Vol. 46 No. SIG 1(PRO 24) IPSJ Transactions on Programming Jan. 2005

Regular Paper

Verification of Concurrent Programs Using the Coq Proof Assistant: A Case Study (Preprint)

Reynald Affeldt ,[†] Naoki Kobayashi [††]
and Akinori Yonezawa[†]

We show how to model and verify a concurrent program using the Coq proof assistant. The program in question is an existing mail server written in Java. The approach we take is to use an original library that provides a language for modeling, a logic, and lemmas for verification of concurrent programs. First, we report on the modeling of the mail server. Using the language provided by the library, we build a model by (1) translating the original program and (2) building appropriate abstractions to model its environment. Second, we report on the verification of a property of the mail server. We compare this library-based approach with an alternative approach that directly appeals to the Coq language and logic for modeling and specification. We show that the library-based approach has many advantages. In particular, non-functional aspects (communications, non-determinism, multi-threading) are handled directly by the library and therefore do not require complicated modeling. Also, the model can be directly run using existing compilers or virtual machines, thus providing us with a certified implementation of the mail server.

1. Introduction

Mechanical verification of programs is important to guarantee their correctness. Among the tools for such verifications, proof assistants are particularly attractive, because they combine inductive reasoning with automation, what makes them more widely applicable, compared to fully automated tools such as model checkers[7].

Proof assistants cannot in general be used directly for program verification. The main reason is that programs may use programming constructs the proof assistant is unaware of. For instance, verification of concurrent programs in a proof assistant based on the λ-calculus requires an additional machinery to handle typical concurrency concepts such as non-determinism.

We have been developing a library[2] to enable mechanical verification of concurrent programs in the Coq proof assistant. This library (called applπ, which stands for "applied π-calculus") provides a modeling language, a specification language, and lemmas for verification of realistic concurrent programs.

This paper reports on the modeling and verification of a concurrent program using the applπ library. More precisely, we model an existing mail server[17] and we verify that it correctly handles requests from clients. In fact, we have already performed this verification, but using a different approach[1] that directly appeals to the Coq language and logic for modeling and verification. Our main contribution is to show the advantages of using the applπ library for the verification of concurrent programs. In particular, we illustrate that modeling is simplified because typical concurrency concepts (communications, non-determinism, multi-threading) are handled by the library and does not require complicated modeling. In addition, it is possible to run the model using existing compilers or virtual machines, thus providing us with a certified implementation of the mail server. We believe that these advantages reinforce confidence in the verification.

This paper is organized as follows. In Section 2, we give an overview of the Coq proof assistant and the applπ library. In Section 3, we describe our case study. In Section 4, we explain how we model the original program into a concurrent model written with the applπ library. In Section 5, we explain how we model the environment of the program. In Section 6, we report on the mechanical verification in itself. In Section 7, we conclude and list related and future work.

† Department of Computer Science, the University of Tokyo
†† Department of Computer Science, Tokyo Institute of Technology

2. Preliminaries

2.1 The Coq Proof Assistant

The Coq proof assistant is the implementation of a typed λ-calculus (namely the Calculus of Inductive Constructions[12]) that can be used to represent datatypes, functions, and predicates, and therefore encode, among others, computer languages and proof systems. In this paper, we use Coq for both implementation and notation. In this section, we give an overview of the Coq proof assistant. Intuitively, it can be thought as an ML-like language with a rich type system.

In Coq, datatypes are represented by inductive types. For example, natural numbers are defined as follows:

```
Inductive nat : Set :=
  O : nat
| S : nat -> nat.
```

This definition introduces the type of natural numbers `nat`, which is itself of type `Set`, a Coq built-in type for datatypes. `O` and `S` are the constructors for natural numbers; observe that `S` is functional. The intent is that the constructor `O` represents the natural 0, `(S O)` represents the natural 1, etc.

Records are represented by a syntax similar to most programming languages. For example, two-dimensional points can be defined as follows:

```
Record point : Set := pt{
 x: nat;
 y: nat}.
```

`pt` is the constructor for two-dimensional points, and `x` and `y` are their projection functions.

Functions are represented by λ-abstractions. For example, the function that computes the predecessor of a natural number can be defined by case analysis as follows:

```
Definition pred [n:nat] := Cases n of
  O => O
| (S m) => m
end.
```

where `[n:nat]` is the Coq syntax for $\lambda n{:}nat$.

