# A Coq Library for Verification of Concurrent Programs

Reynald Affeldt [a,1] Naoki Kobayashi [b,2]

[a] *Department of Computer Science, University of Tokyo, Tokyo, Japan*

[b] *Department of Computer Science, Tokyo Institute of Technology, Tokyo, Japan*

**Abstract**

Thanks to recent advances, modern proof assistants now enable verification of realistic sequential programs. However, regarding the concurrency paradigm, previous work essentially focused on formalization of abstract systems, such as pure concurrent calculi, which are too minimal to be realistic. In this paper, we propose a library that enables verification of realistic concurrent programs in the Coq proof assistant. Our approach is based on an extension of the $\pi$-calculus whose encoding enables such programs to be modeled conveniently. This encoding is coupled with a specification language akin to spatial logics, including in particular a notion of fairness, which is important to write satisfactory specifications for realistic concurrent programs. In order to facilitate formal proof, we propose a collection of lemmas that can be reused in the context of different verifications. Among these lemmas, the most effective for simplifying the proof task take advantage of confluence properties. In order to evaluate feasibility of verification of concurrent programs using this library, we perform verification for a non-trivial application.

*Key words:* Coq, concurrent programs, $\pi$-calculus

## 1 Introduction

Concurrent programs are ubiquitous: multi-threaded programs in network servers, distributed programs for database applications, etc. In order to guarantee their correctness and security properties, it is important to verify them formally. The main difficulty in formally verifying concurrent programs is the size of their state space. The latter can be very large (because of nondeterminism) and even infinite (for non-terminating applications, such as reactive systems).

[1] Email: affeldt@yl.is.s.u-tokyo.ac.jp

[2] Email: kobayasi@kb.cs.titech.ac.jp

*This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science*
*URL:* www.elsevier.nl/locate/entcs

Proof assistants and model checkers can be regarded as complementary tools for formal verification. Model checkers are fully automated but can only handle finite state space systems (without appropriate abstraction techniques). Proof assistants are interactive but they can handle infinite state space systems directly, using inductive reasoning. In this paper, we are concerned with formal verification based on proof assistants.

Proof assistants have been applied successfully to the formal verification of sequential programs. There exist tools to enable practical verification of imperative programs (e.g., [11]). Proof assistants have been used to verify realistic programs (e.g., [3,24,2]).

Regarding concurrency, previous work using proof assistants has focused on abstract concurrent systems rather than on realistic concurrent programs. There are many formalizations of pure concurrent calculi (e.g., [14,10,15,25,26]) and experiments with the combined use of proof assistants and model checkers for minimal concurrent languages (e.g., [29,19]). This work demonstrates the usefulness of proof assistant-based formal verification for concurrent programs. However, the minimality of formalized calculi and languages makes it cumbersome to verify realistic concurrent programs. Moreover, in view of the large proof developments in previous work, it is even questionable whether such verifications can be done in practice. For these reasons, we think that formal verification of realistic concurrent programs has not yet been addressed satisfactorily.

In this paper, we introduce a library that enables verification of realistic concurrent programs using a general-purpose proof assistant, namely Coq [27]. This library consists of:

- A modeling language with attractive features for verification of realistic concurrent programs. This modeling language is based on the $\pi$-calculus [20] (a foundational language for the study of concurrent systems) but is different from encodings developed in previous work in that it allows Coq datatypes and control structures to be used. Consequently, it makes it easy to model realistic concurrent programs and to run these models using existing virtual machines and compilers.
- A specification language for realistic concurrent programs. In particular, it provides a notion of fairness that is necessary to write satisfactory specifications for realistic concurrent programs.
- A collection of lemmas in order to facilitate formal proof. The most effective lemmas are based on confluence properties. They allow for smaller formal proofs by reducing the state space that needs to be explored for the purpose of verification.

To evaluate the feasibility of verification of concurrent programs using our library, we have performed formal verification of an existing mail server.

**Paper Outline**

We explain the three parts of the library in turn (the modeling language, the specification language and the collection of lemmas) and then report on the case study. We use the syntax of Coq (version 7).

## 2 Modeling Language

In this section, we introduce (a Coq encoding of) a simple concurrent language that can be used to model a wide range of realistic concurrent programs. Simplicity and generality are inherited from the $\pi$-calculus, on which this modeling language is based. Because of its minimality, the (pure) $\pi$-calculus is not well-suited to modeling of realistic concurrent programs. The main reason is that datatypes and control structures (conditionals and functions) need to be encoded by means of the concurrent primitives. Our modeling language addresses this shortcoming by extending the $\pi$-calculus with datatypes and functions, similarly to the Pict programming language [23]. We call our modeling language appl$\pi$, which stands for "applied $\pi$-calculus" [3].

In Sect. 2.1 and Sect. 2.2 we discuss the encoding of the syntax and the operational semantics of appl$\pi$, respectively.

