Up to context proofs for the π-calculus in the Coq system

Daniel Hirschkoff

janvier 1997

N^o 97-82

Up to context proofs for the π-calculus in the Coq system

Daniel Hirschkoff

Résumé

La formalisation dans le système Coq de la théorie des progressions de relations de Sangiorgi permet, dans son application au π-calcul, la vérification du théorème de preuve au contexte près. Ce résultat s'avère crucial dans le cadre d'une mécanisation du π-calcul, dans la mesure où il facilite considérablement les preuves de bisimulation, en les rendant plus compactes et plus lisibles. S'agissant de notre implémentation du π-calcul en Coq, basée sur une notation de De Bruijn pour l'ensemble des noms de canaux, cela permet de prouver un certain nombre de résultats classiques en théorie algébrique du π-calcul: nous présentons ici les preuves vérifiées des théorèmes d'équivalence structurelle, ainsi que l'unicité des solutions pour les équations.

Abstract

We present a formalisation of polyadic π-calculus in the Calculus of Inductive Constructions. Processes are implemented using a De Bruijn notation for names, and early transitions semantics is represented with an inductively defined relation. We mechanise some bisimulation proofs for the π-calculus using an up-to context technique, which is proved correct within an implementation of Sangiorgi's theory of progressions [20]. This technique, which allows us to shorten the proofs by reducing the size of the relations one has to exhibit, is applied to prove structural equivalence laws, as well as uniqueness of solutions for equations. Possible applications of this work include the proof of other important theorems in π-calculus, as well as the design of a system to check bisimilarities for processes.

Introduction

Implementing a theory into a logical framework can be of interest for many reasons; on the side of the theory, it can help in understanding the objects it describes, and acts as a benchmark for the techniques it deals with; on the side of the target system, it is a test in terms of expressiveness and efficiency. We describe here such a confrontation, focusing mostly on the subject matter of the implementation, namely the theory of polyadic π-calculus, and the proof techniques presented by Sangiorgi in [20]. The logical framework we use is the Coq proof assistant [4]; this work is indebted to Huet's implementation of λ-calculus into Coq [9], through its goals and its implementation style.

The present work follows the one described in [8]; it contributes to study some points that were left obscure (like for example the need for a canonical representation of output actions - see section 1), and provides new results, namely the adaptation of the theory to a non-finite calculus (which compells to adapt the proof presented here in section 3), and the proofs for the structural equivalence theorems.

Π-calculus has become a widely accepted theoretical model for concurrency; in this process algebra, equivalence between terms is often expressed using bisimulation [12, 15, 19]. Sangiorgi's theory of progression of relations [20], a generalisation of the usual methods for bisimulation proofs, has seemed adequate for the task of implementation, in that it is at the same time very general and remarkably tractable in terms of calculation, as will be seen below. Its implementation allows us to prove some non straightforward results of π-calculus theory, like uniqueness of solutions of equations, dramatically relieving the work needed to establish bisimulation properties. While some of our proofs are direct implementations of the ones in [20], we also had to reformulate some demonstrations, still staying within Sangiorgi's framework.

The system we used, Coq [4], is a proof-assistant based on the Calculus of Inductive Constructions, [23], a higher-order logic with inductive definitions. It allows the specification of objects and the verification of properties of these objects through the use of tactics, in a goal-directed manner. The reader interested in a more detailed presentation of the system should refer to [4] and [10].

In the first section, we describe the implementation of the syntax and operational semantics of the set of processes we have chosen, a sum-free polyadic π-calculus with replication and without matching construct, insisting on the important technical consequences of our choice of a De Bruijn representation for names. We then implement in the second section the gen-

eral theory of progressions of relations by Sangiorgi, in order to use it for π-calculus; this is done in the third section, allowing us to prove the up-to-context theorem. We demonstrate how to apply this result in the fourth section, by deriving structural equivalence laws and uniqueness of solutions for equations. We finally conclude discussing related work, and considering possible extensions and improvements of our implementation.

In the following, we suppose that the reader is familiar with usual definitions of syntax and semantics of the π-calculus; this allows us to sometimes state directly the Coq definitions of the objects we need for our proofs, without giving their usual mathematical formulation. We shall not describe rigorously the Coq syntax in this paper, but rather just give hints to understand it along the quotations we make of the Coq specifications (which will be written in `typewriter` style). The reader familiar with functional notation should not have difficulties in following our Coq definitions.

1 π-calculus implementation

1.1 Syntax of processes

We have implemented a polyadic π-calculus with replications[13]. Names are represented using a De Bruijn notation [5]: each name is an index denoting its binding depth in the term, free names being those for which the binding depth is greater than the depth level inside the term (i.e. the number of binders above the name in the term); a description of a De Bruijn notation for monadic π-calculus can be found in [3].