Predicates are represented by inductive types or functions whose resulting type is the Coq built-in type `Prop`. For example, the following predicate defines even natural numbers:

```
Inductive even : nat -> Prop :=
  base : (even O)
| step : (n:nat)(even n) -> (even (S (S n))).
```

The constructor `base` represents the fact that `O` is an even number, and the constructor `step` let us construct, from any even number `n`, the proof that the natural `(S (S n))` is an even number. Intuitively, `(n:nat)` can also be thought as universal quantification.

Proof goals are stated by the keyword `Lemma`. For example, the following proof goal states a property of the `pred` function:

```
Lemma pred_even : (n:nat)(even n) ->
        (even (pred (pred n))).
```

Once a proof goal is stated, Coq enters an internal loop where the user is prompted to prove the goal interactively by means of pre-existing lemmas. Upon completion of the proof, the lemma is saved in Coq and is reusable for further proof developments.

2.2 The applπ Library

applπ[2] is a Coq library that enables verification of concurrent programs. It provides (1) a modeling language, (2) a specification language (or logic, for short), and (3) a collection of lemmas. Using this library, it is possible to verify concurrent programs using the following approach:

(1) Write a model of the concurrent program using the applπ language.
(2) Write the properties of the concurrent program using the applπ logic.
(3) Prove that the property holds of the model using the lemmas of the library.

In the following, we give an overview of the contents of the applπ library.

The applπ modeling language is a Coq encoding of a minimal concurrent language based on the π-calculus[11] that it extends with datatypes and functions, like the Pict programming language[14]. The applπ modeling language defines two syntactic entities: *channels* and *processes*. Intuitively, processes interact with each other by exchanging values over channels.

Channels are implemented by the functional type `chan`. For some Coq datatype `T`, the type `(chan T)` represents the type of channels that carry data of type `T`. For instance, `(chan nat)` represents the type of channels that carry natural numbers.

The syntax of processes is implemented by the inductive type `proc`. It defines a set of constructors that each represents a basic process:

- `zeroP` represents a terminated process.
- `(inP c Q)`, where `c` is a channel and `Q` is a function, represents a process that waits for some value `v` along the channel `c` and behaves as the process `(Q v)` after reception.

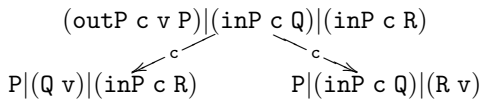


Fig. 1 Example of communications between processes.

- `(rinP c Q)` intuitively represents infinitely many `(inP c Q)` processes in parallel. `rinP` stands for replicated inputs and corresponds to multi-threading.
- `(outP c v Q)` represents the process that emits some value `v` along the channel `c` and then behaves as the process `Q`.
- `(parP P Q)` represents the parallel composition of processes `P` and `Q`.
- `(nuP Q)` represents a process that creates some private channel `c` and then behaves as `(Q c)`.

We define `InAtom` and `OutAtom` as abbreviations for input and output processes whose continuation is `zeroP`.

The semantics of the applπ modeling language is a binary relation between processes that formalizes in particular the notion of communication between processes. In this paper, we informally represent communications using a diagram notation. For example, the possible communications inside the process `(parP (outP c v P) (parP (inP c Q) (inP c R)))` are depicted in **Fig. 1** (the constructor `parP` is written | to save space).

The second part of the applπ library is a logic for specification of applπ programs. This logic consists of:

- a set of state formulas including propositional and spatial formulas[4],
- a satisfaction relation between state formulas and processes noted `sat`,
- a set of temporal formulas similar to temporal logics[6],
- another satisfaction relation between temporal formulas and processes noted `tsat`☆.

The informal semantics of formulas is given in **Fig. 2**.

The last part of the applπ library is a collection of lemmas for verification. We defer overview of these lemmas to Section 6.2, where our case study will provide us with a concrete illustration.

☆ The existence of two satisfaction relations is to prevent temporal formulas from being used inside spatial formulas. This is required to guarantee the soundness of some important lemmas[3].

3. The SMTP Model

In this section, we explain the SMTP model on which the mail server is based. This model is defined in the RFC 821[15].

The mail server consists of several parts, as depicted in **Fig. 3**. The SMTP receiver receives mails from other mail servers and mail clients using the SMTP protocol and stores received mails in a mail queue, implemented by a file system. The SMTP sender extracts mails from the mail queue and sends them to other mail servers or mail clients using the SMTP protocol. In this paper, we are interested in the SMTP receiver part.

