposing a new calculus, which we call *Calcul Parallèle Logique* (CPL), we choose to use proof nets in the paradigm of *proof search as computation*, which characterises the relationship between logic and logic programming. Formulas are processes, logical connectives are process combinators and the construction of proof nets expresses the execution of process computations. Concurrency between processes is represented by parallel construction of different branches of a proof net and communication is expressed by axiom links between these branches. Thus vertices in a proof net represent states of processes, links between formulas and their components represent transitions, and axiom links correspond to asynchronous communications by message passing. Success, deadlock and divergence in the construction of a proof net are interpreted as success, deadlock and divergence of the computation.

https://doi.org/10.1017/S0960129598002655 Published online by Cambridge University Press

To define the syntax of CPL processes, we choose a restricted fragment of first-order intuitionistic linear logic (ILL)†. The main restriction consists of only considering implications in the form $M \multimap P$ with M atomic. Such an implication represents a receiver that expects a message M in order to be transformed into a process P. In this way, the syntax of proof nets is simplified with respect to the classical proof nets of Girard (1987) (boxes are deleted) and even with respect to the intuitionistic proof nets of Lamarche (1996) (output formulas are only atomic formulas). Section 2 is devoted to a presentation of their specific syntax, called *computation structure*.

In Section 3, we are interested in the definition of a specific correctness criterion for these proof nets, which takes advantage of their particular form. This criterion is well suited to the interpretation of correct computation structures as concurrent *process computations*. A computation structure induces three kinds of dependence between its links, which are caused by state transitions, communications and eigenvariable generations, respectively. In a correct computation, this dependence relation must be an *order* because it corresponds to precedence in the course of time.

The geometrical form of computations expresses true concurrency faithfully. In the absence of a global clock, the partial order between communications induces local clocks and is the basis for the measures of *time and space complexity* defined in Section 4. Then we formalise the notions of *success, deadlock* and *divergence* of computations, while still taking this order between communications into account.

Section 5 is devoted to the *sequentialisation of computations* in the linear sequent calculus. In this way, we indirectly prove the equivalence between CPL proof nets and Girard's proof nets for the fragment of ILL used by CPL. Then, in a sequential form, CPL can be compared with process calculi that are presented in the form of transition systems. Here, we only lay the foundations of this comparison, which we will carry out in further papers.

In Section 6, we give an outline of the expressiveness of CPL. In spite of syntactic restrictions, the expressiveness of CPL is very large: recursive definitions of processes, synchronous communication, mobile processes, and so on. In particular, we give some elements that show how the π-calculus can be translated into CPL. By restricting the syntax of CPL, we show how non-determinism can be reduced and concurrency made coarser-grained to give a realistic programming language.

In Section 7, we define testing equivalences for processes (Hennessy and de Nicola 1983). We are especially interested in the equivalence based only on the positive tests, because it can be easily formalised in linear logic proof theory. For a process P, we define its *co-processes*, that is, agents that successfully test one aspect of the possible interaction between P and the outside world. This concept was inspired by the notion of co-agent developed by Miller to define a testing equivalence for CCS (Miller 1993). Kobayashi and Yonezawa (1993) used this notion to establish a strong equivalence between ACL processes. The interest of this notion is that it can be totally expressed in terms of provability in linear logic. Our aim is to exploit the resources of proof theory

† See the sequent calculus of ILL in Appendix A.

https://doi.org/10.1017/S0960129598002655 Published online by Cambridge University Press

to the maximum. So, we develop a whole logic calculus around the notion of *interface*. Interfaces are sets of co-processes used to characterise the interaction between a process and the outside globally. We introduce a relation of *implication between interfaces* that helps to simplify them and to compare processes. We develop a method for determining the interface of a given process, the main point of which is that we can restrict the set of agents used for testing to *trace processes* without undermining the result. Trace processes are sequences of actions that are either message emissions or message receptions. By comparing interfaces, we can compare processes, but these can also be compared directly with the relation of logic deduction, which is stronger than the relation based on interfaces.

For the sake of conciseness, some proofs will only be sketched. Detailed proofs can be found in Perrier (1996).

2. The syntax of the calculus

2.1. *Processes as formulas*

The basic entities of CPL are *processes*, which are represented by formulas. Among the processes, we distinguish *messages*, which are represented by atomic formulas. Considering a message as a process implies that communication in CPL is *asynchronous*. When the communication is synchronous, a message does not have an autonomous existence because it is consumed as soon as it is produced.

The information contained in messages is encoded in a symbolic form, as in Prolog. For this, we consider a term algebra $\mathscr{T}[X]$ on a set of variables X and atomic term symbols and a countable set of predicate symbols $\mathscr{M}$. The set of messages $\mathscr{M}(\mathscr{T}[X])$ is the set of messages that are built from $\mathscr{M}$ and $\mathscr{T}[X]$.

Definition 2.1. CPL processes[†] are linear logic formulas defined by the following grammar:

$$P \ ::= M \mid 1 \mid 0 \mid M \multimap P \mid P \otimes P \mid P \& P \mid !P \mid \forall x P \mid \exists x P$$

where M is any element of $\mathscr{M}(\mathscr{T}[X])$ and x any element of X.

We adopt the following precedence among the operators : $\forall x$, $\exists x$, $! > \multimap > \otimes > \&$. We suppose that the only operator that is not associative $\multimap$ is right-associative. The meaning of the linear connectives used here will be given by the associated logical rules viewed as transition rules, but we can already say that processes fall into two classes:

— *Recursive processes*: these are represented by formulas prefixed with ! and can be duplicated and erased.
— *Linear processes*: these are the other processes, and apart from the process 1, they cannot be duplicated or erased.

We often manipulate not only single processes but *systems of processes* that are in CPL multi-sets of processes. We use Γ, Δ (which may be indexed) as meta-variables over systems of processes. We write $!\Gamma$ to represent a system where all processes result from prefixing the formulas of Γ with !.

[†] Subsequently, CPL processes will simply called processes. This remark also applies to the other objects of CPL that will be defined in the rest of this paper.

https://doi.org/10.1017/S0960129598002655 Published online by Cambridge University Press

2.2. *Process computations as proof nets*

If processes are represented by linear logic formulas, process computations are represented by proof nets. However, we do not use the standard form of proof nets and classical correctness criterion for two reasons:

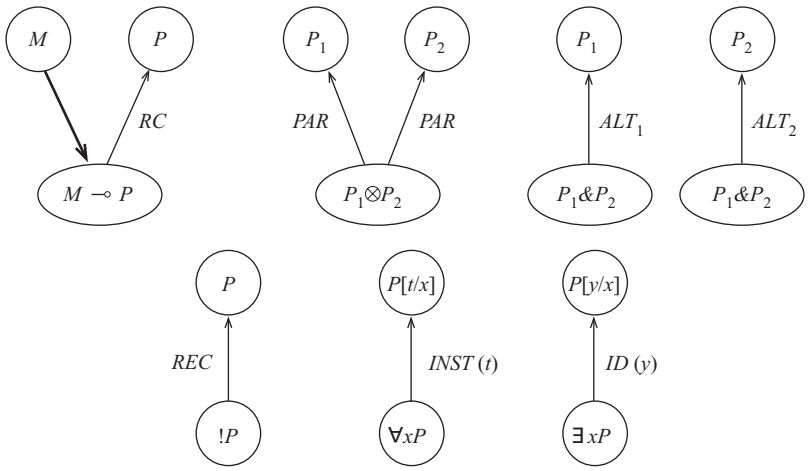
— we aim to take advantage of the specificity of the CPL logical fragment (its oriented character and the restriction on linear implication) for simplifying its proof nets; in particular, we do not need switchings for the treatment of quantifiers, in contrast with the more general case (Girard 1991);
— we do not study proof nets for themselves but as representations of concurrent computations; thus their form must be adjusted to this interpretation.

We start by defining the syntax of these proof nets, which are called *CPL computation structures.* An elementary computation structure, named an *event*, corresponds to a link of a proof net.

Definition 2.2. A *CPL event* is an oriented graph with the following properties:

— vertices, which represent *process states*, are labelled with processes[†];
— edges that represent *transitions* are labelled by a transition rule and edges that represent *communications* are not labelled[‡].

Each event has one of the following types:



Every event of type ID(y) verifies the condition that the eigenvariable y is not free in $\exists x P$.

The *source* $S(e)$ of an event e is the set of its process states without incident transitions and its *result* $R(e)$ is the set of its process states without outgoing transitions.

Events represent the indivisible units of computation. The different types of events express transition rules corresponding to logical rules of LL. In the linear sequent calculus,

[†] When there is no ambiguity, a vertex is denoted by its labelling process.
[‡] They are drawn in bold.

https://doi.org/10.1017/S0960129598002655 Published online by Cambridge University Press

they all correspond to left rules, though there is a dual definition using right rules, as used in Miller (1993) and Kobayashi and Yonezawa (1995).

The first event is made up of a communication, where the process $M \multimap P$ receives the message M, and of transition RC, where the process $M \multimap P$ is changed into P after the reception. In Girard's proof nets, this would be represented by two links: an axiom link for the communication and a link associated to the rule $\multimap_L$ for the transition.

PAR and ALT transitions, respectively, express parallelism and choice. If P_1 or P_2 is atomic and represents a message, a PAR transition from $P_1 \otimes P_2$ can be viewed as a sending and then it is dual to an RC transition, which represents a reception. There is an essential difference between reception and sending, which guarantees the correctness criterion of proof nets in their construction: in contrast with sending, reception is blocking.

A REC transition expresses the generation of a linear process from a recursive process and, as we will see below, this can be iterated any number of times from the same source.

An INST(t) transition expresses the instantiation of a variable x with a term t that represents some information. So we can use $\forall$ as a *generalisation operator* to perform value passing in communication. Put before a receiver, it makes it able to receive not only one message but any one in an expected pattern. For instance, the process $\forall x\ M(f(x)) \multimap P$ is able to receive the message $M(f(a))$ to become $P[a/x]$ in a transition RC preceded by a transition INST(a).

An ID(y) transition expresses the instantiation of a variable x with a private identifier or eigenvariable y. The operator $\exists$ acts as a *restriction operator*. In particular, it allows us to model private communication channels.