The definitions of types `name` (for names), `l_name` (for name lists) and `pi` (for processes) are as follows:

```
Inductive name : Set := Ref : nat -> name.

Inductive l_name : Set := Nil : l_name
 | Cons : name -> l_name -> l_name.

Inductive pi : Set :=
  Skip : pi                          null process                    0
| Res : pi -> pi                     restriction (monadic)           ν
| Ban : pi -> pi                     replication                     !
| Par : pi -> pi -> pi               parallel composition            |
| Inp : name -> nat -> pi -> pi       abstraction: subject, arity,    λ
                                      and continuation
| Out : name -> l_name -> pi -> pi.  concretion: subject, objet,     [ ]
                                      and continuation
```

The above inductive definitions, when submitted to the Coq system, introduce three new types of sort `Set`, with their respective constructors (`Ref` for names, `Nil` and `Cons` for lists, `Skip`, `Res`, `Ban`, `Par`, `Inp` and `Out` for processes).

We do not treat symmetrically the two binders of the calculus (λ and ν): while abstraction has a real polyadic flavour, denoted by its arity, restriction is still a monadic constructor. This simplifies the definition of the semantics, allowing on one side the simple definition of truly polyadic communication (a list of names is communicated in one step), while keeping on the other side some kind of precision in the manipulation of restrictions, through the use of rules RES and OPEN, as will be seen below.

The choice of a replication constructor instead of recursive definitions was mostly guided by implementation considerations, for the task of simplicity, the "bang" being more simple to handle in Coq.

1.2 Operational semantics

Following [20], we implemented *early* semantics for our processes, by defining a transition relation where names are instantiated as soon as possible in a communication.

Actions are defined as follows:

```
Inductive action : Set :=
  Ain : nat -> name -> l_name -> action
| Aou : nat -> name -> l_name -> action
| Tau : action.
```

An emission or a reception is characterised by its subject (the name where the communication occurs), its object (the list of names that is transmitted), and the number of newly created names it involves (the first argument of constructors `Ain` and `Aou`; note that a natural number suffices for this task, thanks to the De Bruijn notation). Emissions and receptions are treated identically because of our *early* choice for the semantics: when we say that an input occurs, we have to know the whole information that is received (i.e. the object of the communication and the number of new names it carries). `Tau` stands for the silent action, the result of an internal communication. Free and bound names of an action are defined in the usual way.

We can now proceed to the definition of `commit`, the transition relation in Coq, which is an object of type `(pi -> action -> pi -> Prop)`, `Prop` being the sort for propositions in Coq. This is done inductively *à la Prolog*, each constructor of `commit` being a clause:

```
Inductive commit : pi -> action -> pi -> Prop :=
  comm_in : (na:name)(n:nat)(p:pi)
    (k:nat)(l:l_name)(l_length l)=n ->
    (commit (Inp na n p) (Ain k na l)
      (low_subst_pi (lift_pi p n k) l O n))
| comm_ou : (na:name)(l:l_name)(p:pi)
    (commit (Out na l p) (Aou O na l) p)
| comm_op : (na:name)(k:nat)(l:l_name)(x,p:pi)
    (commit x (Aou k na l) p)
    -> ~(na = (Ref O)) -> (occ_n_ln l k)    (* side conditions *)
    -> (commit (Res x)
        (Aou (S k) (low_na na O (S O))
          (roll_ln l k (Nkb_f l k k) O))
        (roll p k (Nkb_f l k k) O))
| comm_re : (x,p:pi) (a:action) (commit x a p)
    -> ~(occ_act a)                         (* side condition *)
    -> (commit (Res x) (lower_action a)
               (Res (swap p (bound_action a) O O)))
| comm_c1 : (x,y:pi)(na:name)(k:nat)(l:l_name)(p,q:pi)
    (commit x (Ain k na l) p) -> (commit y (Aou k na l) q)
    -> (commit (Par x y) Tau (add_nus k (Par p q)))
| comm_pl : (x,p:pi)(a:action) (commit x a p) ->
    (y:pi) (commit (Par y x) a
                   (Par (lift_pi y O (bound_action a)) p))
| comm_ba : (x,p:pi) (a:action)
  (commit (Par (Ban x) x) a p) -> (commit (Ban x) a p).
```

In the above definition, product types can have two notations:

- `(x:T)P` is the notation for *dependent* product, and reads as universal quantification;
- `P -> Q` is the notation for *non-dependent* product, and reads as implication.

`~P` denotes the negation of `P`. Let us now describe the constructors of type `commit`, that implement the rules of early transition semantics.

`comm_in` is the axiom for input actions, expressing the fact that an abstraction of the form `(Inp na n p)` is liable to receive a list of names `l`, the `k` first names occurring in `l` being new, and become its continuation, instantiated by the object of the communication; the hypothesis stating `(l_length l)=n` must be verified, in order to preserve arity (recall that `n` is the arity of the abstraction `(Inp na n p)`). Two operators are applied to the continuation

`p`, namely functions `lift_pi` and `low_subst_pi`: without entering into the tedious details of the index manipulations they involve, let us just say that they insure instantiation for the transmitted list, and management of newly created names.