### *2.1 Syntax Encoding*

The syntax of appl$\pi$ consists of *channels* and *processes*. Intuitively, processes perform computations and exchange values with other processes through channels.

Channels are encoded by means of the functional type `chan`. Any type in `Set` can be used as a datatype for communicated values, and channels themselves can be communicated:

```
Axiom chan : Set -> Set.
```

Processes are encoded by means of the inductive type `proc`. Each constructor of `proc` corresponds to a concurrent primitive of the $\pi$-calculus [4]:

```
Inductive proc : Type :=
  zeroP: proc
| inP: (A:Set)(chan A) -> (A -> proc) -> proc
| rinP: (A:Set)(chan A) -> (A -> proc) -> proc
| outP: (A:Set)(chan A) -> A -> proc -> proc
| parP: proc -> proc -> proc
| nuP: (A:Set)((chan A) -> proc) -> proc.
```

[3] We introduce this abbreviation to avoid confusion with Abadi and Fournet's applied $\pi$-calculus [1] which is an extension of the $\pi$-calculus to study security protocols.

[4] The concurrent primitives of appl$\pi$ are more precisely a subset of those of the $\pi$-calculus: replication is restricted to input processes and there is no external choice. These restrictions have little impact on expressiveness, as discussed in [23].

Intuitively, `zeroP` represents the inert process. `(inP c [x:A]P)` represents an input process: it waits for some value `v` of type `A` along the channel `c` and then behaves as process `([x:A]P v)`. `(outP c v P)` represents an output process: it sends the value `v` along the channel `c` and then behaves as process `P`. `(parP P Q)` represents the parallel composition of the processes `P` and `Q`. `(rinP c [x:A]P)` represents replicated input: it waits for some value `v` of type `A` along the channel `c` and then behaves as process `(parP (rinP c [x:A]P) ([x:A]P v))`. The process `(nuP [x:(chan A)]P)` represents channel creation: it creates a new channel `c'` and then behaves as the process `([x:(chan A)]P c')`. Processes are in the `Type` universe so that they cannot be sent as data.

This encoding allows the Coq language to be used as the functional core of applπ. This effect is achieved by *higher-order abstract syntax* (HOAS), an encoding technique used to ease the management of binders. Concretely, process continuations for input and channel creation primitives are taken to be Coq functions. Thus, one can use the Coq language to write applπ processes. Our encoding can be said to be a *deep embedding* because we define the syntax as an inductive type that we use in the next section to define the operational semantics. However, the ability to integrate Coq functions gives it also the flavor of a *shallow embedding*. (See for instance [21] or [25] for definitions.)

The use of dependent types guarantees that channels are used consistently according to their type. For instance, `(inP c [x:A]P)` is rejected by Coq if `c` has not type `chan A`. Without dependent types, we would have to introduce a sum type for values and insert explicit tagging/untagging to perform data emission and reception, what would make modeling in applπ cumbersome. The combined use of HOAS and dependent types makes our encoding different from previous work on encoding of the π-calculus in Coq.

The following definitions are used in the rest of the paper. They represent output and input processes without continuations:

```
Definition OutAtom [A:Set; x:(chan A); v:A] := (outP x v zeroP).
Definition InAtom [A:Set; x:(chan A)] := (inP x [x:A]zeroP).
```

Before discussing the formal operational semantics, we illustrate the practical advantages of applπ as a modeling language.

### *2.1.1 Modeling Realistic Concurrent Programs*

By way of example, we show below how to model a simple client/server program.

The process below represents a simple server. It waits on the channel `i` for a request, more precisely a pair of a natural number and a channel. It computes the successor of the received natural number and sends it back using the received channel:

```
Definition server [i:(chan nat*(chan nat))]:proc :=
(rinP i [ar:?]
  let a = (Fst ar) in let r = (Snd ar) in
  (OutAtom r (plus a (1)))).
```

We observe that it is easy to write realistic programs because our encoding provides us with the Coq language and the Coq standard library (here: `let` construct; `plus`, `Fst`, `Snd` functions).

The process below represents a client for the above server. It sends a request and waits for the answer of the server along a channel it has created. Eventually, it displays the server response along the channel `o`:

```
Definition client [i:(chan nat*(chan nat));o:(chan nat)]:proc :=
(nuP [r:?]
  (parP (OutAtom i ((0),r)) (inP r [x:?](OutAtom o x)))).
```

The parallel composition `(parP (server i) (client i o))` models a simple client/server program. We discuss further modeling issues in our case study.

### *2.1.2 Executing applπ Models*

It is possible to run applπ models with little modification by using the extraction facility of Coq. For instance, the server above can be turned into OCaml code:

```
Coq < Recursive Extraction server.
...
(* various OCaml data structures and functions, including
a datatype for concurrent primitives and the plus function *)
...
let server i =
  RinP (False, i, (fun ar -> OutP (True, (snd ar),
    (plus (fst ar) (S O)), ZeroP)))
```

To run that program using existing virtual machines or compilers, it is sufficient to replace the type constructors for concurrent primitives by OCaml functions with the appropriate semantics. For a sample OCaml module with such functions, see http://web.yl.is.s.u-tokyo.ac.jp/~affeldt/applpi.