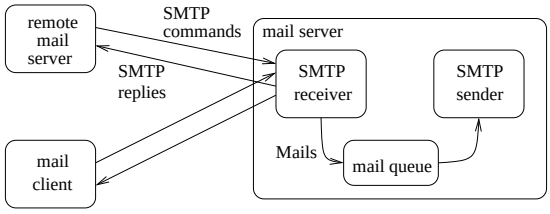


Fig. 3 SMTP model.

The SMTP protocol is depicted in **Fig. 4**. An SMTP protocol session consists of *commands* and some mail contents sent to the mail server, that sends back *replies*. The client starts a session by sending the HELO command; the server replies with its identity and creates an *envelope*. The client sends the MAIL command to set the return path of the envelope with the address of the mail sender. The client then sends one or more RCPT commands to add addresses to the list of recipients of the envelope. The client eventually sends the DATA command, followed by the mail contents and terminated by a "dot". At any moment, it can reset the session using the RSET command, abort using the ABORT command, sends a dummy command NOOP, or closes the session with the QUIT command. For each command, the server answers with an appropriate message, possibly reporting an error.

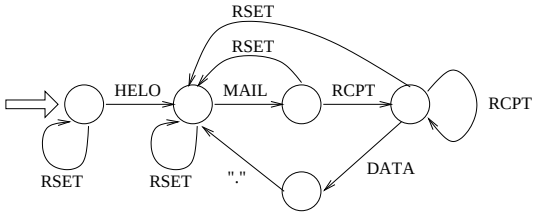


Fig. 4 SMTP commands.

`(sat ISANY P)`		always true
`(sat (NEG f) P)`	iff	`(sat f P)` is false
`(sat (OR f g) P)`	iff	`(sat f P)` or `(sat g P)` is true
`(sat (OUTPUTS c v f) P)`	iff	`P` has a subprocess `(outP c v Q)` such that `(sat f Q)`
`(sat (INPUTS c f) P)`	iff	`P` has a subprocess `(inP c Q)` such that `(sat f (Q v))` for any `v`
`(sat (CONSISTS f g) P)`	iff	`P` can be decomposed into `P1` and `P2` with `(sat f P1)` and `(sat f P2)`
`(tsat (STAT f) P)`	iff	`(sat f P)`
`(tsat (NEGT f) P)`	iff	`(tsat f P)` is false
`(tsat (ORT f g) P)`	iff	`(tsat f P)` or `(tsat g P)` is true
`(tsat (MAYEV f) P)`	iff	for some execution, we eventually have `(tsat f P)`
`(tsat (MUSTEV f) P)`	iff	for any execution, we eventually have `(tsat f P)`

Fig. 2 Informal semantics of applπ logic.

4. Modeling of the Mail Server

In this section, we explain how we model the mail server written in Java using a concurrent program written in the applπ modeling language. Intuitively, modeling consists in translating Java datatypes and control structures into the applπ modeling language. This is facilitated by the fact that the applπ modeling language provides (1) Coq datatypes and control structures which are very close to their Java counterparts, and (2) concurrency primitives to handle directly communications, nondeterminism, and multi-threading (as seen in Section 2.2). In the following, we comment on the main aspects of this translation.

4.1 Datatypes

The mail server defines a number of datatypes that we directly translate into Coq inductive types. For example, the mail server defines constants to implement SMTP commands, like the HELO command:

```
  static final int cmd_helo = 0;
```

Other SMTP commands are similarly implemented by the constants `cmd_mail_from`, `cmd_rcpt_to`, `cmd_data`, `cmd_rset`, `cmd_abort`, `cmd_noop`, `cmd_quit`, and `cmd_unknown` (for unknown commands). We represent these constants in Coq using the following inductive type:

```
Inductive SMTP_cmd : Set :=
  cmd_helo: String -> SMTP_cmd
| cmd_mail_from: String -> SMTP_cmd
| cmd_rcpt_to: String -> SMTP_cmd
| cmd_data: String -> SMTP_cmd
| cmd_noop: SMTP_cmd
| cmd_rset: SMTP_cmd
| cmd_quit: SMTP_cmd
| cmd_abort: SMTP_cmd
| cmd_unknown: SMTP_cmd.
```

Similarly, we represent SMTP replies using an inductive type (we have elided one part of the definition to save space):

```
Inductive ReplyMsg : Set :=
  rep_ok_helo: ReplyMsg
| ...
```

and mails saved by the mail server by a record:

```
Record Mail: Set := mail{
 domain: String;
 rev: Rfc821_path;
 fwd: (list Rfc821_path);
 body: String}.
```

where the fields `domain`, `rev`, and `fwd` correspond to the envelope (as seen in Section 3) and the field `body` corresponds to the mail contents.