The OUT rule, for concretions, is much more simple to implement: constructor `comm_ou` just says that a concretion `(Out na l p)` can emit list `l` along `na` and become the continuation `p`. While this emission does not involve private names (hence the `0` in the output action), *name extrusions* are built using the `comm_op` constructor, which is the direct translation of the usual *OPEN* rule of π-calculus semantics: if a term is liable to perform an output action with `k` bound names, then, if we capture one of the free names of this action by applying the restriction constructor to the concretion, the resulting term emits `k`+1 private names (`(S k)` in Coq, `S` being the successor function). In our implementation, the capture of an emitted name by a restriction involves a permutation between names, in order to preserve a kind of "canonical representation for actions". Let us give an example to illustrate this idea; consider processes

$$P_1 = (\nu a)(\nu b)\bar{x}[a,b].P \quad \text{and} \quad P_2 = (\nu b)(\nu a)\bar{x}[a,b].P.$$

We want of course P_1 and P_2 to be bisimilar; however, their De Bruijn translations are something like

$$\mathbf{P_1} = \nu\nu\bar{\mathbf{i}}[\mathbf{1},\mathbf{0}].\mathbf{P} \quad \text{and} \quad \mathbf{P_2} = \nu\nu\bar{\mathbf{i}}[\mathbf{0},\mathbf{1}].\mathbf{P}\{\mathbf{0} \leftrightarrow \mathbf{1}\}$$

($\mathbf{i}$ being the index representing x in our translation, we will see later on that the choice of $\mathbf{i}$ has to be discussed). In the translation of P_2, we have to exchange indexes 0 and 1 in the continuation since we have swapped the restrictions, which is denoted here informally by the $\leftrightarrow$ symbol; the two translated terms are liable to perform two different actions, and hence cannot said to be bisimilar if we use a naive approach.

To tackle this problem, a solution could be to modify the definition of bisimulation, by adding an "α-conversion" mechanism, in order to identify actions (i.e. say that $\nu\nu\bar{\mathbf{i}}[\mathbf{1},\mathbf{0}]$ is the same action as $\nu\nu\bar{\mathbf{i}}[\mathbf{0},\mathbf{1}]$). Another point of view, which is the one we actually chose, is to define a kind of canonical form for output actions, and hence to avoid α-conversion. This allows to stay close to the "De Bruijn doctrine", where α-conversion is directly implemented by the representation of terms, and also to simplify definitions and bisimulation proofs. The idea is to apply a permutation on the `k`+1 emitted private names, in order to always place the outermost bound name as close as possible to the restrictions, as emission is performed. This is

achieved by the `roll` operator, in conjunction with function `Nkb_f`, that computes a parameter of this permutation (we will not enter further the details of this manipulation here). This way, in our example, as rule `comm_op` is used for the emission of private names a and b, names have to be swapped in $\mathbf{P_2}$ in order to satisfy the above condition, and we get identical actions and identical continuations for P_1 and P_2, which ensures bisimilarity.

Along the ideas that are implemented by the `comm_op` constructor, `comm_re` implements the RES rule, the rule used to add a restriction operator to a term, in the case where the restricted name is not involved in the action (side condition `~(occ_act a)`). Here again, a permutation has to take place, in order to bring the restriction that we add as close as possible to the continuation (in other words, the restriction "crosses" the restrictions occurring in the action itself); this is achieved using the `swap` operator, one of the parameters being the number of bound names occurring in the action, computed with the `bound_action` function. This manipulation actually implements in a way a structural equivalence rule, as will be shown below.

Rule `comm_c1`, and its symmetric version (that we omitted here), is used to build a silent transition; its Coq statement should be self-explanatory, knowing that `(add_nus k P)` is the term built by adding `k` restrictions to `P`.

Another possible behaviour of a term built with a parallel construct, apart from internal communication, is when only one subterm performs the action, while the other is simply "watching"; this is implemented by the `comm_pl` rule, and its symmetrical version `comm_pr` (omitted as well). In the classical literature, this rule has a side condition requiring that bound names involved in the action do not appear free in the inactive process, in order to avoid name clash. Within the De Bruijn setting, this is directly implemented into the rule itself, thanks to the `lift` operator, that insures freshness of transmitted bound names by "making room" for new names in the term that is inactive (the `bound_action` function computes the number of new names involved in the action). We stress the point that this mechanism is quite general in the De Bruijn framework, as our implementation testifies.

`comm_ba` is the constructor for actions performed by a replicated term; it is the direct translation of the classical BANG rule, which says that we can get rid of a process when it is put in parallel with its replicated form.

The above paragraphs often allude to operators managing De Bruijn indexes inside a term of type `pi`; explanations about them have been kept vague on purpose, to preserve concision and readability. However, we still feel that entering in some way the detail of the machinery brought by with

the De Bruijn representation is important, because it reveals the influence that such a choice can have on the style of our proofs. As a last hint on this technical aspect of our work, let us just say that a big amount of lemmas have to be proved together with the definition of the operators on indexes.

2 Bisimulations and progressions of relations

We describe here Sangiorgi's techniques for bisimulation proofs, and the questions that arise as we apply them to π-calculus.