This facility can be used to run applπ models as programs on their own. More radically, one can use applπ not as a modeling language but as a programming language, the Coq interface providing static type checking (a la polyadic π-calculus, thanks to our use of dependent types) and OCaml providing an efficient execution environment for formally verified programs.

### *2.2 Operational Semantics Encoding*

The operational semantics of applπ is a relation between processes, which defines what it means for a process to execute actions such as data emission/reception and channel creation. Similarly to the syntax, the operational semantics is borrowed from the π-calculus. More precisely, it is a non-standard labeled transition semantics. Before explaining the encoding, we justify the need for a non-standard semantics.

Our use of HOAS makes it difficult to encode the standard semantics of the π-calculus. The difficulty comes from the fact that ν-bound channels and conditionals are handled at the meta-level in our syntax encoding (respectively by Coq variables and Coq case analysis). For illustration, let us consider the following applπ process:

```
(nuP [x:?](parP (inP x [_:?](OutAtom x v)) (OutAtom x v)))
```

Using a standard semantics, we would expect it to reduce by communication along channel `x` to the process:

```
(nuP [x:?](OutAtom x v))
```

It is difficult to write in Coq a rule to perform such reductions because the processes that are reduced are inside a meta-level λ-abstraction. Honsell et al. [15] solve this problem in their HOAS encoding of the (pure) π-calculus by introducing several artifacts. For instance, their encoding of the standard rule for channel creation requires an additional predicate to check occurrence of a channel in a process (predicate `notin` in the rule `fRES` in [15]). However, such a predicate cannot be defined for applπ because conditionals are represented by Coq case analysis (whereas they are represented by type constructors in [15]).

Our solution is to distinguish between channels already created and channels to be created. For this purpose, instead of considering sole processes, we consider *states*, i.e. pairs of a process with the list of the channels created so far. In a state, channels already created appear in the list and the channels to be created appear as ν-bound channel in the process. (In comparison, both kinds of channels are represented by ν-bound channels in standard semantics.) We denote by `L#P` the state composed of the list `L` and the process `P`, by `nilC` the empty list, and by `&` the addition of an element to a list. The applπ process above is rewritten into the state:

```
nilC#(nuP [x:?](parP (inP x [_:?](OutAtom x v)) (OutAtom x v)))
```

Using our non-standard semantics, it first creates a new channel `x'` to replace `x`:

```
x'&nilC#(parP (inP x' [_:?](OutAtom x' v)) (OutAtom x' v))
```

and then reduces by communication along channel `x'`:

```
x'&nilC#(OutAtom x' v)
```

Concretely, the operational semantics is encoded by means of two inductive

predicates `Trans` and `Redwith`.

`(Trans P l Q)` means that process `P` reduces to process `Q` by performing the elementary action `l` (of type `TrLabel`, representing either data emission, data reception, channel creation or communication). The formal definition of the `Trans` predicate is similar to standard labeled transition semantics except for the rule for channel creation:

```
Inductive Trans: proc-> TrLabel -> proc -> Prop :=
...
| tr_new: (A:Set)(C:(chan A)->proc)(x:(chan A))
        (Trans (nuP C) (NewL x) (C x))
...
```

`(Redwith S l S')` means that state `S` reduces to state `S'` by performing a communication or a channel creation (action of type `RedLabel`). In particular, it captures what it means for a channel to be new (or fresh): simply that it does not appear in the list of channels created so far.

```
Inductive Redwith: state -> RedLabel -> state -> Prop :=
...
| red_new: (L:ChanList)(P,Q:proc)(A:Set)(x:(chan A))
        (Trans P (NewL x) Q) -> (fresh x L) ->
                (Redwith L#P (New x) (x&L)#Q).
```

In the following, when `(Redwith S l S')` is true for some `l`, we write `(Red S S')`. We also write `Reds` for the reflexive, transitive closure of `Red`.

## 3 Specification Language

Specification of concurrent programs deals with questions such as reachability of desirable states. There are several specification languages (or logics) designed for that purpose, such as spatial logics [4] or Dam's $\pi$-$\mu$-calculus [9]. The specification language provided in our library is based on Cardelli and Gordon's spatial logic [4], because we found it expressive enough for our purpose.

Concerning temporal formulas, an important issue developed in our specification language is formalization of strong fairness. Intuitively, strong fairness is a system property enjoyed by execution environments in which communications that can execute infinitely often are eventually scheduled for execution. It is an important assumption without which we cannot write satisfactory specifications for realistic concurrent programs. For instance, let us consider the following program:

```
(parP (parP (OutAtom d v) (inP d [_:?](OutAtom e v)))
      (parP (OutAtom c v) (rinP c [_:?](OutAtom c v))))
```

A property that one might want to check is that the process `(OutAtom e v)` is eventually revealed. However, without the fairness assumption, this property

does not even hold.