4.2 Communications

In the mail server, communications are implemented by means of the `java.io` package. For example, the input stream of SMTP commands is implemented by an instance of a subclass of the class `java.io.InputStream`:

```
  PushbackInputStream from_client;
```

Similarly, the output stream of SMTP replies is implemented by an instance of a subclass of the class `java.io.OutputStream`:

```
  PrintStream to_client;
```

Input and output operations in the mail server are implemented by calls to adequate methods. For example, the method that receives an SMTP command from the input stream is implemented by the `read` method:

```
int get_cmd() throws IOException {
 ...
 int b = from_client.read();
 ...
}
```

Similarly, the method that sends the SMTP reply following the HELO command is implemented by the `println` method:

```
void reply_ok_helo() throws IOException {
 to_client.println
        ("250 " + hostname + " hello");
```

```
}
```

Translation to applπ communication primitives is direct. We represent input and output streams by channels of the appropriate type:

```
Definition InStream := (chan SMTP_cmd).
Definition OutStream := (chan ReplyMsg).
```

and input and output operations by input and output processes:

```
Definition get_cmd [c:InStream; cont:
        SMTP_cmd -> proc] := (inP c cont).
Definition reply [r:ReplyMsg; c:OutStream;
        cont: proc] := (outP c r cont).
Definition reply_ok_helo :=
        (reply rep_ok_helo).
```

We have seen above how to model communications between the mail server and a client. Similarly, communications between the mail server and the file system are modeled by a channel of type `(chan Mail)` and corresponding input and output processes.

4.3 Server State

The mail server features a number of variables (fields of Java objects) that capture its state. To represent these variables, we introduce a representation of the state of the mail server using a record that is intended to be passed around following the flow of execution:

```
Record STATE : Set := smtp_state{
 to_client: OutStream;
 in_stream: InStream;
 queue_dir: File;
 buf: Buffer;
 to_fs : (chan Mail);
 server_name: String;
 from_domain: String;
 rev_path: Rfc821_path;
 fwd_paths: Rfc821_pathlist;}.
```

The fields `in_stream` and `to_client` contain the channels used for communication with the client. The fields `queue_dir` and `buf` represent respectively the directory and the file that implement the mail queue. The field `to_fs` contains the channel for communication with the file system. The field `server_name` contains an identifier that the mail server uses for SMTP replies. The fields `from_domain`, `rev_path`, and `fwd_paths` correspond to the envelope being built.

4.4 Control Structures

The mail server appeals to a wide variety of control structures. Basic control structures such as conditionals are easily translated into case analyses. However, non-terminating loops cannot be represented directly in Coq (they are rejected because they make proof checking undecidable). In the following, we explain how we translate the main loop of the mail server into the applπ modeling language. The idea is to implement it using communications with respect to replicated inputs.

The main loop of the mail server (**Fig. 5**, on the left) waits for incoming requests and, upon reception of an HELO command, it enters a loop in which subsequent commands are processed. This processing of subsequent SMTP commands is ensured by methods `get_helo`, `get_mail_from`, `get_rcpt_to`, and `get_data`.

We represent the main loop of the mail server by means of replicated inputs that exchange the state of the mail server through a set of private channels: `heloc`, `mailc`, and `rcptc` (Fig. 5, on the right). These replicated inputs are written in such a way that they emulate the flow of control of the original program. It should be observed that, even though we use replicated inputs, the resulting process represent a single thread of computation. (Multiple threads of computation will appear when composing the mail server with its environment, see next section.)

Each method called from the main loop of the mail server is modeled in a systematic way as a process. We illustrate this modeling using the example of the `get_helo` Java method.

The Java method `get_helo` (**Fig. 6**, on the left) tries to fetch incoming HELO commands, replies appropriately, and redirects the flow of control to other methods. It is essentially a switch statement.

We represent the Java method `get_helo` by means of the process `get_helo_def` (Fig. 6, on the right). The switch statement is translated into a case analysis. The control flow is emulated by means of communications: break statements are replaced by communications along the `heloc` channel, return statements are replaced by communications along the `mailc` channel, etc. Each method call in the original program is translated to a call to the corresponding function resulting from the translation. Finally, successful termination is modeled by the presence of a dummy value along a special channel `resultc`:

```
Variable resultc : (chan unit).
Definition succ := (OutAtom resultc tt).
```

(The statement `Variable` declares a global variable in Coq. `tt` is the only element of type `unit`.)