The constants 1 and 0 play a role in the outcome of computations, which will be explained in Subsection 4.2.

Now, we define a relation of causality on sets of events as follows.

Definition 2.3. Let e_1 and e_2 be two elements of a set of events. e_1 is said to *cause* e_2 *immediately* if $R(e_1)$ and $S(e_2)$ are not disjoint. It is denoted: $e_1 \lessdot e_2$.
Let $e_0, \cdots, e_n$ be a sequence of n events with n>0. e_0 is said to *cause* e_n if, for any k such that k<n, $e_k \lessdot e_{k+1}$. It is denoted: $e_0 < e_n$.

By composing events according to some rules that preserve the linearity of computations and the finiteness of causality, we can build complex computation structures.

Definition 2.4. A *CPL computation structure* S is a union of events with possibly isolated process states that respects the following conditions:

(i) every linear process state of S belongs to the source of at most one event;
(ii) every process state of S belongs to the result of at most one event;
(iii) for each event e of S, the set of events e' such that $e' < e$ is finite.

The first two conditions express the linearity of computation structures. The fact that we can perform any number of REC transitions from a recursive process, expresses the power of the exponential operator ! – like the corresponding operator of the π-calculus (Milner 1991), it allows duplication.

The last condition of the above definition expresses the finiteness of causality.

To describe computation structures, we need a minimal vocabulary, which is defined as follows.

https://doi.org/10.1017/S0960129598002655 Published online by Cambridge University Press

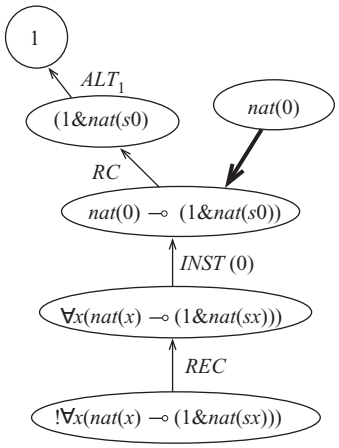
Definition 2.5. If S is a computation structure, the set $Init(S)$ of process states that do not belong to the result of some event represents the *initial state* of S.
In S, the set $Res(S)$ of process states that do not belong to the source of some event, along with the recursive process states, is called the *residue* of S.

If a computation structure is not empty, its initial state is never empty because of Condition 3 in Definition 2.4, which means that every process of a computation started one day.

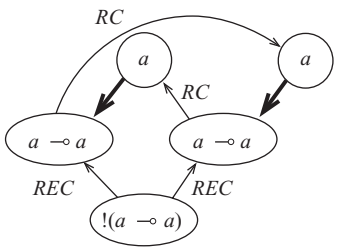
The residue is the set of the processes that remain after a computation and it can constitute the initial state of a new computation. All recursive processes belong to the residue because they can always be the starting point of new linear processes according to Condition 1 in Definition 2.4.

A computation structure with a finite number of process states is said to be *finite*; otherwise it is said to be *infinite*. Given a finite computation, its residue is identical to its final state, but this is not true for an infinite computation, as Example 2.8 below shows.

Example 2.6. The computation structure S_1 below is *finite* and it processes integers, which are represented in symbolic form. In particular, sx represents the successor of x. Its initial state $Init(S_1)$ is equal to $\{nat(0),\ !\forall x(nat(x) \multimap (1 \& nat(sx)))\}$, and its residue $Res(S_1)$ is equal to $\{1,\ !\forall x(nat(x) \multimap (1 \& nat(sx)))\}$.



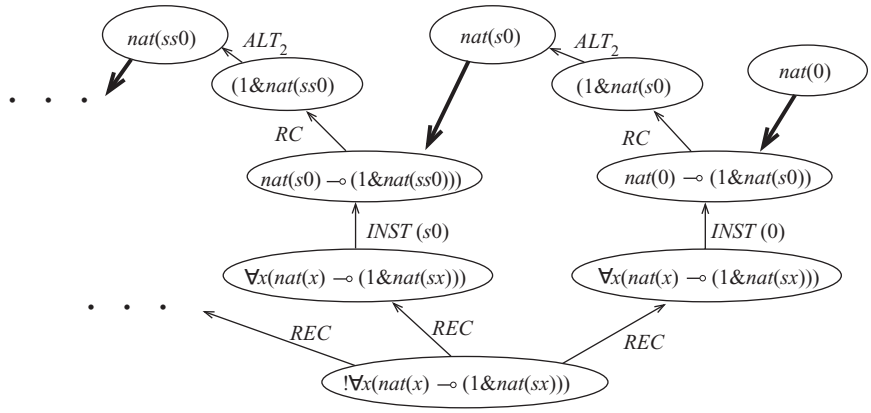
Example 2.7. The computation structure S_2 below is *finite*. Its initial state $Init(S_2)$ and its residue $Res(S_2)$ are identical: they are equal to $\{!(a \multimap a)\}$.



Example 2.8. The computation structure S_3 below has the same initial state as S_2 but it is *infinite* because of an infinite number of transitions REC. Its residue $Res(S_3)$ is equal

https://doi.org/10.1017/S0960129598002655 Published online by Cambridge University Press

to $\{!\forall x(nat(x) \multimap (1\&nat(sx)))\}$. In Section 5, we will see that $Res(S_3)$ is not really a final state.

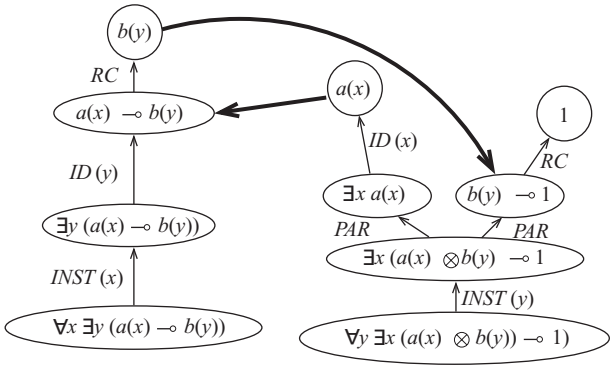


3. Correctness criterion for process computations

Not every computation structure can be interpreted as a process computation. First, communication links introduce a dependence relation between initially independent processes; this relation must be an order because it expresses precedence between events in the course of time. Thus a computation structure must be without cycles to represent a process computation. In Example 2.7, S_2 cannot be a process computation because it includes a cycle. This is not the case for Examples 2.6 and 2.8.

In some other computations, because of the presence of transitions ID, the absence of cycles is not sufficient. The private nature of eigenvariables must also be guaranteed. The following example illustrates how the private nature of some eigenvariables can be violated.

Example 3.1. In the computation structure below, transition INST(x) must follow transition ID(x) because the former uses the eigenvariable x generated by the latter. For the same reason, transition INST(y) must follow transition ID(y), which contradicts the first constraint.



To characterise the constraints induced by eigenvariables, we need a notion of *eigenvariable scope*.

https://doi.org/10.1017/S0960129598002655 Published online by Cambridge University Press

Definition 3.2. Let S be a computation structure. Let x be an eigenvariable of S and P_x the result of the associated event. The *scope* $S(x)$ of x is the least subgraph of S that verifies the following properties:

(i) if x is free in P_x, then $P_x \in S(x)$;
(ii) if $P \in S(x)$, then every neighbour of P where x is free is a vertex of $S(x)$.
(iii) if some element of $Res(S)$ belongs to $S(x)$, all elements of $Res(S)$ where x is free belong to $S(x)$.

The vertices of $S(x)$ without incident edges are called the *entering points* of $S(x)$.

If x is not free in P_x, then $S(x)$ is empty, otherwise $S(x)$ represents the private domain of the eigenvariable x that is shared by all processes belonging to $S(x)$.

The last condition of Definition 3.2 guarantees the sequential compositionality of computations. Suppose that a process state P where the eigenvariable x is free belongs to $Res(S)$ but not to $S(x)$. If we continue the computation, it is possible for P to enter $S(x)$ by means of a communication. This is why $S(x)$ includes not only the process states that are effectively in the eigenvariable scope but also includes those that are potentially in this scope.

Let us examine the nature of the entering points of $S(x)$. Three cases are possible:

— An entering point P belongs to $Init(S)$. We cannot allow such a possibility in a correct process computation, again, because of sequential compositionality, but in the opposite direction compared with the previous argument. By extending S from its initial state, we can obtain a process state that belongs to the scope $S(x)$ but which is generated before the transition ID(x).
— An entering point of $S(x)$ is the result of another transition ID(x). Obviously, the computation is not correct.
— An entering point of $S(x)$ is the result of a transition INST(t). The private nature of the scope $S(x)$ implies that this transition must follow the transition ID(x) that generates the eigenvariable x. That is why the computation structure of Example 3.1 is not correct. The scope of the eigenvariable x is made up of the process states $a(x)$, $a(x) \multimap b(y)$, $\exists y(a(x) \multimap b(y))$. Then the transition INST(x) must follow ID(x). The scope of the eigenvariable y is made up of the process states $a(x) \multimap b(y)$, $b(y)$, $b(y) \multimap 1$, $\exists x a(x) \otimes b(y) \multimap 1$. Then the transition INST(y) must follow ID(y), which leads to a contradiction.

The dependencies induced by eigenvariables come in addition to those resulting from communications and a way of characterising them all is to define a precedence relation between events of a computation structure.

Definition 3.3. The *precedence relation* of a computation structure S is the least transitive relation on events of S that includes the relation of causality $<$ and that verifies the following property: 'every transition ID(x) of S precedes the transitions that have entering points of $S(x)$ as results'.
For two events e_1 and e_2 of S, e_1 *precedes* e_2 is denoted: $e_1 \prec e_2$.

Now, with this relation of precedence, we are in a position to give a correctness criterion for process computations.

https://doi.org/10.1017/S0960129598002655 Published online by Cambridge University Press

Definition 3.4. A computation structure S is a *process computation* if it verifies two conditions[†]:

(i) its precedence relation is a strict order;
(ii) there is no entering points of eigenvariable scopes in the initial state $Init(S)$.