We show in Sect. 3.1 how we encode the fairness assumption and in Sect. 3.2 we give the semantics of the formulas of our specification language.

### *3.1 Encoding of the Fairness Assumption*

Fairness is expressed by means of quantifications over runs of concurrent programs. We first explain how we encode runs.

#### *3.1.1 Encoding of Runs*

A run intuitively consists of a maximal sequence of successive reductions.

A *state sequence* is an indexed set of optional states. A *stable* state is a state that cannot evolve anymore.

```
Definition stateSeq : Type := nat -> (optionT state).
Definition Stable [S:state] : Prop := ~(EXT T:state | (Red S T)).
```

A *reduction sequence* is a state sequence such that each state is obtained by a reduction of its predecessor:

```
Definition isRedSeq [PS:stateSeq] : Prop :=
(n:nat)((S:state)(PS n)==(SomeT ? S) ->
        (EXT S':state | (PS (S n))==(SomeT ? S')/\(Red S S')) \/
        (PS (S n))==(NoneT ?)) /\
     ((PS n)==(NoneT ?) -> (PS (S n))==(NoneT ?)).
```

A *maximal reduction sequence* (or a *run*) is a reduction sequence whose last state is stable, or an infinite reduction sequence:

```
Definition isMaxRedSeq [PS:stateSeq] : Prop := (isRedSeq PS) /\
 ((n:nat)(P:state)
   ((PS n)==(SomeT ? P) -> (PS (S n))==(NoneT ?) -> (Stable P))).
```

One may observe that empty sequences are valid runs. In the encoding of formulas, we enforce the condition that a run starts with some state.

#### *3.1.2 Encoding of Fairness*

We formalize the notion of *strong fairness*. Informally, strong fairness says that any process that is infinitely often enabled is eventually reduced [5].

We need a few intermediate definitions. We say that `P` is a subprocess of `Q` when `Q` consists of the parallel composition of `P` with some other process(es). The predicate `(reduced P Q R)` intuitively means `Q` reduces to `R` by reducing its subprocess `P`. The formal definition is omitted for lack of space.

We define what it means for a subprocess to be *enabled* and *eventually reduced*:

[5] *Weak fairness* says that a continuously and infinitely enabled process is eventually reduced. Strong fairness subsumes weak fairness.

```
Definition enabled [P:proc; Q:state] : Prop :=
 (EXT R:state | (reduced P Q R)).
Definition ev_reduced [P:proc; PS:stateSeq] : Prop :=
 (EX n:nat | (EXT S:state | (EXT S':state |
    (PS n)==(SomeT ? S) /\ (PS (S n))==(SomeT ? S') /\
       (reduced P S S')))).
```

We define what it means for a property to hold *infinitely often*:

```
Definition is_postfix [PS',PS:stateSeq] : Prop :=
 (EX n:nat | (m:nat)(PS' m)==(PS (plus m n))).
Definition infinitely_often [p:state->Prop; PS:stateSeq] : Prop :=
 (PS':stateSeq)(is_postfix PS' PS) ->
      (EX n:nat | (EXT S:state | (PS' n)==(SomeT ? S) /\ (p S))).
```

A *fair reduction sequence* is a state sequence such that there is no process that is infinitely often enabled but never reduced:

```
Definition isFairRedSeq [PS:stateSeq] : Prop :=
   (PS':procSeq)(is_postfix PS' PS)->
        (P:proc)(infinitely_often [Q:state](enabled P Q) PS') ->
              (ev_reduced P PS').
```

### *3.2 Available Formulas*

Our specification language consists of a set of logical and spatial formulas (of type `form`) and a set of temporal formulas (of type `tform`). The semantics of formulas is implemented by means of two satisfaction relations (`sat` of type `form->state->Prop` and `tsat` of type `tform->state->Prop`). The explicit distinction between logical and spatial formulas, and temporal formulas is required for the confluence properties introduced in the next section to hold. The informal semantics of basic formulas can be found in Table 1. Observe that we make use of a predicate `Cong` that encodes the standard notion of *structural congruence* (which intuitively relates processes that only differ by spatial rearrangements).