2.1 Sangiorgi's techniques

In [20], Sangiorgi describes a theory of relations between elements of a set of *processes* $\mathcal{P}r$, on which a transition relation $\mathcal{T}rans$ included in $\mathcal{P}r \times \mathcal{A}ct \times \mathcal{P}r$ is given ($\mathcal{A}ct$ being a set of *actions*). In the following, we shall write $P \xrightarrow{a} P'$ for $(P, a, P') \in \mathcal{T}rans$. This theory aims at generalising the usual proof techniques for bisimulation, by introducing the notion of *progression of relations*.

Definition 2.1 (Progression of relations) *Given two relations $\mathcal{R}$ and $\mathcal{S}$ on elements of $\mathcal{P}r$, we say that $\mathcal{R}$ progresses to $\mathcal{S}$, written $\mathcal{R} \longrightarrow \mathcal{S}$, if $P\mathcal{R}Q$ implies:*

1. *whenever $P \xrightarrow{\mu} P'$, there is Q' s.t. $Q \xrightarrow{\mu} Q'$ and $P'\mathcal{S}Q'$;*
2. *the converse, i.e., whenever $Q \xrightarrow{\mu} Q'$, there is P' s.t. $P \xrightarrow{\mu} P'$ and $P'\mathcal{S}Q'$.*

Bisimulation and bisimilarity are then naturally defined in terms of progressions:

Definition 2.2 (Bisimulation, bisimilarity) *$\mathcal{R}$ is a bisimulation relation if $\mathcal{R} \longrightarrow \mathcal{R}$ holds; two processes P and Q are bisimilar, written $P \sim Q$ if $P\,\mathcal{R}\,Q$ holds for some bisimulation relation $\mathcal{R}$.*

Sangiorgi then considers what he calls *first-order-functions*, or more simply *functions* (we will adopt this convention in the remainder of the paper), i.e. functions from relations to relations, and states two useful properties about them, namely *soundness* and *respectfulness*:

Definition 2.3 (Soundness) *A function $\mathcal{F}$ is sound if, for any $\mathcal{R}$, $\mathcal{R} \longrightarrow \mathcal{F}(\mathcal{R})$ implies $\mathcal{R} \subseteq \sim$.*

Definition 2.4 (Respectfulness) *A function $\mathcal{F}$ is respectful if whenever $\mathcal{R} \subseteq \mathcal{S}$ and $\mathcal{R} \longrightarrow \mathcal{S}$ holds, then $\mathcal{F}(\mathcal{R}) \subseteq \mathcal{F}(\mathcal{S})$ and $\mathcal{F}(\mathcal{R}) \longrightarrow \mathcal{F}(\mathcal{S})$ also holds.*

The main result of this general theory is that for functions, respectfulness implies soundness. Its proof is easy and translates quite straightforwardly into the Coq system; we just state here the Coq text of a corollary we get from this theorem, which gives a useful technique when we want to prove two terms to be bisimilar:

```
Lemma tech : (p,q:Pr) (R:(relation Pr)) (R p q) ->
  (F:(relation Pr) -> (relation Pr))
  (respectful Pr Act trans F) ->
  (progress Pr Act trans R (F R)) ->
  (bisimilar Pr Act trans p q).
```

Lemma `tech` above says: *"proving two processes P and Q bisimilar amounts to exhibit a relation $\mathcal{R}$ containing (P,Q) and a respectful function $\mathcal{F}$ such that $\mathcal{R} \rightarrow \mathcal{F}(R)$".*

This approach is very general and turns out to be adequate for implementation for many reasons. Let us recall the usual definition of simulation:

$$\text{"}P \sim Q \text{ if } (P \xrightarrow{a} P') \Rightarrow \exists Q'; (Q \xrightarrow{a} Q') \wedge (P' \sim Q')\text{"}.$$

If we consider the above definition from a nearly syntactic point of view, we see that the symbol $\sim$ occurs twice, in the hypothesis and in the conclusion. This introduces circularity in the definition, which is a serious drawback in the practice of bisimulation proofs, in that the relations one has to exhibit to establish bisimulation dramatically increase. To reduce the size of the relations, "up-to" techniques have been introduced [12, 21]. Sangiorgi's alternative formulation of bisimulation, focusing on the function, concentrates in a way the so-called circularity in the function: the function plays the role of actually building the progression, which allows to considerably reduce the size of the relation it is applied to. Such a point of view is naturally adequate for the task of mechanisation of bisimulation proofs, as will be shown below.

Another point we want to stress is that respectfulness is a more interesting property than soundness for our purposes, since it enjoys nice compositional properties. Without entering the details of Sangiorgi's paper (where *second-order functions* are introduced, in order to combine respectful functions), let us just state, as an example of the flexibility and usefulness of

respectful functions, the proof of the bisimulation up to bisimilarity technique.