The computation structures of Examples 2.6 and 2.8 are process computations but this is not the case for those of Examples 2.7 and 3.1. For instance, the following precedences occur among the events of the computation structure in Example 3.1: $\mathrm{ID}(x) \prec \mathrm{INST}(x) \prec \mathrm{ID}(y)$ and $\mathrm{ID}(y) \prec \mathrm{INST}(y) \prec \mathrm{PAR} \prec \mathrm{ID}(x)$. This shows that $\prec$ is not a strict order here.

Since we have been careful in establishing the correctness criterion of process computations, we can deduce two properties of compositionality. The first property is *sequential compositionality*.

Theorem 3.5. Let C_1 and C_2 be two process computations such that $Res(C_1) = Init(C_2)$. The union of C_1 and C_2 is a process computation.

Proof. Let C be the union of C_1 and C_2. First, C is a computation structure because the only common process states of C_1 and C_2 are those of $Res(C_1)$ and $Init(C_2)$ and their sets of events are disjoints.

Second, $Init(C) = Init(C_1)$ and entering points of eigenvariable scopes of C cannot be present in $Init(C)$ because of Condition 3 in Definition 3.2 and Condition 2 in Definition 3.4.

Third, if the precedence relation of C were not an order, this would mean a transition $\mathrm{ID}(x)$ of C_2 would precede a transition INST of C_1 that has an entering point of $S(x)$ as result. This is not possible because of Condition 2 in Definition 3.4. □

The second property is *parallel compositionality*, which requires a slight restriction.

Theorem 3.6. Let C_1 and C_2 be two disjoint process computations. If the eigenvariables of C_1 that are free in $Res(C_1)$ are not free in $Res(C_2)$ and *vice versa*, then the union of C_1 and C_2 is a process computation.

Proof. Let C be the union of C_1 and C_2. First, C is a computation structure because C_1 and C_2 are disjoint.

Second, $Init(C) = Init(C_1) \cup Init(C_2)$ and entering points of eigenvariable scopes of C cannot be present in $Init(C)$ because of the restriction on the eigenvariables of C_1 and C_2 in the hypotheses.

Third, the precedence relation of C is the union of the precedence relations of C_1 and C_2 because of the restriction on the eigenvariables of C_1 and C_2 in the hypotheses: no new precedence is created in C. □

[†] We can make the management of process computations easier with an additional condition, which is only technical. This condition is that eigenvariables are not free outside their scope in the computation structure.

https://doi.org/10.1017/S0960129598002655 Published online by Cambridge University Press

4. Measure of complexity and the outcome of computations

4.1. *Time and space complexity of computations*

Our formalism does not provide us with explicit time and space. Moreover, our model contradicts the notion of global time. On the contrary, its interest is to give a geometrical representation of true concurrency. However, the partial order $\prec$ on communications induces notions of space and local time that clearly display their orthogonality. Among the events of a process computation, we take only communications into account to measure the complexity because they are the only ones that are significant for considerations of time and space.

Definition 4.1. A process computation is a *computation in infinite time* if it includes an infinite strictly increasing sequence of communications for the order $\prec$. If not, its *time complexity* is the size of the longest increasing sequence of communications.

The computation of Example 2.8 is a computation in infinite time ; the time complexity of this of Example 2.6 is 1.

It is no surprise that the space complexity of a computation appears as a notion orthogonal to its time complexity. To measure it, we have to consider the maximum number of communications running concurrently. From this, we get the following definition.

Definition 4.2. A process computation is a *computation in infinite space* if it includes an infinite set of independent communications (that is, communications that are not comparable by the relation $\prec$). If not, its *space complexity* is the maximum size that a set of independent communications can have.

The space complexity of the computations of Examples 2.6 and 2.8 is 1.

4.2. *Success, deadlock and divergence of computations*

Let us begin by defining successful computations. First, they must be finite in time and space but this is not sufficient: we have to consider their residue, which here is also the final state. If we take into account the fact that all recursive processes generated in a computation are persistent, we are aware that they are present in the residue. Then the two constants 1 and 0 determine two forms of success for a computation:

— a smooth termination that is expressed by the fact that the only linear processes present in the final state are constants 1; these constants mark the end of each linear process occuring in the computation;
— a sudden termination that is expressed by the presence of the constant 0 in the final state; this special process has the power of killing all other linear processes present in the final state.

This can be formalised as follows.

Definition 4.3. A finite process computation is *successful* if it verifies one of the following conditions:

— its residue includes only recursive processes or constants 1;
— its residue includes the constant 0.

https://doi.org/10.1017/S0960129598002655 Published online by Cambridge University Press

With this definition, the computation of Example 2.6 is successful but that of Example 2.8 fails because it is infinite. Our definition of success for computations is more general than the definition of Kobayashi and Yonezawa, who only consider sudden termination as success (Kobayashi and Yonezawa 1995), and, since they use the right rules of CLL, their terminator is not 0 but its dual $\top$.

From a logical viewpoint, running a computation consists of building a proof net and the computation succeeds when the proof net is complete. So, a successful computation is a proof net. Of course, this proof net does not look like a classical proof net, though the standard form can be easily recovered from it. In the following section, we will show indirectly that our proof nets are equivalent to Girard proof nets (Girard 1987) in the logical fragment of CPL.

Now, the failure of a computation can occur for two reasons, which gives us the following definition.

Definition 4.4. A finite computation is *in deadlock* if it is not successful and if from its residue, considered as the initial state of a new computation, no more communication is possible. An infinite computation is said to *diverge*.
A computation that is in deadlock or diverges is said to *fail*.

5. Sequentialisation of process computations

In the previous section, we claimed that successful process computations are particular proof nets in LL. To establish their equivalence with the classical proof nets, we choose an indirect method involving sequentialising process computations into proofs of the linear sequent calculus. Sequentialising process computations presents another interest: it is a first step for designing a standard transition system for CPL, which is essential for comparing CPL with classical process calculi such as the π-calculus (Milner *et al.* 1992).

5.1. *The sequent calculus of CPL*

The linear formulas we manipulate in CPL have a restricted syntax, so we can particularise and simplify the formal system of the corresponding linear sequent calculus. Since this formal system will be used later as a framework for comparing processes (see Section 7), we consider a larger fragment than necessary for translating computations. It is made of intuitionistic sequents that have the form $\Gamma \vdash P$ where Γ is a process system and P is process. Such sequents are called CPL sequents. With respect to the whole sequent calculus of ILL (see Appendix A), the main restriction is that linear implications have only atomic formulas on the left. Thus, context splitting for the rule $\multimap_L$ is simplified to give the rule $\multimap'_L$ of CPL. Another simplification concerning exponentials is more usual and holds in whole ILL. After these restrictions and simplifications, we obtain the *sequent calculus of CPL* given in Appendix B. The following theorem guarantees the soundness of the transformation and more.

Theorem 5.1. (soundness) If there is a derivation of the CPL sequent $\Gamma \vdash P$ from the CPL sequents $\Gamma_1 \vdash P_1, \ldots, \Gamma_n \vdash P_n$[†] in the sequent calculus of CPL, there is a derivation of

[†] $n = 0$ means that the derivation is without premises, then we call it a proof.

https://doi.org/10.1017/S0960129598002655 Published online by Cambridge University Press

the same sequent from the same premises in ILL and these premises are used in equal numbers in both derivations.

Proof. The proof follows from the fact that all rules of the CPL sequent calculus are derivable in ILL without changing the number of times a hypothesis is used. For each rule of CPL that differs from a rule of ILL, one can easily find a derivation in ILL. □

Following the proof above, we can design other process calculi by particularising the sequent calculus of ILL. The method always consists of merging several inferences of ILL into a single new inference and the level of atomicity and nondeterminism of the resulting calculus is inversely proportional to the average number of merged inferences. That is why, for instance, the granularity of ACL, the calculus of Kobayashi and Yonezawa (1995), is coarser and its nondeterminism is lower than in CPL.

It is to be hoped that process calculi designed in this way are complete with respect to LL. That means that provability in ILL[‡] always corresponds to success of computation. Completeness holds for ACL and the following theorem guarantees it for CPL also.

Theorem 5.2. (conservativity) If the CPL sequent $\Gamma \vdash P$ is provable in the ILL sequent calculus, then it is provable in the CPL sequent calculus.

Proof. We proceed by induction on the structure of proofs in ILL. We are led to distinguish different cases according to the nature of their last inference I. When I has the type $\otimes_R$ and one component of its main formula is atomic or when I has the type $\multimap_L$, $!_L$, $w!_L$ or $c!_L$, the means of recovering a proof in the CPL sequent calculus is not immediate and requires some technical developments. □

In contrast to the other direction, the translation of derivations with premises is not always possible from ILL to CPL because of the finer granularity of ILL with respect to CPL. However, when we use CPL sequents in the form $\Gamma \vdash 1$, the translation of linear derivations with premises is always possible. As we will show in the next paragraph, these derivations represent process computations.

Theorem 5.3. If the CPL sequent $\Gamma \vdash 1$ is derivable from the CPL sequent $\Gamma' \vdash 1$ in the ILL sequent calculus, then it is derivable from the same sequent in the CPL sequent calculus.

Proof. We proceed in the same way as for Theorem 5.2. □

5.2. *From process computations to proofs in the CPL sequent calculus*

By establishing equivalence between provability in the CPL sequent calculus and provability in the ILL sequent calculus for CPL sequents, we are in position to show that successful process computations correspond with Girard proof nets in a fragment of LL. We only have to prove that a successful computation can be translated into some proof of the CPL sequent calculus and that this translation is reversible.

For this, we are guided by the same idea that has been used by Andreoli and Pareschi (1991), Kobayashi and Yonezawa (1995) and Miller (1993) to design their calculi: a proof

‡ Here, provability in ILL is equivalent to provability in LL because of the absence of the constants $\top$ and $\bot$.

https://doi.org/10.1017/S0960129598002655 Published online by Cambridge University Press

in the linear sequent calculus, when read from the bottom to the top, can be viewed as a sequence of state transitions, each transition being represented by an inference in the proof.

Now, notice that transition rules for process computations match with the left rules for the CPL sequent calculus when read from the bottom to the top (see Appendix B); so we can draw up a matching table between the two forms of rules.