By way of example, we show the formal semantics of the `FMUSTEV` temporal formula. It is defined by quantification over all possible fair runs, as defined in the previous section:

```
Axiom FMUSTEV_satisfaction : (P:state)(f:form)
(tsat (FMUSTEV f) P) <->
 ((PS:stateSeq)(PS 0)==(SomeT ? P) ->
  (isMaxRedSeq PS) -> (isFairRedSeq PS) ->
  (EXT S:state | (EX n:nat | (PS n)==(SomeT ? S) /\ (sat f S)))).
```

In the implementation, the satisfaction relations are axiomatized. This is because the formula for negation does not respect the positivity constraints imposed by Coq. This problem has already been observed in [26]. This is not

| | |
|---|---|
| | Logical Formulas |
| `(sat ISANY S)` | iff `True` |
| `(sat NEG f S)` | iff `~(sat f S)` |
| `(sat OR f g S)` | iff `(sat f S)\/(sat g S)` |
| | Spatial Formulas |
| `(sat INPUTS c f L#P)` | iff `(Cong P (parP (inP c Q) R))` and `(sat f L#(Q v))` for any `v` |
| `(sat OUTPUTS c v f L#P)` | iff `(Cong P (parP (outP c v Q) R))` and `(sat f L#Q)` |
| `(sat CONSISTS f g L#P)` | iff `(Cong P (parP Q R))` with `(sat f L#Q)` and `(sat g L#R)` |
| | Temporal Formulas |
| `(tsat (MAYEV f) S)` | iff for some run, there exists `S'` such that `(Reds S S')` and `(sat f S')` |
| `(tsat (FMUSTEV f) S)` | iff for any fair run, there exists `S'` such that `(Reds S S')` and `(sat f S')` |

Table 1
Basic Formulas

problematic as long as we do not study formally the properties of the formulas.

## 4 Collection of Lemmas

At this point, we are able to write a concurrent program `P`, a (temporal) property `f`, and we can try to prove `(tsat f P)` using Coq tactics. This direct approach is tedious because the Coq native tactics are too low-level and not adapted to the problem at hand. Our solution is to propose a collection of lemmas (and accompanying tactics) to facilitate formal proof.

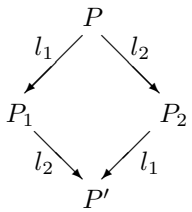
The main difficulty in proving properties of concurrent programs is non-determinism. In order to prove a property for some program, one often needs to check all possible runs. This is at best costly and often impossible because there may be infinitely many runs or because some process is unknown. To deal with these situations, we propose several lemmas based on confluence properties.

In Sect. 4.1, we explain lemmas based on confluence properties and in Sect. 4.2, we give an overview of the whole library.

### *4.1 Confluence Properties*

#### *4.1.1 Basic Idea*

We say that two reductions are *confluent* when they can be executed in either order to reach the same result. More precisely, if $P$ is a process such that $P \xrightarrow{l_1} P_1$ and $P \xrightarrow{l_2} P_2$ are confluent, then for any $P'$ such that $P_2 \xrightarrow{l_1} P'$, we have $P_1 \xrightarrow{l_2} P'$. Graphically, $P$ has the following "diamond property":




Since we know that, no matter the run, $P$ necessarily reduces to $P'$, it is not always necessary to explore both runs to verify a `FMUSTEV` property. This is the basic idea behind lemmas based on confluence properties.

#### *4.1.2 Partial Confluence and Linearized Channels*

In order to identify "diamond properties", we appeal to the notion of partial confluence, which is more general than confluence and often occurs in pratice.

We say that a reduction is *partially confluent* [18] when it is confluent with any other reduction. More precisely, if $P$ is a process such that $P \xrightarrow{l_1} P_1$ is a partially confluent reduction, then for any reduction $P \xrightarrow{l_2} P_2$ and for any $P'$ such that $P_2 \xrightarrow{l_1} P'$, we have $P_1 \xrightarrow{l_2} P'$.

The property of partial confluence is enjoyed by *linearized channels* [18]. A linearized channel is a generalization of a linear channel. It can be used more than once, but only in a sequential manner: an output process `(outP c v P)` can reuse `c` again for output in `P`, an input process `(inP c P)` can reuse `c` again for input in `(P v)` for any `v` of the appropriate type.

We introduce linearized channels in applπ by adding some boolean information to the type of channels `chan`:

```
Axiom chan : Set -> bool -> Set.
```

and by adding a new constructor to the type of processes `proc`:

```
Inductive proc : Type :=
...
| nuP: (A:Set)((chan A false)->proc) -> proc (* non-linearized *)
| nuPl: (A:Set)((chan A true)->proc) -> proc. (* linearized *)
```

The operational semantics is modified accordingly.

For the time being, we assume that linearized channels are correctly annotated. Verification that a process is well-annotated can be done by the type system proposed in [17], Sect. 6.

### *4.1.3 Sample Confluence Property*

The following example is taken from our library:

```
Axiom conf_red_com :
 (L:ChanList; P,P':proc; A:Set; c:(chan A true))
 (well_annotated L#P) ->
 (Redwith L#P (epsilon c) L#P')->
 (f:form)(K:ChanList)(free_chans K f) ->
 ~(in_ChanList c K) ->          (* f does not depend on the     *)
 (M:ChanList)(guard M L#P') -> (* channels that are consumed   *)
 (inter K M nilC) ->            (* or revealed by communication *)
 (tsat (FMUSTEV f) L#P')->
 (tsat (FMUSTEV f) L#P).
```

Intuitively, it says that if `L#P` reduces to `L#P'` by a linearized communication, then in order to prove `(tsat (FMUSTEV f) L#P)`, it is sufficient to prove `(tsat (FMUSTEV f) L#P')` (modulo some conditions that we do not explain here in detail for lack of space).