We start by defining two trivial functions, namely the identity and the constant-to-$\sim$:

```
Definition Ident : relation -> relation := [r:relation]r.

Definition U : relation -> relation := [_:relation]bisimilar.
```

([..] is the Coq notation for abstraction). We then combine these functions using the *chaining* constructor, a function taking two functions as argument and returning a function:

```
Definition chaining : constructor :=
  [F,G:relation->relation][R:relation]
  [P,Q:Pr](Ex [P':Pr] ((F R) P P') /\ ((G R) P' Q)).
```

(Ex is the existential quantifier in Coq, while /\ is the *and* connector). chaining is a *respectful constructor*, in that when applied to two respectful functions as argument, it returns a respectful function. Since Ident and U are trivially respectful functions, function (chaining U (chaining Ident U)) (or with a "mathematical" notation: $\sim \_ \sim$) is respectful as well. This shows the correctness of the bisimulation up to bisimilarity technique in our framework.

2.2 Applying the general framework to π-calculus

The above discussed theory is specialised to π-calculus, through instantiation of the parameters corresponding to the process algebra (pi), the actions (action) and the transition relation (commit); this is achieved in Coq using a Section mechanism, that implements modularity in the specifications. In [20], Sangiorgi has to reformulate the definition of progression in the case of π-calculus, in order to handle possible name clashes. Within the De Bruijn setting, such a side condition is given for free, since the semantics guarantees freshness for extruded names, hence progression has not to be redefined. As a counterpart to this simplification, however, the class of relations we consider has to be restricted, because of the question of representation of free names.

Indeed, when one considers relations between processes, some stability properties have to be guaranteed in order to keep our framework realistic: as seen above, when a subterm of a parallel construct is inactive during a communication, it gets "lifted" in order to manage fresh names introduced

by the communication. If two such terms are in a relation $\mathcal{R}$, their respective "liftings" should reasonably also be in $\mathcal{R}$: this compels to restrict the set of relations over processes we consider, through the definition of a predicate. This predicate says that we are interested in relations that are preserved by injective substitutions on free names (actually, in a very general setting, the manipulations we make on indexes just to guarantee freshness of names can represent any injective substitution), and is defined as follows:

```
Definition good : (pi->pi->Prop)->Prop :=
  [R: (relation pi)] (p,q:pi) (R p q) ->
  (l:l_name) (injective l) ->
  (le (max_nat (max_free_0 p) (max_free_0 q)) (l_length l)) ->
  (R (pi_subst_n p l 0) (pi_subst_n q l 0)).
```

We therefore have to reformulate the general theory of progressions, parametrising it with the `good` predicate (and its derivative `f_good`, insuring that a function preserves the property); actually, we could say that *what is gained on bound names is in some way lost for free names* with the De Bruijn notation. The results, however, are easily adapted to this modification, and we can reformulate the `tech` lemma seen above within the π-calculus setting:

```
Lemma tech : (p,q:pi)
  (R:(relation pi)) (good R) -> (R p q) ->
  (F:(relation pi) -> (relation pi)) (f_good pi good F) ->
  (respectful pi action commit good F) ->
  (progress pi action commit R (F R)) ->
  (bisimilar pi action commit good p q).
```

The `good` condition on the relations we consider actually gives a result from [20] for free, namely the closure under injective substitutions theorem, stating respectfulness for the closure under injective substitutions function. This theorem is actually "built-in" in our implementation; however, we believe that we do not loose expressiveness using our `good` predicate, since weakening the condition on relations could lead to non-realistic situations.

Incidentally, the classical notion of bisimulation is also straightforwardly proved equivalent to the formulation in terms of functions on relations in Coq.

3 The respectfulness theorem

A very useful technique for proving bisimulation properties is the ability to cancel common contexts in two bisimilar terms. To achieve this, we consider the *closure under contexts* function, in a way that is different from

Sangiorgi's, because of our choice of a replication constructor instead of recursive definitions: we work with polyadic contexts (instead of monadic ones), and prove respectfulness for the transitive closure of the closure under contexts of a relation (written $\_^{\mathcal{C}^\top}$).

The implementation of polyadic contexts is similar to the definition of processes; we then define the notion of guarded context: a context is guarded if an occurrence of the hole is under some prefix. This property is useful to get a respectful function when building the closure under contexts of a relation: indeed, when a term performs an input, the names substitutions that are generated influence its behaviour. Hence, when we use a guarded context to build the closure of a relation, we have to consider all possible substitutions of names for the elements of the relation.

Definition 3.1 (closure under contexts function)

$$\mathcal{C}(\mathcal{R}) \quad = \quad \cup_{C \ non\_guarded}\{(C[P], C[Q]) : (P,Q) \in \mathcal{R}\} \quad \bigcup$$

$$\cup_{C \ guarded}\{(C[P], C[Q]) : (P\sigma, Q\sigma) \in \mathcal{R}, \textit{ for all subst. } \sigma\},$$

and in Coq:

```
Inductive Close [R:pi -> pi -> Prop] : pi -> pi -> Prop :=
  Clo_cons : (c:context)(p,q:pi)(close_cons c R p q) ->
    (Close R (c2pi c p) (c2pi c q)).
```

`close_cons` is a function that returns the requirement on objects `R`, `p` and `q` corresponding to the guardness of its first argument (context `c`). Function $\_^{\mathcal{C}^\top}$ is then straightforwardly implemented as function `T_Close` in Coq.