CPL process computations	RC	PAR	ALT_1	ALT_2	REC	INST(t)	ID(y)
CPL sequential proofs	$\multimap'_L$	$\otimes_L$	$\&^1_L$	$\&^2_L$	$!'_L$	$\forall_L$	$\exists_L$

The left rules 1_L 0_L do not appear in this table because they express the two forms of termination for a process. The rule $!'_L$ conforms to the fact that a recursive process can always be the starting point of a new linear process. In the dual version of Andreoli and Pareschi (1991), Kobayashi and Yonezawa (1995) and Miller (1993), transition rules match with right rules of CLL sequent calculus.

Noticing that all left rules of the CPL sequent calculus have zero or one premiss, we can sketch the sequential form of successful process computations.

$$\dfrac{\dfrac{\dfrac{}{\Gamma_n \vdash P}}{\vdots}}{\dfrac{\Gamma_1 \vdash P}{\Gamma_0 \vdash P}}$$

$\Gamma_0, \Gamma_1, \ldots, \Gamma_n$ represents the sequence of the states of the process system being computed. The process P does not play any role during the computation, it interferes only at the end through the axiom that terminates the computation. The sequent calculus of CPL presents three kinds of axioms. Let us examine if each one can be interpreted as closing a computation successfully:

— if the axiom has the type id', this means the final state Γ_n has the form $M, !\Gamma$ and includes a linear process M, which is not possible according to the notion of success we have defined.

— if the axiom has the type $1'_R$, this means the final state Γ_n has the form $!\Gamma$ and we are in the case of a smooth termination. It also implies that the process P is equal to the constant 1. For this reason, we fix the value of P to 1.

— if the axiom has the type 0_L, this means the final state Γ_n has the form $0, \Gamma$ and we are in the case of a sudden termination.

In conclusion, successful computations have two sequential forms:

$$\dfrac{\dfrac{\dfrac{}{!\Gamma \vdash 1}\,(1'_R)}{\vdots}}{\dfrac{\Gamma_1 \vdash 1}{\Gamma_0 \vdash 1}} \qquad \dfrac{\dfrac{\dfrac{}{0, \Gamma \vdash 1}\,(0_L)}{\vdots}}{\dfrac{\Gamma_1 \vdash 1}{\Gamma_0 \vdash 1}}$$

In ACL (Kobayashi and Yonezawa 1995), the more restricted definition of success implies

https://doi.org/10.1017/S0960129598002655 Published online by Cambridge University Press

there is only one form for successful computations, which corresponds to the right pattern above.

Now that we have fixed the form of successful sequential computations, we have to establish how to translate concurrent computations into sequential computations. To solve this contradiction, we have to introduce arbitrary sequentiality between independent process states. In this way, we change true concurrency into concurrency by interleaving. The translation applies not just to successful computations but to all computations. For this, we introduce a concept of state for a process system inside a computation. This concept resorts to an order between process states, which is the transposition of the precedence relation between events according to the following rule: in a process computation, a process state P_1 precedes a process state P_2 if P_1 belongs to the source of an event that precedes or is equal to another event that has P_2 in its result.

Definition 5.4. In a process computation C, a *system state* is a set Γ of process states that has the following properties:

(i) all linear processes belonging to Γ are independent;
(ii) all recursive processes that precede an element of Γ belong to C;
(iii) all process states of C are comparable with an element of Γ.

It is easy to show that the initial state of a computation satisfies this definition but not the residue in the general case. If a computation C is in infinite time, $Res(C)$ only represents a partial state of the system. To obtain a complete state, we have to add states of divergent processes. For instance, in Example 2.8, $Res(C)$ reduces to the single process $!\forall x(nat(x) \multimap (1 \& nat(sx)))$, which is not a system state. By adding any one of the process states $nat(0)$, $nat(s0)$, … , $nat(s \ldots s0)$, we obtain a system state, but the system never reaches a final state.

In a computation, an elementary transition from a system state Γ_1 to a system state Γ_2 results from a single transition of a process of Γ_1. If we can translate the computation into a derivation of the CPL sequent calculus, this transition will be represented by an inference of the derivation read from the bottom to the top. If the computation is finite, the sequence of the system transitions from the initial state to the final state will be represented by the whole derivation. This is established by the following theorem. Renaming may be necessary in the final state because of the eigenvariable condition for the rule $\exists_L$ in the sequent calculus. This constraint is only technical: it does not express an essential difference between the sequential presentation and the presentation with nets, it only expresses that the former is more rigid.

Theorem 5.5. If there exists a finite process computation that has Γ as initial state and Γ' as residue, there is a cut-free derivation of the CPL sequent $\Gamma \vdash 1$ from some CPL sequent $\Gamma'' \vdash 1$ in the ILL sequent calculus and Γ'' is obtained from Γ' by renaming of some free variables.

Proof. Let $\mathscr{C}$ be a finite process computation with Γ as initial state and Γ' as residue. According to Theorem 5.1, we only have to prove that there exists a cut-free derivation $\mathscr{D}$ of $\Gamma \vdash 1$ from some $\Gamma'' \vdash 1$ in the CPL sequent calculus, and that Γ'' is obtained from Γ' by renaming some free variables. We proceed by induction on the number n of events in $\mathscr{C}$. The induction depends on the choice of a minimal event in $\mathscr{C}$. Since there are usually

https://doi.org/10.1017/S0960129598002655 Published online by Cambridge University Press

several minimal events in a computation, we can build many derivations from the same computation. All derivations only differ by inference permutations. So, concurrency is indirectly represented using the property of inference permutability, whereas it appears explicitly in nets. □

The translation in the opposite direction is easier and does not need renaming.

Theorem 5.6. If there is a cut-free derivation of the CPL sequent $\Gamma \vdash 1$ from the CPL sequent $\Gamma' \vdash 1$ in the ILL sequent calculus, then there exists a finite process computation that has Γ as initial state and Γ' as residue.

Proof. We consider any cut-free derivation of $\Gamma \vdash 1$ from $\Gamma' \vdash 1$ in the ILL sequent calculus. According to Theorem 5.3, there is a cut-free derivation $\mathscr{D}$ of $\Gamma \vdash 1$ from $\Gamma' \vdash 1$ in the CPL sequent calculus. Then we proceed by induction on the structure of $\mathscr{D}$ to show that there is a process computation $\mathscr{C}$ with Γ as initial state and Γ' as residue. Two cases require more attention, the cases for which the last inference of $\mathscr{D}$ has the type $\forall_L$ or $\exists_L$. □

Corollary 5.7. A process computation with Γ as initial state is successful iff the CPL sequent $\Gamma \vdash 1$ is provable in LL.

Proof. The proof follows immediately from Theorems 5.5 and 5.6 and Definition 4.3. □

Now that we have a linear sequent calculus for CPL we can compare it with other calculi that use the same formalism (Andreoli and Pareschi 1991; Miller 1993; Kobayashi and Yonezawa 1995). Among these, ACL pushes the integration of concurrency into logic the furthest. Whereas ACL uses the right-hand-sided sequent calculus of CLL, CPL uses the left rules of ILL sequent calculus, though there is a duality between the formalisms, so ACL and CPL are very close. However, they present three differences:

— the notion of successful computation is more restricted in ACL because it only takes sudden termination into account and not smooth termination like CPL;
— the use of the operator ! is limited to recursive definitions of processes in ACL whereas it is unbounded in CPL; for instance, recursive processes can be generated during a computation in CPL but this is not possible in ACL;
— the granularity of computations is finer in CPL than in ACL but nondeterminism is greater.

6. Expressiveness of CPL

In this section we address some aspects that illustrate the extent of the expressiveness of CPL.

6.1. *Recursive definitions of processes*

In Section 7 we will give a meaning to the notion of process equality by defining congruences on processes. Assuming this notion, we want to be able to define processes by means of recursive equations of the form $X = P$, where P is a process built from the unknown process X and other atomic processes with the combinators of CPL. The

https://doi.org/10.1017/S0960129598002655 Published online by Cambridge University Press

process $p \otimes !(p \multimap P[p/X])$, where p is a special message that is not present in P, represents a possible solution. Here, we only want to illustrate the suitability of such an approach by means of an example.

Example 6.1. Let us consider a buffer with an input port *in* and an output port *out*. The process X that represents this buffer can be defined recursively by the following equation:

$$X \;=\; \forall x\,(in(x) \multimap (out(x) \otimes X)).$$

A solution of this equation is the process $p \otimes !(p \multimap (\forall x\,(in(x) \multimap (out(x) \otimes p))))$. The message p is used as a token, whose role is to activate the buffer.

6.2. *Synchronous communication*

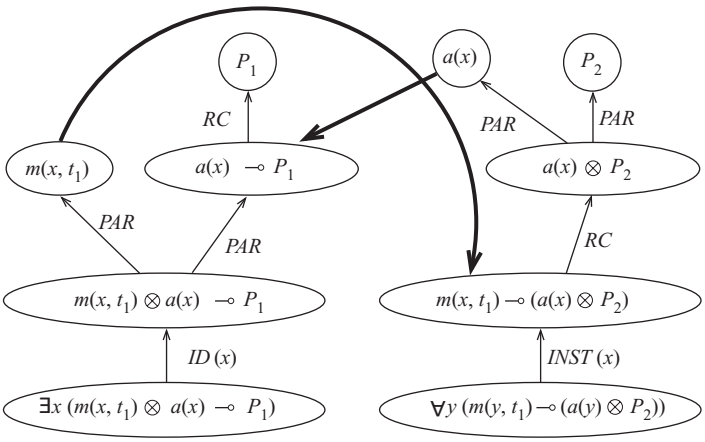
In CPL, communication is asynchronous because a process that sends a message is not stalled until some other process receives this message; the sender and the message each continue on their way independently. P. de Groote suggested using the first-order quantifiers to model synchronous communication in the following way. When a process P_s sends a message, the origin of this message is marked with an eigenvariable x linked to the sender. Then a message can be represented by an atomic formula $m(x,t)$ where t is a term that expresses the content of the message. After sending, the process P_s is stalled waiting for an acknowledgement $a(x)$ marked with the same eigenvariable.

The future receiver P_r of the message $m(x,t)$ is waiting for a message $m(y,t)$ where y represents a universally quantified variable, which can then be replaced by a private identifier such as x. The reception of the message $m(x,t)$ performs this substitution and then the acknowledgement $a(y)$ that P_r will return, become $a(x)$ and then find its addressee.