Currently, these lemmas are axiomatized. They are similar to partial order reduction techniques used in model checking and can be informally justified as such (see for instance [6], Chap. 10).

## *4.2 Library Overview*

The library consists of the applπ language as defined in Sect. 2 (extended with linearized channels), the specification language as defined in Sect. 3, and a collection of lemmas. Although the library is very large (at the time of this writing, 35 proof scripts, 11178 commands for 14951 lines), only a few lemmas are axiomatized (most axioms have actually been discussed in this paper). Not all lemmas are equally important.

During formal proof, the most important lemmas are those that simplify the goal. For instance, confluence properties such as the one seen above are such lemmas: they basically act by simplifying the process that appears in the goal. Similarly, the properties of the formulas of the specification language (distributivity laws, etc.) act by simplifying the formula that appears in the goal.

There are a large number of lemmas that are not intended to be used directly during formal proof but that are very important because they are ubiquitously used to prove other lemmas. Such technical lemmas prove properties about the applπ language (injection, inversion) whose proofs are not immediate because of our use of dependent types, and properties about structural congruence (structural congruence is a bisimulation, etc.).

See http://web.yl.is.s.u-tokyo.ac.jp/~affeldt/applpi for details.

## 5 Case Study

We evaluate feasibility of verification of concurrent programs using our library. Our case study is the SMTP receiver part of an existing mail server. In short, this program receives and processes SMTP commands, sends back SMTP replies and queues received electronic mail.

We chose this application for the purpose of comparison. Indeed, we have already performed verification of this application in Coq using a different approach that consists of building a faithful functional model [2]. In short, the original Java implementation was turned into a Coq function using monadic style programming, third-party programs (client and file system) were modeled using Coq predicates and non-software aspects were modeled using functional constructs (for instance, non-deterministic system failures were modeled using infinite lists to serve as test oracles). Arguably, this approach has little overhead because it takes advantage of the Coq built-in support for functional programs. Therefore, comparison should highlight the overhead of using our library for verification.

In the following, we first explain how we model the mail server using our library, and then we comment on the formal proof that it correctly implements the SMTP protocol.

See http://web.yl.is.s.u-tokyo.ac.jp/~affeldt/applpi for details.

### Modeling of the Main Program

The mail server is modeled as a process (`work`) that is itself the parallel composition of several subprocesses that handle incoming SMTP requests (`get_helo_def`, etc.). The state of the application is reified as a data structure that is communicated from one subprocess to the other. The flow of communication reproduces the flow of control of the Java program. Subprocesses correspond to the Java methods that (are supposed to) implement the SMTP protocol. The reified state corresponds to fields of the server object:

```
Definition work
   [c1:InputStream;c2:OutputStream;tofs:ToFileSystem]:proc :=
let st = initial_state in
   (nuPl [heloc:(chan STATE ?)]
   (nuPl [mailc:(chan STATE ?)]
   (nuPl [rcptc:(chan STATE ?)]
        (parP (rinP heloc (get_helo_def heloc mailc))
        (parP (rinP mailc (get_mail_def mailc rcptc))
        (parP (rinP rcptc (get_rcpt_def mailc rcptc))
        (OutAtom heloc st)))))).
```

Modeling benefits from the fact that applπ is based on the π-calculus. The connections between the mail server and third-party programs (client and file-system) are modeled using channels (instead of sockets in the original Java

implementation) that are aggregated into the reified state and "move" around with the state during computation. Acknowledgments are modeled by a typical $\pi$-calculus idiom: a fresh channel is sent to receive the acknowledgment on it.

### Modeling of Third-party Programs

Third-party programs are modeled by Coq predicates. For instance, the client is modeled by predicates (`speaks_valid_protocol` and `acknowledges_replies`) that implement the SMTP protocol as defined in RFC 821.

### Modeling of System Errors

System errors are modeled by channels. A system failure (resp. a network error) is modeled by outputting some value along the channel `system_failure_chan` (resp. `IOexn_chan`). Since non-determinism is inherent to appl$\pi$, we can model non-deterministic system failures by a process:

```
Definition may_fail :=
(nuP [x:?](parP (InAtom x)
          (parP (OutAtom x tt)
                (inP x [_:?](OutAtom system_failure_chan tt))))).
```

This is more elegant than infinite lists that serve as test oracles in the functional model discussed above. Indeed, the process `may_fail` is clearly separated from the model of the main program, so that we can extract an ML program for the server without pollution from the modeling of system errors.