We now turn to the respectfulness theorem:

Theorem 3.2 (up to context bisimulation)

```
Theorem Close_respectful: (respectful pi action commit good T_Close).
```

Proof. For the nontrivial part of this proof, we consider two relations $\mathcal{R}$ and $\mathcal{S}$ such that $\mathcal{R} \subseteq \mathcal{S}$ and $\mathcal{R} \rightarrow \mathcal{S}$, and prove $\mathcal{R}^{\mathcal{C}^\top} \rightarrow \mathcal{S}^{\mathcal{C}^\top}$. We therefore consider a context C, processes P, P' and Q, and an action λ such that $P\,\mathcal{R}^{\mathcal{C}^\top}\,Q$ and $P \xrightarrow{\lambda} P'$. We first perform an elimination on the inductive function that builds the transitive closure. The transitivity case is straightforward; we thus suppose $P\,\mathcal{R}^{\mathcal{C}}\,Q$, and we have to exhibit Q' such that $Q \xrightarrow{\lambda} Q'$ and $P'\,\mathcal{S}^{\mathcal{C}^\top}\,Q'$.

In [20], this proof is achieved by structural induction on the context C (actually, it is only detailed for the application of the general theory to CCS, the proof for π-calculus having the same shape); we have checked that the original proof faithfully translates into the Coq system for the finite case (using monadic contexts). If we want to consider also non finite contexts, however, we rather have to eliminate the inductively defined proposition stating the transition relation $C[P] \xrightarrow{\lambda} P'$, otherwise an induction on C would generate circularity in the proof.

Due to lack of space, we just give as an example the shape of the Coq proof for the case where the transition relation is obtained using the `comm_pl` rule above. The goal to prove is the following:

```
H : (good R)
H0 : (good S)
H1 : (r_incl pi R S)                          (R ⊆ S)
Hind : (q:pi)                                 (induction hypothesis)
        (close_cons c R p q)
         ->(Ex [q':pi]
               (commit (c2pi c q) lambda q')/\(T_Close S p' q'))
Hclose : (close_cons (cPar c' c) R p q)       (induction hypothesis)
H3 : (commit (c2pi c p) lambda p')            (induction hypothesis)
============================
 (Ex [q':pi]                                  (goal to prove)
      (commit (Par (c2pi c' q) (c2pi c q)) lambda q')
      /\(T_Close S
          (Par (lift_pi (c2pi c' p) 0 (bound_action lambda)) p') q'))
```

(`(c2pi c p)` is the Coq translation of $C[P]$). The idea is to use the induction hypothesis `Hind` with process `q` the following way (`(close_cons c R p q)`, which has to be proved in order to use `Hind`, is a direct consequence of `Hclose`):

```
Elim (Hind q).
Intros q0 Hfoo; Elim Hfoo; Clear Hfoo; Intros.
```

This generates the new hypotheses:

```
H4 : (commit (c2pi c q) lambda q0)
H5 : (T_Close S p' q0)
============================
```

We can now exhibit a process satisfying the conditions of the conclusion:

```
Exists (Par (lift_pi (c2pi c' q) 0 (bound_action lambda)) q0); Split.
```

The proof of the left part of the conclusion is done easily using the `comm_pl` rule:

```
Apply comm_pl; Trivial.
```

For the right part of the conclusion, we use the transitivity property, and some basic manipulations lead to the following goal:

```
  H6 : (Close S (c2pi c' p) (c2pi c' q))
  ============================
   (Close S (lift_pi (c2pi c' p) 0 (bound_action lambda))
     (lift_pi (c2pi c' q) 0 (bound_action lambda)))
```

The proof of this subgoal is conducted using hypothesis `H0 : (good S)`, which allows to remove the `lift_pi` operator from the conclusion of the goal, thanks to a lemma stating that this function can be represented by an injective substitution. We conclude this way the proof for this sub case, and work similarly for the other cases. We thus show respectfulness of the $\_^{\mathcal{C}^\top}$ function, which means correctness for the up to context bisimulation technique.

4 Applying the up-to-context proof technique

In this section we derive some classical results of π-calculus theory; after proving some algebraic laws, known as structural equivalence, we turn to a less technical result, namely uniqueness of solutions for equations. The proofs are performed in a remarkably uniform manner, by taking advantage of theorem 3.2, in conjunction with the bisimilarity up to bisimulation proof technique (see section 2): in each case, the method is to exhibit a relation $\mathcal{R}$ and to prove that $\mathcal{R} \rightarrow\!\!\sim\ \mathcal{R}^{\mathcal{C}^\top} \sim$ (this is enough to show bisimulation, thanks to theorem 3.2 and to respectfulness of the chaining operator). With respect to classical proofs, this technique allows to reduce considerably the size of relation $\mathcal{R}$, and hence reduces the number of cases to consider when performing an induction over the transitions of elements of $\mathcal{R}$.

For lack of space, we only sketch the shape of the proofs below, without entering the technical details.