In brief, a synchronous sender P_s and a synchronous receiver P_r are defined thus:

$$P_s \;=\; \exists x\,(m(x,t) \otimes a(x) \multimap P) \qquad P_r \;=\; \forall y\,(m(y,t) \multimap (a(y) \otimes P)).$$

Now let us represent the computation that expresses a synchronous communication between such a sender and such a receiver.



https://doi.org/10.1017/S0960129598002655 Published online by Cambridge University Press

In this graph we notice that both states P_1 and P_2 of the processes after the communication depend on each other's previous states, whereas in an asynchronous communication the state of the sender after the emission does not depend on the receiver. This corresponds to the fact that the sender is not stalled waiting for a receiver.

It is also easy to see that the acknowledgement message $a(x)$ cannot go to a bad sender because the eigenvariable x is unique and thus can be used as an identifier of the sender.

So, we can restrict the syntax of the processes to obtain a synchronous version of our calculus, which is complete with respect to LL. In this way, contrary to the opinion of Kobayashi and Yonezawa (1996), LL shows its ability to capture synchronous communication: for instance, it can 'distinguish between a process that receives a message m and then n, and a process that receives a message n and then m'. This extends the work of Miller (1993), in which synchronous communication is represented by proper axioms, and the work of Kobayashi and Yonezawa (1995), in which only asynchronous communication is expressed logically.

6.3. *Mobility of processes*

This property is at the source of the π-calculus (Milner *et al.* 1992). A way of proving that CPL can express this property is to translate the π-calculus into CPL. We only wish to show how it is possible with a meaningful example, that of scope extrusion (Milner *et al.* 1992). To simplify the presentation, we consider communication to be asynchronous.

Example 6.2. Let us consider the following transition in the π-calculus:

$$(x)(\overline{y}x.P'|R)|y(z).Q' \xrightarrow{\tau} (x')(P'\{x'/x\}|R\{x'/x\}|Q'\{x'/z\})$$

We have changed the private name x to x' because we assume that x is free in $y(z).Q'$. The initial process can be represented in CPL by the process:

$$P_1 \;=\; \exists x((y(x) \otimes P') \otimes R) \otimes \forall z(y(z) \multimap Q').$$

Since in the π-calculus the only information is names, which are represented in CPL by variables, all messages in the translation of the π-calculus are binary predicates, which we write as $y(x)$ where the first argument y is a channel name and the second argument x is the emitted information or the place for the expected information. Now, the process computation that represents the π-calculus transition mentioned above has P_1 as its initial state and the three following processes as its residue: $P'\{x'/x\}$, $R'\{x'/x\}$, $Q'\{x'/z\}$. This corresponds to the final state in the transition of the π-calculus.

We do not want to go further in the translation of the π-calculus into CPL, which will be the topic of further work.

6.4. *From non-determinism to determinism by syntax restriction*

There are generally many possible computations from a process system because of the many sources of non-determinism in the construction of these computations:

— the choice of P_1 or P_2 from a process $P_1 \& P_2$;

https://doi.org/10.1017/S0960129598002655 Published online by Cambridge University Press

— the number of transitions REC from a recursive process $!P$;
— the term t that instantiates a variable x in a transition INST;
— the choice of a particular receiver from the several that are able to accept the same message.

Principally, CPL is a theoretical framework, and because of the importance of non-determinism, it cannot be used immediately as a programming language. For this, we must reduce the sources of non-determinism mentioned above. A way of doing this is to restrict the syntax of the processes while keeping the expressiveness of the language. We propose to restrict the first three sources of non-determinism significantly.

First, we propose to apply universal quantification to receivers only. A *generalised receiver* R_g is defined by the grammar

$$R_g \ := \ M \multimap P \mid \forall x R_g.$$

The instantiation of the variable x will be made in a communication link by *matching* the sent message with the expected message.

We then propose to restrict the operator & to generalised receivers so that we obtain guarded processes. A *guarded process* P_g is defined by the grammar

$$P_g \ := \ R_g \mid R_g \& P_g.$$

A guarded process waits for a message that activates one of its components associated with a guard that unifies with this message.

The recursion operator ! is only used before a generalised message to express broadcast communication and before a guarded process so that its activation is triggered by the reception of a message and not spontaneously. Finally, the general processes of the restricted CPL language are defined by the following grammar:

$$P \ := \ M \mid !M \mid 1 \mid 0 \mid P_g \mid !P_g \mid P \otimes P \mid \exists x P.$$

It is not surprising that we almost obtain the syntax of ACL (Kobayashi and Yonezawa 1995) in a dual version, but with one important difference: the use of the operator ! is larger in CPL than in ACL.

After this syntax reduction we can increase the granularity of computations by merging some events in new units of computation while preserving the completeness of the calculus with respect to LL. The restricted calculus includes five kinds of event only:

— PAR and ID(x) do not change;
— three new communication units result from merging one or several INST and ALT transitions with a single communication and eventually with a REC transition; if there is no REC transition, the event is a simple communication; if there is a REC transition with the transmitted message as result, the event is a broadcast communication and if there is a REC transition with the receiver of the message as result, the event is the activation of a recursive process.

The restricted calculus remains complete with respect to LL, which means that provability in LL is always expressed by success in computation.

This restriction is not the only possible one, but it illustrates a general method of building a coarser-grained and more deterministic calculus from CPL.

https://doi.org/10.1017/S0960129598002655 Published online by Cambridge University Press

7. Testing equivalences for processes

Testing equivalences for processes (Hennessy and de Nicola 1983) are well suited when we are essentially interested in the outcome of computations and not in their progress. The principle is to make processes interact with a set of testors and then to compare the results of the tests.

Here we apply this idea in a logical framework. Processes are linear logic formulas and we choose the set $\mathscr{P}^{\dagger}$ of processes as a set of testors. Therefore, tests are process computations, *i.e.*, proof nets under construction. In Section 4 we defined the notions of success and failure for computations and we will use them again to characterise the outcomes of tests. So, if, from a logical viewpoint, the construction of the proof net succeeds, the test succeeds, otherwise it fails.

According to the relative importance given to success and failure in the outcome of tests, Hennessy and de Nicola (1983) distinguish three possible equivalences. We use the same classification for the equivalences we will define in the following subsection.

7.1. *Three ways of taking success and failure into account*

Let us begin with some definitions.

Definition 7.1. Let P and T be two processes, the second one being considered as a testor.
P *may satisfy* T if there exists a successful computation with $\{P, T\}$ as initial state.
P *must satisfy* T if no computation with $\{P, T\}$ as initial state ever fails.

It is easy to show that if P must satisfy T, then P may satisfy T.

Example 7.2. $!a$ may satisfy the testor $a \multimap 1$ but $!a$ must not satisfy $a \multimap 1$. On the other hand, $!a$ must satisfy $!(a \multimap 1)$.

In the same way as in Hennessy and de Nicola (1983), we formulate three equivalences as generated by three pre-orders.

Definition 7.3. Let P_1 and P_2 be two processes.

— $P_1 \sqsubseteq_3 P_2$ if, for any testor T, if P_1 may satisfy T, then P_2 may satisfy T;
— $P_1 \sqsubseteq_2 P_2$ if, for any testor T, if P_1 must satisfy T, then P_2 must satisfy T;
— $P_1 \sqsubseteq_1 P_2$ if $P_1 \sqsubseteq_3 P_2$ and $P_1 \sqsubseteq_2 P_2$.

The three corresponding equivalences are denoted $\simeq_3$, $\simeq_2$ and $\simeq_1$. $\simeq_3$ indicates the possibility of tests succeeding simultaneously and $\simeq_2$ indicates the possibility of tests failing simultaneously. $\simeq_1$ is the intersection of $\simeq_2$ and $\simeq_3$. There is no simple relationship between the pre-orders $\sqsubseteq_3$ and $\sqsubseteq_2$, as the following example shows.

Example 7.4. We begin by comparing the processes a and $a\&b$ with the relation $\sqsubseteq_3$. We have $a \sqsubseteq_3 a\&b$, but $a\&b \not\sqsubseteq_3 a$. With relation $\sqsubseteq_2$, we obtain $a \not\sqsubseteq_2 a\&b$ and $a\&b \sqsubseteq_2 a$.
Since the relationship of two processes by $\sqsubseteq_2$ is not always easy to prove, let us show how to do this for the example $a\&b \sqsubseteq_2 a$. Let T be any testor such that $a\&b$ must satisfy T. Let us show that a must satisfy T. We proceed by *reductio ad absurdum* by supposing

† This set depends on the given set $\mathscr{M}(\mathscr{T}[X])$ of messages (see Definition 2.1).

https://doi.org/10.1017/S0960129598002655 Published online by Cambridge University Press

there exists a computation C_1 with $\{a, T\}$ as the initial state and C_1 fails. We add a transition ALT_1 to C_1, the source of which is $a\&b$ and the result is a, and we obtain a computation C_2 that has $\{a\&b, T\}$ as the initial state and that fails. This contradicts the hypothesis that $a\&b$ must satisfy T.

In practice, we do not always need to consider the whole set $\mathscr{P}$ of testors to compare processes. By restricting this set to a subset $\mathscr{T}$, we obtain weaker pre-orders, denoted $\sqsubseteq_i^{\mathscr{T}}$, and weaker equivalences, denoted $\simeq_i^{\mathscr{T}}$. It is interesting to restrict CPL to a fragment where computations are more deterministic. In this way, tests are also more deterministic and the three pre-orders get closer to the three equivalences so that a relationship between them appears. Such a study was made for ACL in Kobayashi and Yonezawa (1993), but it has not yet been done for CPL.

7.2. *Logical expression of a notion of interface*

Here, we are especially interested in the pre-order $\sqsubseteq_3$ and the associated equivalence $\simeq_3$ because they can be expressed by using the resources of proof theory.

Theorem 7.5. The process P may satisfy the testor T iff the sequent $P, T \vdash 1$ is provable in the CPL sequent calculus[†]. Then T is called a *co-process* of P.

Proof. The proof is immediate. □

Our notion of co-process is an adaptation of the notion of co-agent introduced by Miller (Miller 1993) to define a testing equivalence for CCS. Whereas Miller uses this notion in Classical Linear Logic (CLL), we use it in the CPL fragment of ILL.