Original Java Program:

```
void work() throws IOException,
        Smtpdialog_bug,
        Mailqueue_fatal_exception {
    ...
    get_helo();
    int msg_no = 0;
    while (!quit) {
        do_rset();
        if (!get_mail_from()) continue;
        if (!get_rcpt_to()) continue;
        get_data(msg_no++);
    }
    ...
}
```

applπ Model:

```
Definition work
 [i:InStream; o: OutStream] : (chan Mail)->proc :=
[tofs:(chan Mail)]
let st = initial_state in
(nuP [heloc:(chan STATE)]
(nuP [mailc:(chan STATE)]
(nuP [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)))))).
```

Fig. 5 Mail server main loop.

Original Java Program:

```
void get_helo() throws IOException {
 while (true) {
  ...
  int cmd = get_cmd();
  String arg = get_arg();
  switch(cmd) {
  case cmd_unknown:
      reply_unknown_cmd(); break;
  case cmd_abort:
      reply_ok_quit(); quit = true;
      return;
  case cmd_quit:
      reply_ok_quit(); quit = true;
      return;
  case cmd_rset:
      do_rset();
      reply_ok_rset(); break;
  case cmd_noop:
      reply_ok_noop(); break;
  case cmd_helo:
      if (do_helo(arg)) return;
      else break;
  case cmd_rcpt_to:
      reply_no_mail_from(); break;
  default:
      reply_no_helo(); break;
  }
 }
}
```

applπ Model:

```
Definition get_helo_def
 [heloc: (chan STATE); mailc: (chan STATE)] :
 STATE -> proc :=
 [st:STATE](get_cmd (in_stream st)
 [c:SMTP_cmd](get_arg (in_stream st)
 [_:unit]
 Cases c of
  cmd_unknown =>
   (reply_unknown_cmd (to_client st) (OutAtom heloc st))
   | cmd_abort =>
   (reply_ok_quit (to_client st) succ)
   | cmd_quit =>
   (reply_ok_quit (to_client st) succ)
   | cmd_rset =>
   (reply_ok_rset (to_client st) (OutAtom heloc st))
   | cmd_noop =>
   (reply_ok_noop (to_client st) (OutAtom heloc st))
   | (cmd_helo arg) =>
   (do_helo arg st
     [x:bool;st:STATE]
      if x then (OutAtom mailc st) else (OutAtom heloc st))
   | (cmd_rcpt_to b) =>
   (reply_no_mail_from (to_client st) (OutAtom heloc st))
   | _ =>
   (reply_no_helo (to_client st) (OutAtom heloc st))
 end)).
```

Fig. 6 Function to handle HELO commands.

5. Modeling of the Environment

In order to verify that the mail server correctly handles client requests, we need to make some hypotheses on its environment. In particular, we assume that we perform the verification in presence of a correct client and a correct file system. The network communications and the host computer are also part of the environment of the mail server. In general, we cannot

```
Inductive val [s:InStream] : proc->Prop :=
  say_helo: (P:proc)(c: SMTP_cmd)
  (valid_cmd_helo c) ->
  (val_after_helo s P) ->
  (val s (outP s c P))
| say_quit: (val s (OutAtom s cmd_quit))
| say_abort: (val s (OutAtom s cmd_abort))
| say_skip: (P:proc)(c: SMTP_cmd)
  ~(valid_cmd_helo c) ->
  ~c=cmd_quit -> ~c=cmd_abort ->
  (val s P) ->
  (val s (outP s c P))
| say_io_error: (P:proc)
  (val s P) ->
  (val s (outP IOexnc tt P))
with val_after_helo [s:InStream] :
                        proc->Prop :=
...
```

Fig. 7 Specification of the input stream.

expect network communications and the host computer to be reliable.

In this section, we show how to model a correct mail client and a correct file system and the hypotheses of unreliable network communications and of an unreliable host computer.

5.1 Network Errors

We model network errors by the presence of a dummy value along a special channel `IOexnc`:

```
Variable IOexnc : (chan unit).
```

Network errors may occur during communications between the mail server and the mail client, and between the mail server and the file system. We therefore use the special channel `IOexnc` in the specification of the client and of the file system, as explained below.