### Formal Proof

We have formally proved that the parallel composition of the mail server, a valid client, a valid file-system, and a non-deterministic system failures generator ends up with a successful termination (modeled by channel `result_chan`, similarly to system errors), a system failure or a network error (formula `reports_succ_or_error`):

```
Definition reports_succ_or_error : form :=
 (OR (OUTPUTS result_chan tt ISANY)
   (OR (OUTPUTS IOexn_chan tt ISANY)
     (OUTPUTS system_failure_chan tt ISANY))).

Goal
  (Client:InputStream->OutputStream->proc)
    (s:InputStream)(y:OutputStream)(valid_client (Client s y))->
  (file_system:ToFileSystem->proc)
    (tofs:ToFileSystem)(valid_fs tofs (file_system tofs)) ->
  (* channels for termination detection are distinct *)
  (is_set result_chan&(IOexn_chan&(system_failure_chan&nilC)))->
(tsat (FMUSTEV reports_succ_or_error)
```

```
(result_chan&(IOexn_chan&(system_failure_chan&nilC)))#
(nuPl [s1:InputStream]
(nuPl [s2:OutputStream]
(nuPl [tofs:ToFileSystem]
    (parP (Client s1 s2) (parP (file_system tofs)
    (parP (work s1 s2 tofs) may_fail)))))).
```

Verification using our library requires 3927 commands. This is large compared to the 1059 commands required by the verification using the functional model. However, there are several ways to reduce the size of the proof. In particular, we used for verification a confluence property that is weaker (but easier to use in practice) than the one presented in Sect. 4.1. Also, it should be observed that the appl$\pi$ model is more satisfactory than the functional model in many respects: the `work` process takes multi-threading into account and it can easily be run as an ML program, which was not the case of the functional model.

## 6 Conclusion

In this paper, we proposed a Coq library to verify realistic concurrent programs. We have formalized a modeling language based on the $\pi$-calculus that is convenient to write and run (models of) realistic concurrent programs. We have introduced a specification language based on spatial logics extended with the notion of strong fairness. In order to facilitate formal proof, we have built a collection of lemmas among which confluence properties of the modeling language significantly simplify proofs. We have evaluated the feasibility of our approach by verifying a non-trivial application using our library.

### Related Work

There exist several formalizations of pure concurrent calculi in proof assistants. In Coq, Hirschkoff proposes a first-order abstract syntax encoding of the $\pi$-calculus and formalizes proof techniques [14]; Despeyroux formalizes a proof of subject reduction for the $\pi$-calculus [10]; Honsell et al. formalize the foundational paper on the $\pi$-calculus [15]; Scagnetto and Miculan formalize the ambient calculus (a derivative of the $\pi$-calculus) and its spatial logic [26]. In Isabelle, Röckl et al. propose a HOAS encoding of the $\pi$-caculus and formalize the Theory of Contexts [25]. This work focuses on the formalization of the theory of pure concurrent calculi. In contrast, we are concerned with verification of realistic concurrent programs and we aim at building a practical library for that purpose.

The verification of concurrent programs using proof assistants is also addressed using the UNITY formalism. In particular, there exist several formalizations of compositional reasoning (e.g., [13,22]) that is useful to tackle realistic examples. Our work is complementary: we use the $\pi$-calulus as an

underlying formalism and therefore we can benefit from known analyses to formalize additional proof techniques (e.g., lemmas based on confluence properties).

Watkins et al. propose a logical framework [28] with built-in facilities for reasoning about concurrency. Using this concurrent logical framework (CLF), Cervesato et al. encode several concurrent systems [5], including the $\pi$-calculus. Although the authors have not addressed directly the issue of verification of realistic concurrent programs, it seems that an implementation of CLF may ease the development of a library similar to ours.

Coupet-Grimal [7] proposes an encoding of linear temporal logic in Coq. Temporal formulas are defined for an abstract transition system and their properties are collected into a library that has been used to prove correctness of a garbage-collection algorithm [8]. It would be useful to integrate similar reasoning on temporal formulas for our language.

**Future Work**

As stated in Sect. 4.1, we assume that channels are annotated so as to reflect partial confluence. In order to verify that channels are correctly annotated, we plan to formalize an adequate type system inside Coq, similarly to the work by Gay [12] who formalizes the type system of Kobayashi et al. [18] in Isabelle. We also plan to provide mechanical proofs for the lemmas based on confluence properties that are axiomatized for the time being.

We have been formalizing new formulas (such as fixed points) to enhance expressiveness but they are not yet integrated in the library.

In order to reduce the size of formal proofs, we are improving automation and investigating additional proof techniques based on other type systems, such as Kobayashi's type system for lock-freedom [16].

## References

[1] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *Proceedings of the 28th ACM Symposium on Principles of Programming Languages*, pages 104–115, Jan. 2001.