Example 7.6. Let us consider the program $x := y \parallel y := y + 1$ in some language where $\parallel$ is an operator of parallel composition. In the program, x and y are integer variables. We want to give a semantics to this program by translating it into the following process P:

$$[\forall u \forall v(nat(x,u) \multimap nat(y,v) \multimap (nat(x,v) \otimes nat(y,v)))] \otimes [\forall u(nat(y,u) \multimap nat(y,u+1)].$$

In P, each message is represented by a predicate, the name of which represents the type of its contents (here *nat*), the first argument of which is the name of the used communication channel and the second argument of which is the contents of the message. Communication channels correspond to variables of the program and contents of messages to values of variables.

Here are some co-processes of P:

$$\begin{aligned}
T_1 &= nat(x,0) \otimes nat(y,1) \otimes nat(x,1) \multimap nat(y,2) \multimap 1\\
T_2 &= nat(x,0) \otimes nat(y,1) \otimes nat(x,2) \multimap nat(y,2) \multimap 1\\
T_3 &= nat(x,0) \otimes nat(y,1) \otimes \forall u \forall v(nat(x,u) \multimap nat(y,v) \multimap 1)\\
T_4 &= nat(y,1) \otimes nat(y,2) \multimap (nat(x,0) \otimes nat(y,2) \otimes (nat(x,2) \multimap nat(y,2) \multimap 1))\\
T_5 &= \exists u \exists v(nat(y,v) \otimes nat(y,v+1) \multimap\\
&\qquad (nat(x,u) \otimes nat(y,v+1) \otimes (nat(x,v+1) \multimap nat(y,v+1) \multimap 1))\\
T_6 &= nat(x,0) \otimes nat(y,1) \otimes nat(x,1) \multimap 1 \otimes nat(y,2) \multimap 1.
\end{aligned}$$

[†] This is equivalent to provability in the LL sequent calculus according to Theorem 5.2.

https://doi.org/10.1017/S0960129598002655 Published online by Cambridge University Press

Each co-process expresses one aspect of the possible successful interaction between P and the outside. So, T_1 characterises the fact that P is able to accept the values 0 and 1 as inputs for x and y and return the values 1 and 2 as outputs for the same variables. According to T_2, for the same inputs, the outputs are the values 2 and 2. T_3 expresses a weaker property because it says nothing about the values of the outputs.

To compare processes with the relation $\sqsubseteq_3$, it is interesting to equip them with a denotation so that comparison will be made indirectly on denotations. From this, we get the notion of *process interface*.

Definition 7.7. The interface $\mathscr{I}(P)$ of a process P is the set of all its co-processes.

Now, comparing two processes with the relation $\sqsubseteq_3$ amounts to comparing their interfaces with the relation of set inclusion.

Proposition 7.8. Let P_1 and P_2 be two processes. $P_1 \sqsubseteq_3 P_2$ iff $\mathscr{I}(P_1) \subseteq \mathscr{I}(P_2)$.

This transposition does not immediately bring simplifications: interfaces are infinite sets, which makes testing inclusion very difficult. But we can consider interfaces as objects independent of processes, which can be studied for themselves. By using the resources of proof theory, we will highlight properties and relations on interfaces, and then we will use these to simplify process comparison. Now, we have two parallel worlds – the world of processes and the world of interfaces – and the link between them is made by a relation of satisfiability, which is defined as follows:

Definition 7.9. A CPL interface is a set of processes.
A process P is said to *satisfy* an interface I if all elements of I are co-processes of P.

From the deduction relation in linear logic, we can define an implication relation between interfaces.

Definition 7.10. An interface I_1 *implies* an interface I_2 if for any element T_1 of I_1, there exists an element T_2 of I_2 such that $T_1 \vdash T_2$ is provable in the CPL sequent calculus, that is in LL. We then write $I_1 \Rightarrow I_2$.
If $I_1 \Rightarrow I_2$ and $I_2 \Rightarrow I_1$, then I_1 is said to be equivalent to I_2 and we write: $I_1 \Leftrightarrow I_2$.

It is easy to show that the relation of inclusion for interfaces is stronger than the relation of implication. Then the cut rule of LL provides a property that justifies our interest in this relation.

Theorem 7.11. If an interface I_1 implies an interface I_2, then any process that satisfies I_2, satisfies I_1.

Proof. Let P be a process that satisfies I_2. We consider any co-process $T_1 \in I_1$. Since $I_1 \Rightarrow I_2$, there exists a co-process $T_2 \in I_2$ such that $T_1 \vdash T_2$ is provable. Since P satisfies I_2, by definition, the sequent $P, T_2 \vdash 1$ is provable. And from $T_1 \vdash T_2$, we can infer by the cut rule that $P, T_1 \vdash 1$. Therefore P satisfies I_1. □

Corollary 7.12. A process satisfies an interface I if and only if it satisfies an interface equivalent to I.

One application of the relation $\Rightarrow$ is to simplify an interface I by reducing it, which can be defined formally as follows.

https://doi.org/10.1017/S0960129598002655 Published online by Cambridge University Press

Definition 7.13. An interface I_1 is said to be a *reduction* of an interface I_2 if $I_1 \Leftrightarrow I_2$ and $I_1 \subseteq I_2$.

In practice, to find a reduction of an interface I, we search pairs $\{P_1, P_2\}$ of elements of I such that the sequent $P_1 \vdash P_2$ is provable in CPL and we delete P_1 from I. So, the co-processes T_1, T_2, T_3, T_4, T_5 and T_6 constitute an interface I_1, which reduces to the interface I_2 made up of T_1 and T_5 because the sequents $T_2 \vdash T_5$, $T_3 \vdash T_5$, $T_4 \vdash T_5$ and $T_6 \vdash T_1$ are provable.

In the previous paragraph we saw that we can weaken the relations $\sqsubseteq_3$ and $\simeq_3$ by restricting the set of testors to a subset $\mathscr{T}$ of $\mathscr{P}$. We obtain a pre-order $\sqsubseteq_3^{\mathscr{T}}$ and an equivalence $\simeq_3^{\mathscr{T}}$, which can be characterised in terms of an interface.

Definition 7.14. The interface $\mathscr{I}_{\mathscr{T}}(P)$ of a process *relative to* a set of testors $\mathscr{T}$ is the set of all co-processes of P that belong to $\mathscr{T}$.

Proposition 7.15. Let P_1 and P_2 be two processes and $\mathscr{T}$ be a set of testors.

$$P_1 \sqsubseteq_3^{\mathscr{T}} P_2 \quad \text{iff} \quad \mathscr{I}_{\mathscr{T}}(P_1) \subseteq \mathscr{I}_{\mathscr{T}}(P_2).$$

For computing, equivalences are interesting if they are compatible with the operations of the calculus. Here, all equivalences $\simeq_3^{\mathscr{T}}$ have this property.

Theorem 7.16. If $\mathscr{T}$ is any set of testors, then $\simeq_3^{\mathscr{T}}$ is a congruence.

Proof. Let $\mathscr{T}$ be any set of testors. We have to show that $\simeq_3^{\mathscr{T}}$ is compatible with operations over processes. Let P_1 and P_2 be two processes such that $P_1 \simeq_3^{\mathscr{T}} P_2$. We perform the same operation on both processes, and we have to prove that the resulting processes P_1' and P_2' are such that $P_1' \simeq_3^{\mathscr{T}} P_2'$. According to the type of the operation performed on P_1 and P_2, we have to distinguish different cases. The only problematic case is the one for which the operation is a generalisation. This case requires the following substitution lemma, which is proved by induction on the proofs of the sequent $\Gamma_1[t/x],\ \Gamma_2 \vdash 1$. □

Lemma 7.17. If the sequent $\Gamma_1[t/x],\ \Gamma_2 \vdash 1$ is provable in the CPL sequent calculus and if x is not free in Γ_2 and t, then there exists a process system Γ_3 such that $\Gamma_3[t/x] = \Gamma_2$ and the sequent $\Gamma_1,\ \Gamma_3 \vdash 1$ is provable.

Corollary 7.18. The relation $\simeq_3$ is a congruence.

7.3. *Reduction of a process interface to a trace interface*

The notion of interface reduction provides us with a tool to simplify a given interface, but we do not have a method for building the interface of a given process P: to inspect all elements of $\mathscr{P}$ systematically is unimaginable. A way out is to restrict the set of testors to a subset $\mathscr{T}$ of $\mathscr{P}$ that preserves the meaning of process interfaces. In other words, we are searching for a set $\mathscr{T}$ of testors that has the following property: for any process P, $\mathscr{I}(P) \Leftrightarrow \mathscr{I}_{\mathscr{T}}(P)$. Such a subset exists: the set of *trace processes*. Roughly speaking, a trace process is a sequence of message emissions and receptions (recursion, choice and parallel composition that is not message emission are not present in such processes). These are defined as follows.

https://doi.org/10.1017/S0960129598002655 Published online by Cambridge University Press

Definition 7.19. A *trace process* is a process defined inductively by the following grammar:

$$Tr \ ::= \ 1 \mid M \otimes Tr \mid M \multimap Tr \mid \forall x Tr \mid \exists x Tr.$$

A *trace interface* is a set of trace processes.

Among the co-processes of P in Example 7.6, T_6 is the only process that is not a trace process.

To prove that the interface of any process P reduces to a trace interface, we proceed by induction on the structure of P. We try to build such an interface from the trace interfaces of the components of P. As this construction is a little tedious, we only give the main features. The way of building a trace interface I for P from those of its components depends on the type of operation from which P results.

— P `has the form` $M \multimap P'$, $M \otimes P'$, $\forall x P'$ `or` $\exists x P'$
Let I' be a trace interface that is a reduction of $\mathscr{I}(P')$. We prefix every element of I' with the dual operator of that used to build P from P' and we obtain the sought for interface I. For instance, if $P = M \multimap P'$, then every element of I has the form $M \otimes Tr$, Tr being any element of I'.

— P `has the form` $P_1 \& P_2$
I is obtained by union of the already built trace interfaces of P_1 and P_2.

— P `has the form` $P_1 \otimes P_2$ `with` P_1 `and` P_2 `not atomic`
The construction of I is more complicated. It requires an operation of *interleaving* trace processes, which is defined as follows.