5.2 Client Specification

A client is correct if it emits valid streams of SMTP commands, as specified by the RFC 821[15]. This requirement amounts to specifying the set of valid streams of SMTP commands. We render this specification by means of the predicate `val` in **Fig. 7**.

Informally, the predicate `val` can be read as follows. The client emits valid streams of SMTP commands if:

- after emitting a valid HELO command it still emits valid streams of SMTP commands, as defined by the predicate `val_after_helo` (constructor `say_helo`),
- it emits a QUIT or ABORT command (constructors `say_quit` and `say_abort`),
- it emits any other command such that the rest of the emission is still valid (constructor `say_skip`).

The constructor `say_io_error` does not correspond to the definition of validity of streams of SMTP commands, we add it to take into account the possibility for a network error.

Similarly, we take into account the possibility for a network error prior to emission of SMTP replies. This is rendered by the predicate `ack` in **Fig. 8**.

```
Inductive ack : OutStream -> proc -> Prop :=
  ack_rep : (y:OutStream)(P:ReplyMsg -> proc)
 ((x:ReplyMsg)(ack y (P x))) -> (ack (inP y P))
| ack_rep_io_error : (y:OutStream)(P:proc)
 (ack y P) -> (ack y (outP IOexnc tt P)).
```

Fig. 8 Specification of the output stream.

The constructor `ack_rep` corresponds to the usual situation where the SMTP reply is sent to the client, and the constructor `ack_rep_io_error` corresponds to the exceptional situation where a network error prevents the emission of the SMTP reply.

We pack above `val` and `ack` predicates into a single `valid_client` predicate that specifies correct clients. Similarly, we define a predicate `valid_fs` that specifies correct file systems.

5.3 System Failures

An unreliable host computer is modeled by the non-deterministic possibility for a a failure. A failure is modeled by the presence of a dummy value along a special channel `failc`:

```
Variable failc : (chan unit).
```

A failure may occur at any moment. We model this non-determinism by a process that non-deterministically output a dummy value along the above special channel:

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

(The question mark `?` in Coq automatically infers the corresponding type.) The possible reductions of this non-deterministic failure generator are depicted in **Fig. 9**.

6. Formal Verification

6.1 Goal Statement

We are interested in verifying that the mail server modeled in Section 4 executed under the environment modeled in Section 5 correctly handles incoming streams of SMTP commands. In other words, we want to verify that, for any possible execution, the process formed by the

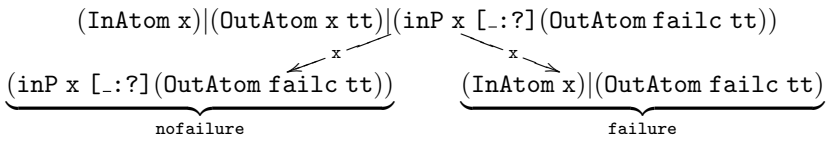


Fig. 9 Reductions of the failure generator.

```
Lemma valid_protocol:
 (client:InStream->OutStream->proc)
 ((i:?)(o:?)(valid_client (client i o))) ->
 (fs:(chan Mail)->proc)
 ((tofs:?)(valid_fs (fs tofs))) ->
 (is_set resultc&(IOexnc&(failc&nilC))) ->
 let P = (resultc&(IOexnc&(failc&nilC)))#
   (nuP [i:InStream]
     (nuP [o:OutStream]
       (nuP [tofs:(chan Mail)]
         (parP (client i o)
           (parP (work i o tofs)
             (parP (fs tofs)
               (may_fail))))))) in
 (tsat reports_succ_or_error P).
```

Fig. 10 Goal statement.

parallel composition of a correct client, a correct file system, (the model of) the mail server, and a non-deterministic failure generator results in successful termination, a network error, or a system failure. Observe that the resulting process has multiple threads of computation.

The property informally stated above can be formally written using the applπ logic as follows:

```
Definition reports_succ_or_error :=
 (MUSTEV (STAT
   (OR (OUTPUTS resultc tt ISANY)
     (OR (OUTPUTS IOexnc tt ISANY)
       (OUTPUTS failc tt ISANY))))).
```

(Special channels `resultc`, `IOexnc`, and `failc` are used to observe respectively successful termination, network error, and failures.)

Let us write `P` for the process formed by the parallel composition of a correct client, a correct file system, the mail server, and a non-deterministic failure generator. The proof goal can be stated as follows:

```
Lemma valid_protocol: ...
 (tsat reports_succ_or_error P).
```

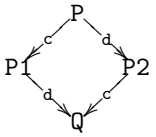
The complete statement is given in **Fig. 10**. (Symbols `nilC` and `&` respectively represent the empty list and cons cells.)