4.1 Structural equivalence results

We prove here some of the bisimilarity properties called *structural equivalence* in [13], or alternatively *π-calculus equivalence* in [1].

Theorem 4.1 (structural equivalence) *We have:*

- *(restriction construct)*
  1. $\forall P, \forall x, y,\ (\nu x)(\nu y)P \ \sim\ (\nu y)(\nu x)P$;
  2. $\forall P, Q, \forall x,\ (\nu x)(P \mid Q) \ \sim\ P \mid (\nu x)Q$ *if* $x \notin fn(P)$;
- *(parallel construct) Parallel composition is symmetrical and associative with respect to bisimulation, and admits* `Skip` *as neutral element.*
- *(replication construct)* $\forall P,\ !P \mid P \ \sim\ !P$.

Proof The proofs for the lemmas for the restriction operator are somewhat tedious, in that they involve heavy De Bruijn indexes manipulation (for result 1, however, work can be factorised by proving a simulation instead of a bisimulation, since the operation that exchanges two names inside a term is an involution). Once we have proved these results, the laws for parallel composition and replication are easily derived, in almost "mathematical" style.

It has to be noted that for these proofs, we do not use the full power of theorem 3.2, as only closure under restrictions is needed. This allows anyway to shorten our proofs: if we were to work without the up-to context technique, this would compell to close every relation `R` we exhibit under replications. As we work with inductively defined relations, this would mean adding a constructor of type `((p,q:pi)(R p q) -> (R (Res p) (Res q)))`. Every time an induction over transitions of elements in `R` is performed, two supplementary cases would have to be considered in our proof (according to the possible transition rules for terms beginning with a restriction).

4.2 Unique solution of equations

We prove here another result, following the method given for CCS in [20]. We first need a lemma:

Lemma 4.2 *Let C be a guarded context; if we have $C[P] \xrightarrow{\mu} P'$, there exists a context C', and a substitution σ such that $P' = (C'[P])\sigma$, and, moreover, for any Q, we have $C[Q] \xrightarrow{\mu} (C'[Q])\sigma$.*

This lemma is proved by induction over the term stating the transition relation $C[P] \xrightarrow{\mu} P'$. We can then state the uniqueness of solutions for equations theorem, in which we work with the *induced congruence*, written $\sim^c$, and defined by "$P \sim^c Q$ if for all substitution σ, $P\sigma \sim Q\sigma$".

Theorem 4.3 (unique solution of equations, proposition 4.14 (2) in [12]) *Suppose C is a guarded context, with $P \sim^c C[P]$ and $Q \sim^c C[Q]$. Then $P \sim^c Q$.*

Proof. The proof boils down to show that $\mathcal{R} = \{(P\sigma, Q\sigma), \sigma \; substitution\}$ progresses to $\sim \mathcal{R}^{\mathcal{C}^\top} \sim$. Very informally, the congruence hypotheses are used for the "$\sim$" part of $\sim \mathcal{R}^{\mathcal{C}^\top} \sim$, while lemma 4.2 is used for the "$\_^{\mathcal{C}^\top}$" part. It has to be noticed that this proof does not use the structural equivalence properties above, and can be conducted in a very formal way, following [20].

5 Conclusion

We have presented an implementation of π-calculus, together with an account of classical results of π-calculus theory. Proofs for these results have been conducted using the up to context technique, which has turned out to be efficient in performing short and elegant proofs. Technical work, however, still represents the biggest part of our implementation, mainly due to the managing of De Bruijn indexes: indeed, as stressed above, the De Bruijn notation, while drastically simplifying work for bound names, requires accuracy in dealing with free names. Of our 700 proved lemmas, about 500 are concerned with operators on free names; the implementation consists of 64 files of Coq specifications and proofs in Coq V6.1 [4].

The work that can be considered as the closest to ours has been done within the HOL system. M. Nesi has defined CCS and a modal logic for this algebra in [16]; regarding π-calculus, the original formalisation by T. Melham [11] has been used as a starting point for the work of O. Aït-Mohamed, which consisted in proving equivalence [1], and building a system to prove bisimilarities for processes interactively [2]. Differences from our implementation and these works arise at the level of technical issues, as well as at the level of the general presentation. Regarding the calculus itself and the way it is implemented, our choice of the De Bruijn notation is a key issue (while in [16], assumptions are made about the avoiding of name clashes, in [11] and [2], renaming has to be done in order to preserve freshness for bound names). Moreover, our choice of the replication construct instead of recur-

sive definitions as in [2], provides a more clean mathematical formulation for the possibly infinite behaviour of a term (replication was also chosen in [11]).

Regarding the approach we had in our work, and from a more theoretical point of view, the implementation of Sangiorgi's theory of progressions and its application to π-calculus tends to be more general and closer to a "mathematical" (as opposed to "technical" or "computational") formulation. Although we cannot get rid of technical work involving De Bruijn indexes manipulation, the general shape of our bisimulation proofs in section 4 turned out to be reasonably simple and clear, and quite remarkably it has been possible to confine in some way the tedious technical work to specific parts of the proof, keeping the backbone of the derivation readable. Moreover, perspectives are quite open regarding results of π-calculus theory we can hope to prove in the future, thanks to the generality of our approach.