Definition 7.20. Let Tr_1 and Tr_2 two trace processes where all bounded names are distinct and different from the free names of the two processes.
The set $Tr_1 * Tr_2$ of all processes obtained by *interleaving* of Tr_1 and Tr_2 is defined inductively as follows:

- if $Tr_1 = 1$, then $Tr_1 * Tr_2 = \{Tr_2\}$;
- if $Tr_2 = 1$, then $Tr_1 * Tr_2 = \{Tr_1\}$;
- if $Tr_1 = \alpha_1 Tr_1'$ and $Tr_2 = \alpha_2 Tr_2'$ with α_1 and α_2 members of $\{M\otimes,\ M\multimap,\ \forall x,\ \exists x\}$, then $Tr_1 * Tr_2 = \{\alpha_1 Tr \mid Tr \in Tr_1' * Tr_2\} \cup \{\alpha_2 Tr \mid Tr \in Tr_1 * Tr_2'\}$.

Now, the interface I is the set of the trace processes that result from interleaving any process of the trace interface of P_1 with any process of the trace interface of P_2. This operation can explode I but a lot of elements generated in this way are redundant and can be deleted from I.

— P `has the form` $!P'$
The interface I results from the union of the trace interface of the processes 1, P', $P' \otimes P'$, $P' \otimes P' \otimes P'$, ... As in the previous case, the explosion of I is contained by deleting the redundant elements.

Let us illustrate this method by computing the interface of the process P of Example 7.6.

Example 7.21. Let us recall the form of the process P:

$$[\forall u \forall v (nat(x,u) \multimap nat(y,v) \multimap (nat(x,v) \otimes nat(y,v)))]$$
$$\otimes$$
$$[\forall u (nat(y,u) \multimap nat(y,u+1))].$$

https://doi.org/10.1017/S0960129598002655 Published online by Cambridge University Press

Let P_1 and P_2 be the components of P. So: $P_1 = \forall u \forall v(nat(x,u) \multimap nat(y,v) \multimap (nat(x,v) \otimes nat(y,v)))$ and $P_2 = \forall u(nat(y,u) \multimap nat(y,u+1))$. We start by building a trace interface of P_1. As P_1 is a trace process, the method amounts to taking the dual process of P_1.

$\mathscr{I}(nat(y,v)) \Leftrightarrow \{nat(y,v) \multimap 1\}$

$\mathscr{I}(nat(x,v) \otimes nat(y,v)) \Leftrightarrow \{nat(x,v) \multimap nat(y,v) \multimap 1\}$

$\mathscr{I}(nat(y,v) \multimap (nat(x,v) \otimes nat(y,v))) \Leftrightarrow \{nat(y,v) \otimes nat(x,v) \multimap nat(y,v) \multimap 1\}$

$\mathscr{I}(nat(x,u) \multimap nat(y,v) \multimap (nat(x,v) \otimes nat(y,v))) \Leftrightarrow \{nat(x,u) \otimes nat(y,v) \otimes nat(x,v) \multimap nat(y,v) \multimap 1\}$

$\mathscr{I}(P_1) \Leftrightarrow \{\exists u \exists v(nat(x,u) \otimes nat(y,v) \otimes nat(x,v) \multimap nat(y,v) \multimap 1)\}$.

As P_2 is also a trace process, we apply the same method.

$\mathscr{I}(nat(y,u+1)) \Leftrightarrow \{nat(y,u+1) \multimap 1\}$

$\mathscr{I}(nat(y,u) \multimap nat(y,u+1)) \Leftrightarrow \{nat(y,u) \otimes nat(y,u+1) \multimap 1\}$

$\mathscr{I}(P_2) \Leftrightarrow \{\exists u(nat(y,u) \otimes nat(y,u+1) \multimap 1)\}$.

Now, as $P = P_1 \otimes P_2$, the method consists of interleaving every element of the interface of P_1 with every element of the interface of P_2. Before we can do this, to avoid illicit capture of variables, we have to rename the bound variable u of the second interface with a new variable w.

So $\mathscr{I}(\forall u(nat(y,u) \multimap nat(y,u+1))) \Leftrightarrow \{\exists w(nat(y,w) \otimes nat(y,w+1) \multimap 1)\}$.

The result of interleaving is a set of 8 trace processes, but after deleting the redundant elements, it is reduced to the set of the following three processes:

$$\begin{aligned}
&\exists u \exists v \exists w(nat(y,w) \otimes nat(y,w+1) \multimap \\
&\qquad (nat(x,u) \otimes nat(y,v) \otimes nat(x,v) \multimap nat(y,v) \multimap 1) \\
&\exists u \exists v \exists w(nat(x,u) \otimes nat(y,v) \otimes \\
&\qquad (nat(x,v) \multimap (nat(y,w) \otimes nat(y,w+1) \multimap nat(y,v) \multimap 1))) \\
&\exists u \exists v \exists w(nat(x,u) \otimes nat(y,v) \otimes \\
&\qquad (nat(x,v) \multimap nat(y,v) \multimap (nat(y,w) \otimes nat(y,w+1) \multimap 1)))
\end{aligned}$$

In conclusion, $\mathscr{I}(P) \Leftrightarrow I_3$ with I_3 made up of the three processes above. Here, we have summarised all the possible interactions of P that lead to success. If we return to the program $x := y \parallel y := y+1$ of Example 7.6, which was the starting point of our computation, I_3 gives an interpretation to it, but this interpretation goes beyond imperative programming: x, y and z can be viewed as boxes that are able to contain many values at the same time. When the instruction $x := y$ is performed, one value of the box y is read and it takes the place of one value of the box x. When the instruction $y := y+1$ is performed, one value of the box y is replaced with its successor. Then, the operator $\parallel$ means that both instructions can be executed simultaneously or in any order. From this, if we want to recover imperative programming, we initialise every box with a single value. In CPL, this corresponds to a restriction on the set of testors: we consider processes that are able to put one value in exactly each box. In this context, the operator $\parallel$ means that both instructions $x := y$ and $y := y+1$ are atomic and can be performed in any order.

https://doi.org/10.1017/S0960129598002655 Published online by Cambridge University Press

The method we have illustrated through this example is used to establish the following theorem, which justifies our interest in trace interfaces.

Theorem 7.22. For any process P, there exists a trace interface I that is a reduction of its interface $\mathscr{I}(P)$ and that has the same free variables. Moreover, if 0 is not a sub-process of P, then the elements of I only include messages present in P with possibly other bound variables.

Proof. We proceed by induction on the structure of P following the method given in Example 7.21 and distinguishing different cases according to the form of P. Two problematic cases require lemmas: when $P = P_1 \otimes P_2$ (with P_1 and P_2 not atomic) and $P = \forall x P'$. □

7.4. *Relation of logical deduction between processes*

The framework of the CPL sequent calculus naturally provides us with a logical relation between processes: the relation of deduction.

Definition 7.23. A process P_1 is *deducible* from a process P_2 if the sequent $P_2 \vdash P_1$ is provable in the CPL sequent calculus. Then it is denoted $P_1 \sqsubseteq P_2$.

The interest in this relation with respect to the relations $\sqsubseteq_i$ is that verifying the linkage of two processes amounts to proving a theorem in the CPL sequent calculus. Obviously, the relation is a pre-order and the associated equivalence is the logical equivalence, which is denoted $\simeq$. Now it remains to know what this relation expresses by comparing it with the previous ones. Its relationship with $\sqsubseteq_3$ is very simple and it is given by the following theorem.

Theorem 7.24. For any processes P_1 and P_2, if $P_1 \sqsubseteq P_2$, then $P_1 \sqsubseteq_3 P_2$.

Proof. Let us assume $P_1 \sqsubseteq P_2$. This means the sequent $P_2 \vdash P_1$ is provable. To establish $P_1 \sqsubseteq_3 P_2$, we have to prove that every co-process of P_1 is a co-process of P_2. Let T be any co-process of P_1. By definition, the sequent $P_1, T \vdash 1$ is provable. Then by cut with the sequent $P_2 \vdash P_1$, we infer $P_2, T \vdash 1$. That means T is a co-process of P_2. □

Therefore, the relation $\sqsubseteq$ is stronger than $\sqsubseteq_3$. It can be interpreted as a relation of implementation between processes and can find interesting applications in the modelling of some notions used in computer science: refinement of concurrent programs, folding and unfolding in logic programming, and so on.

As specifications and programs are represented in CPL at the same level, the relation $\sqsubseteq$ can be also interpreted as satisfiability of a specification by a program, or refinement of a specification into another.

The relationship between $\sqsubseteq$ and $\sqsubseteq_2$ is more complex than between $\sqsubseteq$ and $\sqsubseteq_3$, as the following example shows.

Example 7.25. Let us consider the processes $!a$ and $!a\&a$. If we compare them with $\sqsubseteq$, we easily obtain $!a \simeq !a\&a$. Now, let us compare them with $\sqsubseteq_2$. If we choose $!(a \multimap 1)$ as a testor, we have $!a$ *must satisfy* $!(a \multimap 1)$ but $!a\&a$ *must not satisfy* $!(a \multimap 1)$. Therefore $!a \not\sqsubseteq_2 !a\&a$. On the other hand, let us show: $!a\&a \sqsubseteq_2 !a$. For this, we consider a testor T such that there exists a computation with $\{!a, T\}$ as initial state, and this computation

https://doi.org/10.1017/S0960129598002655 Published online by Cambridge University Press

fails. By adding a transition ALT_1 with $!a\&a$ as source, we obtain a computation with $\{!a\&a,\ T\}$, as initial state, and this computation fails.

8. Conclusion

In this paper, we have used CPL to draw the main characteristics of a *logic model of concurrency*. This model presents two original features:

— concurrency and communication find a geometrical representation in proof nets;
— testing equivalences based on a notion of interface exploit the power of proof theory.

Behavioural equivalences of processes in CPL remain to be studied. To define bisimulations between processes, we have to express CPL in the form of a transition system. From this, we can compare CPL with more classical process calculi. A translation of the π-calculus into CPL is under way.

The complete framework of CPL is too non-deterministic to be the starting point of realistic applications. For this, it is important to study suitable fragments of CPL. This paper already gives an idea of such fragments but the study must be gone into more closely.