6.2 Formal Proof

We have formally proved using Coq the goal stated above. The proof is by induction on the predicate `val` (excerpt in Fig. 7). It requires 3927 commands using the applπ library (for a 400 lines model and 200 lines specification).

An important aspect of the formal proof is how we deal with interleaving sequences of communications. Since the mail server runs in parallel with the failure generator, there are several possible execution paths that only differ by the moment when the failure generator is scheduled for execution. The proliferation of such different possible execution paths is harmful because they considerably augment the number of subgoals of the formal proof. This situation is an instance of the state-space explosion problem.

The applπ library provides lemmas that enable *partial order reduction* to deal with the state-space explosion problem. We illustrate the basic idea of partial order reduction with the following example. Let us consider some process `P` in which two communications along channels `c` and `d` are enabled, such that both communications can be executed in whatever order to reach the same process `Q`:



To verify a formula of the form `(MUSTEV f)` in this situation, it is often sufficient to explore only one execution path. More generally, partial order reduction reduces the number of execution paths to be explored for the purpose of verification to a subset representative of all the possible orderings of communications. The applπ library provides lemmas that enable partial order reduction for the applπ language and its logic[2),3)].

Lemmas from the applπ library that enable partial order reduction are particularly useful to verify the mail server. Let us consider the fragment of the state space of the whole system depicted in **Fig. 11**. The initial process is shortened to `client|work|fs|may_fail` to save space. From the initial process, it is possible to perform either (1) a communication along channel `i` through which the client sends a first

SMTP command to the mail server, leading to the process client′|work′|fs|may_fail, or (2) one of the two communications along channel x enabled by the non-deterministic failure generator. Lemmas from the applπ library tell us that we can safely ignore the execution paths starting with the failure generator (dotted lines in Fig. 11), and resume verification from the process client′|work′|fs|may_fail.

6.3 Discussion

The size of the formal proof is large but can be substantially reduced. In fact, prior to the case study presented in this paper, we had already verified the same mail server using a different approach[1]. The basic idea of this approach was to model the mail server using only functional constructs provided by the Coq language. The formal proof that the mail server correctly handles client requests was almost four times smaller (1059 commands). However, it appears that both approaches lead to essentially the same proof tree thanks to partial order reduction. Therefore, the overhead induced by the applπ library is not a fundamental issue and can be alleviated by improving automation.

Despite this overhead, the verification of concurrent programs using the applπ library is still a satisfactory approach because it handles multi-threading explicitly and because it is possible to extract a runnable concurrent program from the model. The latter was not possible with our previous model[1] because it was polluted with functional constructs whose purpose was only the modeling of non-determinism. In contrast, modeling of non-deterministic failures in the applπ model does not interfere with the modeling of the original program. Consequently, it is possible to execute the code of the model as it is.

To execute the code of the model, it is possible to write a virtual machine to interpret the applπ modeling language. However, the extraction facility of Coq provides a more effective solution. This extraction facility turns Coq programs into ML programs (OCaml, Haskell, etc.) by associating to each Coq inductive type a corresponding ML datatype, to each Coq function a corresponding ML function, etc. In particular, the concurrency primitives of the applπ modeling language are extracted in the form of the constructors of the following ML datatype (we use OCaml syntax for concreteness):

```
Coq < Extraction proc.
type proc =
  | ZeroP
  | InP of Obj.t * (Obj.t -> proc)
  | RinP of Obj.t * (Obj.t -> proc)
  | OutP of Obj.t * Obj.t * proc
  | ParP of proc * proc
  | NuP of (Obj.t -> proc)
```

where the type Obj.t corresponds to channels (this is because we have only provided Coq with the type of channels, not their implementation). The idea to execute the extracted OCaml code is to pre-process it to replace each call to one of the constructor above by a call to an OCaml function that implements the appropriate semantics. In other words, given some (parameterized) type channel and a set of functions zeroP, inP, etc. that implement the semantics of the applπ modeling language, we have a complete mechanism to run models.