[2] Reynald Affeldt and Naoki Kobayashi. Formalization and verification of a mail server in Coq. In Mitsuhiro Okada, Benjamin Pierce, Andre Scedrov, Hideyuki Tokuda, and Akinori Yonezawa, editors, *International Symposium on Software Security, Tokyo, Japan, November 8–10, 2002*, volume 2609 of *Lecture Notes in Computer Science*, pages 217–233. Springer, Feb. 2003.

[3] Paul E. Black. *Axiomatic Semantics Verification of a Secure Web Server*. PhD thesis, Department of Computer Science, Brigham Young University, Feb. 1998.

[4] Luca Cardelli and Andrew D. Gordon. Anytime, anywhere: modal logics for mobile ambients. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on principles of programming languages*, pages 365–377, 2000.

[5] Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-02-102, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Mar. 2002. Revised May 2003.

[6] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.

[7] Solange Coupet-Grimal. An axiomatization of linear temporal logic in the calculus of inductive constructions. *Journal of Logic and Computation*, 13(6):801–813, 2003.

[8] Solange Coupet-Grimal and Catherine Nouvet. Formal verification of an incremental garbage collector. *Journal of Logic and Computation*, 13(6):815–833, 2003.

[9] Mads Dam. *Logic for Concurrency and Synchronisation*, chapter Proof Systems for the pi-calculus Logics. Kluwer Academic Publishers, 2003.

[10] Joëlle Despeyroux. A higher-order specification of the $\pi$-calculus. In Jan van Leeuwen, Osamu Watanabe, Masami Hagiya, Peter D. Mosses, and Takayasu Ito, editors, *International Conference IFIP TCS 2000*, volume 1872 of *Lecture Notes in Computer Science*, pages 425–439. Springer, Aug. 2002.

[11] Jean-Christophe Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, Mar. 2003.

[12] Simon J. Gay. A framework for the formalisation of pi calculus type systems in Isabelle/HOL. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics, Edinburgh, Scotland, UK*, volume 2152 of *Lecture Notes in Computer Science*, pages 217–232. Springer, Sep. 2001.

[13] Barbara Heyd and Pierre Crégut. A modular coding of UNITY in Coq. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1125 of *Lecture Notes in Computer Science*, pages 251–266. Springer, Aug. 1996.

[14] Daniel Hirschkoff. *Mise en œuvre de preuves de bisimulation*. PhD thesis, École Nationale des Ponts et Chaussées, 1999.

[15] Furio Honsell, Marino Miculan, and Ivan Scagnetto. $\pi$-calculus in (co)inductive type theory. *Theoretical Computer Science*, 253(2):239–285, Feb. 2001.

[16] Naoki Kobayashi. A type system for lock-free processes. *Information and Computation*, 177(2):122–159, Sep. 2002.

[17] Naoki Kobayashi. Type systems for concurrent programs. In *Proceedings of UNU/IIST 10th Anniversary Colloquium, March 2002, Lisbon, Portugal*. Springer-Verlag, 2002. Tutorial.

[18] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the Pi-Calculus. In *Proceedings of ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages (POPL'96)*, pages 358–371. ACM Press, 1996.

[19] Panagiotis Manolios. *Mechanical Verification of Reactive Systems*. PhD thesis, The University of Texas at Austin, Department of Computer Sciences, Austin, TX, Aug. 2001.

[20] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100(1):1–77, Sep. 1992.

[21] Tobias Nipkow, David von Oheimb, and Cornelia Pusch. $\mu$Java: Embedding a programming language in a theorem prover. In Friedrich L. Bauer and Ralf Steinbrüggen, editors, *Foundations of Secure Computation*, volume 175 of *NATO Science Series F: Computer and Systems Sciences*, pages 117–144. IOS Press, 2000.

[22] Lawrence C. Paulson. Mechanizing a theory of program composition for UNITY. *ACM Transactions on Programming Languages and Systems*, 23(5):626–656, 2001.

[23] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[24] Benjamin C. Pierce and Jérôme Vouillon. Specifying a file synchronizer (full version). Draft, Mar. 2002.

[25] Christine Röckl, Daniel Hirschkoff, and Stefan Berghofer. Higher-order abstract syntax with induction in Isabelle/HOL: Formalizing the pi-calculus and mechanizing the theory of contexts. In *FOSSACS'01*, number 2030 in Lecture Notes in Computer Science. Springer, 2001.

[26] Ivan Scagnetto and Marino Miculan. Ambient calculus and its logic in the calculus of inductive constructions. In Frank Pfenning, editor, *Electronic Notes in Theoretical Computer Science*, volume 70. Elsevier Science Publishers, 2002.

[27] The Coq Development Team, LogiCal Project. *The Coq Proof Assistant, Reference Manual*. INRIA, 2002.

[28] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Mar. 2002. Revised May 2003.

[29] Shen-Wei Yu. *Formal Verification of Concurrent Programs Based on Type Theory*. PhD thesis, Department of Computer Science, University of Durham, Oct. 1998.