With regard to the specification and the verification of concurrent programs, CPL allows us to express programs and specifications in the same logical formalism, so we can apply the power of logic as a specification language and the power of proof theory as a computational framework to the same objects. As a specification language for concurrency, we still need to compare CPL with other formalisms based on logic, such as temporal logics, modal logics and so on.

Appendix A. The sequent calculus of ILL

Identity group

$$\frac{}{A \vdash A}\,(id) \qquad \frac{\Gamma_1 \vdash F \quad F, \Gamma_2 \vdash G}{\Gamma_1, \Gamma_2 \vdash G}\,(cut)$$

Logical rules

multiplicatives

$$\frac{F_1, F_2, \Gamma \vdash G}{F_1 \otimes F_2, \Gamma \vdash G}\,(\otimes_L) \qquad \frac{\Gamma_1 \vdash F_1 \quad \Gamma_2 \vdash F_2}{\Gamma_1, \Gamma_2 \vdash F_1 \otimes F_2}\,(\otimes_R) \qquad \frac{\Gamma \vdash F}{1, \Gamma \vdash F}\,(1_L) \qquad \frac{}{\vdash 1}\,(1_R)$$

$$\frac{\Gamma_1 \vdash F_1 \quad F_2, \Gamma_2 \vdash G}{F_1 \multimap F_2, \Gamma_1, \Gamma_2 \vdash G}\,(\multimap_L) \qquad \frac{F_1, \Gamma \vdash F_2}{\Gamma \vdash F_1 \multimap F_2}\,(\multimap_R)$$

additives

$$\frac{F_1, \Gamma \vdash G}{F_1 \& F_2, \Gamma \vdash G}\,(\&_L^1) \qquad \frac{F_2, \Gamma \vdash G}{F_1 \& F_2, \Gamma \vdash G}\,(\&_L^2) \qquad \frac{\Gamma \vdash F_1 \quad \Gamma \vdash F_2}{\Gamma \vdash F_1 \& F_2}\,(\&_R)$$

https://doi.org/10.1017/S0960129598002655 Published online by Cambridge University Press

$$\frac{F_1, \Gamma \vdash G \quad F_2, \Gamma \vdash G}{F_1 \oplus F_2, \Gamma \vdash G}\,(\oplus_L) \qquad \frac{\Gamma \vdash F_1}{\Gamma \vdash F_1 \oplus F_2}\,(\oplus_R^1) \qquad \frac{\Gamma \vdash F_2}{\Gamma \vdash F_1 \oplus F_2}\,(\oplus_R^2) \qquad \frac{}{0, \Gamma \vdash F}\,(0_L)$$

exponential

$$\frac{F, \Gamma \vdash G}{!F, \Gamma \vdash G}\,(!_L) \qquad \frac{!\Gamma \vdash F}{!\Gamma \vdash !F}\,(!_R) \qquad \frac{\Gamma \vdash G}{!F, \Gamma \vdash G}\,(w!_L) \qquad \frac{!F, !F, \Gamma \vdash G}{!F, \Gamma \vdash G}\,(c!_L)$$

quantifiers

$$\frac{F[t/x], \Gamma \vdash G}{\forall x\ F, \Gamma \vdash G}\,(\forall_L) \qquad \frac{\Gamma \vdash F[y/x]}{\Gamma \vdash \forall x\ F}\,(\forall_R)$$

$$\frac{F[y/x], \Gamma \vdash G}{\exists x\ F, \Gamma \vdash G}\,(\exists_L) \qquad \frac{\Gamma \vdash F[t/x]}{\Gamma \vdash \exists x\ F}\,(\exists_R)$$

with y not free in the conclusion for the rules $\forall_R$ and $\exists_L$

Appendix B. The sequent calculus of CPL

Identity group

$$\frac{}{M, !\Gamma \vdash M}\,(id') \qquad \frac{\Gamma_1, !\Gamma \vdash P \quad P, \Gamma_2, !\Gamma \vdash P'}{\Gamma_1, \Gamma_2, !\Gamma \vdash P'}\,(cut')$$

Logical rules

multiplicatives

$$\frac{P_1, P_2, \Gamma \vdash P'}{P_1 \otimes P_2, \Gamma \vdash P'}\,(\otimes_L) \qquad \frac{\Gamma \vdash P'}{M,\ \Gamma \vdash M \otimes P'}\,(\otimes_R^1) \qquad \frac{\Gamma \vdash P'}{M,\ \Gamma \vdash P' \otimes M}\,(\otimes_R^2)$$

$$\frac{\Gamma_1,\ !\Gamma \vdash P_1' \quad \Gamma_2,\ !\Gamma \vdash P_2'}{\Gamma_1,\ \Gamma_2,\ !\Gamma \vdash P_1' \otimes P_2'}\,(\otimes_R')$$

with P_1' and P_2' not atomic for the rule $\otimes_R'$

$$\frac{\Gamma \vdash P'}{1, \Gamma \vdash P'}\,(1_L) \qquad \frac{}{!\Gamma \vdash 1}\,(1_R)$$

$$\frac{P,\ \Gamma \vdash P'}{M,\ M \multimap P,\ \Gamma \vdash P'}\,(\multimap_L') \qquad \frac{M, \Gamma \vdash P}{\Gamma \vdash M \multimap P}\,(\multimap_R)$$

additives

$$\frac{P_1, \Gamma \vdash P'}{P_1 \& P_2, \Gamma \vdash P'}\,(\&_L^1) \qquad \frac{P_2, \Gamma \vdash P'}{P_1 \& P_2, \Gamma \vdash P'}\,(\&_L^2) \qquad \frac{\Gamma \vdash P_1' \quad \Gamma \vdash P_2'}{\Gamma \vdash P_1' \& P_2'}\,(\&_R) \qquad \frac{}{0, \Gamma \vdash P'}\,(0_L)$$

exponential

https://doi.org/10.1017/S0960129598002655 Published online by Cambridge University Press

$$\frac{!P,\ P,\ \Gamma\ \vdash\ P'}{!P,\ \Gamma\ \vdash\ P'}\ (!'_L) \qquad \frac{!\Gamma\ \vdash\ P'}{!\Gamma\ \vdash\ !P'}\ (!_R)$$

quantifiers

$$\frac{P[t/x], \Gamma \vdash G}{\forall x\ P, \Gamma \vdash G}\ (\forall_L) \qquad \frac{\Gamma \vdash P[y/x]}{\Gamma \vdash \forall x\ P}\ (\forall_R)$$

$$\frac{P[y/x], \Gamma \vdash G}{\exists x\ P, \Gamma \vdash G}\ (\exists_L) \qquad \frac{\Gamma \vdash P[t/x]}{\Gamma \vdash \exists x\ P}\ (\exists_R)$$

with y not free in the conclusion for the rules $\forall_R$ and $\exists_L$

Acknowledgement

Thanks are due to François Lamarche who reread this article.

References

Abramsky, S. (1990) Computational interpretations of linear logic. 1990 Workshop on Math. Found. Prog. Semantics. *Theoretical Computer Science* **111** 3–57.

Andreoli, J.-M. and Pareschi, R. (1991) Linear objects: Logical processes with built-in inheritance. *New Generation Computing* **9** (3-4) 445–473.

Bellin, G. and Scott, P. J. (1994) On the π-calculus and linear logic. *Theoretical Computer Science* **135** (1) 11–65.

Galmiche, D. and Perrier, G. (1994) On proof normalisation in linear logic. *Theoretical Computer Science* **135** (1) 67–110.

Girard, J.-Y. (1987) Linear logic. *Theoretical Computer Science* **50** (1) 1–102.

Girard, J.-Y. (1989) Towards a geometry of interaction. In: Gray, J. W. and Scedrov, A. (eds.) Proceedings of the AMS-IMS-SIAM Joint Summer Research Conference on Categories in Computer Science and Logic, Boulder, Colorado, June 14–20, 1987. *Contemporary Mathematics* **92** 69–108.

Girard, J.-Y. (1991) Quantifiers in Linear Logic II. In: Corsi and Sambin (eds.) *Proceedings of the Conference on 'Nuovi problemi della logica e della filosopia della scienza', Bologna 1991*, CLUEB 79–90.

Hennessy, M. and de Nicola, R. (1983) Testing Equivalence for Processes. *Theoretical Computer Science* **34** 83–133.

Kobayashi, N. and Yonezawa, A. (1993) Logical, testing, and observation equivalence for processes in a linear logic programming. Technical Report 93-4, Department of Information Science, the University of Tokyo. (Presented at Linear Logic Workshop, Cornell University.)

Kobayashi, N. and Yonezawa, A. (1995) Asynchronous communication model based on linear logic. *Formal Aspects of Computing* **7** (2) 113–149.

Kobayashi, N. and Yonezawa, A. (1996) Roles of Linear Logic in the HACL Project. *Electronic Notes in Theoretical Computer Science* **3**. (Special Issue for Linear Logic'96, Tokyo Meeting, March 1996.)

Lamarche, F. (1996) From Proof Nets to Games (extended abstract). *Electronic Notes in Theoretical Computer Science* **3**. (Special Issue for Linear Logic'96, Tokyo Meeting, March 1996.)

Lincoln, P. and Saraswat, V. (1992) Higher-order, linear, concurrent constraint programming. (Draft paper.)

https://doi.org/10.1017/S0960129598002655 Published online by Cambridge University Press

Miller, D. (1993) The π-calculus as a theory in linear logic: Preliminary results. In: Lamma, E. and Mello, P. (eds.) Proceedings of the 1992 Workshop on Extensions to Logic Programming. *Springer-Verlag Lecture Notes in Computer Science* **660** 242–265.

Milner, R. (1991) The Polyadic π-Calculus: a Tutorial. Technical Report 91-180, University of Edinburgh, ECS-LFCS.

Milner, R., Parrow, J. and Walker, D. (1992) A Calculus of Mobile Processes, I. and II. *Information and Computation* **100** (1) 1–77.

Perrier, G. (1996) Concurrent Programming as Proof Net Construction. Research Report 96-R-132, CRIN-CNRS, Nancy.

https://doi.org/10.1017/S0960129598002655 Published online by Cambridge University Press