Other relevant implementations of π-calculus include the Mobility Workbench [22], a tool for checking open bisimilarities, as well as PICT [17], which is a programming language based on π-calculus, the way ML is based on λ-calculus; PICT is aimed at showing that a real-size language can be constructed on top of a very small π-calculus, supporting functionality in a natural way.

Finally, let us mention as well Giménez work in Coq on coinductive types [7] includes an implementation of CBS; we did not use coinductive types, except in a toy example, to prove that our notion of bisimulation coincides with a formulation in terms of a greatest fix point of a combinator (using the fact that a coinductive definition implicitly implements a greatest fixpoint).

Future work on this implementation should follow two directions. On one side, one could think of proving a rewriting algorithm more or less in the same fashion as in Aït-Mohamed's PIC system [2], and take advantage of Coq's extraction capability to obtain a verified system for checking bisimilarities. It should be of interest for that approach to see how we can exploit our proofs involving the replication operator, a rather original feature with respect to other π-calculus implementations which are designed towards verification.

Another interesting opportunity is the exploitation and development of our proofs in a more theoretical direction. Work has already been done in proving the so-called replication theorems (still with the up-to context proof technique), which should allow to consider the verification of the *functions as processes* paradigm [14], which is a key result often used as a benchmark for theoretical models of concurrency [6], [18].

Acknowledgments

We thank Davide Sangiorgi and René Lalement for enlightening discussions about this work.

References

[1] O. Ait-Mohamed. Vérification de l'équivalence du π-calcul dans HOL. Rapport de recherche 2412, INRIA-Lorraine, Nov. 1994. (In French).

[2] O. Ait-Mohamed. PIC: A proof checker for the π-calculus in Higher Order Logic. Technical report, INRIA-Lorraine, 1995.

[3] S. J. Ambler. A de Bruijn notation for the π-calculus. Technical Report 569, Dept. of Computer Science, Queen Mary and Westfield College, London, May 1991.

[4] B. Barras, S. Boutin, C. Cornes, J. Courant, J. Filliâtre, E. Gimenez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, and B. Werner. *The Coq Proof Assistant Reference Manual*. Projet Coq, INRIA Rocquencourt / CNRS - ENS Lyon, 1996.

[5] N. de Bruijn. Lambda Calculus Notation with Nameless Dummies. In *Indagationes Mathematicae*, volume 34, pages 381–392. 1972.

[6] C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In *POPL 96*.

[7] E. Giménez. *Un calcul de constructions infinies et son application á la vérification de systèmes communicants*. PhD thesis, E.N.S. Lyon, 1996.

[8] D. Hirschkoff. Bisimulation proofs for the π-calculus in the calculus proof assistant. Technical Report 96-62, CERMICS, Noisy-le-Grand, april 1996.

[9] G. Huet. Residual theory in λ-calculus: A formal development. Technical Report 2009, INRIA, Rocquencourt - France, Août 1993.

[10] G. Huet, G. Kahn, and C. Paulin-Mohring. *The Coq Proof Assistant, A Tutorial*. Projet Coq, INRIA Rocquencourt / CNRS - ENS Lyon, Février 1995.

[11] T. F. Melham. A mechanized theory of the π-calculus in HOL. *Nordic Journal of Computing*, 1(1):50–76, 1994.

[12] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[13] R. Milner. The polyadic π-calculus: a tutorial. Technical report, LFCS, Dept. of Computer Science, University of Edinburgh, october 1991.

[14] R. Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[15] R. Milner and D. Sangiorgi. Barbed bisimulation. In LNCS, editor, *Proc. of ICALP*, pages 685–695, 1992.

[16] M. Nesi. A formalisation of the CCS process algebra in higher order logic. Technical Report 278, Computer Laboratory, University of Cambridge, december 1992.

[17] B. Pierce. *Programming in the Pi-Calculus (Tutorial Notes)*. Computer Laboratory, Cambridge - UK, November 1995.

[18] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, University of Edimburgh, 1992.

[19] D. Sangiorgi. A Theory of Bisimulation for the π-calculus. Technical report, LFCS, Department of Computer Science, University of Edinburgh, May 1993.

[20] D. Sangiorgi. On the bisimulation proof method. Revised version of Technical Report ECS–LFCS–94–299, University of Edinburgh, 1994. An extended abstract can be found in Proc. of MFCS'95, LNCS 969, 1994.

[21] D. Sangiorgi and R. Milner. Techniques of "weak bisimulation up to". In S. Verlag, editor, *CONCUR '92*, number 630 in LNCS, 1992.

[22] B. Victor and F. Moller. The Mobility Workbench — a tool for the π-calculus. In D. Dill, editor, *Proceedings of CAV'94*, volume 818 of *Lecture Notes in Computer Science*, pages 428–440. Springer-Verlag, 1994.

[23] B. Werner. *Une théorie des constructions inductives*. Thèse de doctorat, Université Paris 7, 1994.