7. Conclusion

In this paper, we have explained how one can verify an existing concurrent program using a Coq library. More precisely, we gave an overview of the verification of an existing mail server written in Java using the applπ library. First, we introduced the Coq proof assistant and the applπ library. Second, we explained how to model the mail server into an applπ program. Third, we explained how to model the environment of the mail server, including modeling of system errors. Last, we reported on the formal proof that the mail server correctly handles client requests. We compared the results of verification with an alternative approach and observed that (1) the overhead induces by the applπ library can potentially be eliminated through better automation, and that (2) the applπ model is more satisfactory because in particular it can be run as it is. This case study shows that the applπ library provides us with a complete solution to write, verify, and run concurrent programs in the Coq interface.

8. Related Work

The issue of verification of concurrent programs in proof assistants has been addressed through formalization of the UNITY formalism[5),8),13)]. This work also includes various verifications of non-trivial concurrent programs. The originality of our case study is that we verify an existing implementation and discuss for that purpose several reusable techniques for modeling.

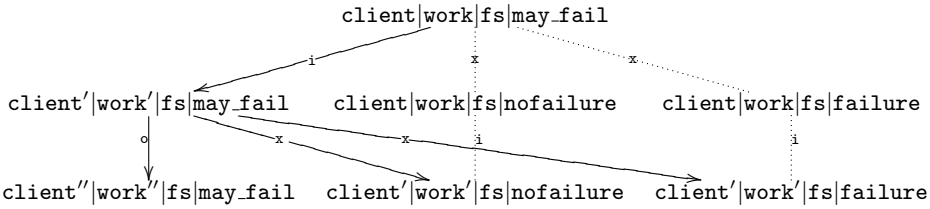


Fig. 11 Effect of partial order reduction on verification of the mail server.

There exist several encodings of the π-calculus in proof assistants with accompanying libraries[9),10),16)]. This work essentially focuses on verification of meta-properties of the pure π-calculus. In comparison, the applπ library is built above an applied version of the π-calculus and we focus on verification of properties of programs.

In this paper, we used a proof assistant to verify a concurrent program. Model checking is an alternative approach[7)] that could have equally-well applied to verification of the mail server. The advantage of proof assistants is that they can handle directly infinite state-spaces thanks to induction, contrary to model checkers that are limited to finite state-spaces (unless one resorts to appropriate abstraction techniques). This is the reason why we investigate the usage of proof assistants to verify concurrent programs.

9. Future Work

We plan to tackle the issue of reducing the size of formal proofs by improving automation in the applπ library and combining interactive proofs with model checking for the applπ modeling language and its logic.

Acknowledgments This work is partially supported by a research project funded by Japanese Ministry of Education and Science's research program "e-Society."

References

1) Affeldt, R. and Kobayashi, N.: Formalization and verification of a mail server in Coq, Okada, M., Pierce, B., Scedrov, A., Tokuda, H. and Yonezawa A. (eds), *International Symposium on Software Security*, Tokyo, Japan, November 8–10, 2002, Vol.2609 of *Lecture Notes in Computer Science*, pp.217–233, Springer (Feb. 2003). Coq scripts available at: `http://web.yl.is.s.u-tokyo.ac.jp/~affeldt/mail-system.tar.gz`.

2) Affeldt, R. and Kobayashi, N.: A Coq library for verification of concurrent programs, *4th International Workshop on Logical Frameworks and Meta-Languages (LFM 2004)*, Cork, Ireland, July 5, 2004, pp.66–83. Preliminary proceedings available at: `http://cs-www.cs.yale.edu/homes/carsten/lfm04/`. Formal proceedings are to appear in *Electronic Notes in Theoretical Computer Science*, Elsevier. Coq documentation available at: `http://web.yl.is.s.u-tokyo.ac.jp/~affeldt/applpi/`.

3) Affeldt, R. and Kobayashi, N.: Partial order reduction for verification of spatial properties of pi-calculus processes, *11th International Workshop on Expressiveness in Concurrency (EXPRESS 2004)*, London, UK, August 30, 2004, pp.113–127. Preliminary proceedings. Formal proceedings are to appear in *Electronic Notes in Theoretical Computer Science*, Elsevier.

4) Cardelli, L. and Gordon, A.D.: Anytime, anywhere: modal logics for mobile ambients, *27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2000)*, Boston, Massachusetts, USA, January 19–21, 2000, pp.365–377, ACM Press (2000).

5) Chetali, B.: Formal verification of concurrent programs using the Larch Prover. *IEEE Transactions on Software Engineering*, Vol.24, No.1, pp.46–62 (1998).

6) Clarke, E.M., Grumberg, O. and Peled, D.A.: *Model Checking*, MIT Press (2000).

7) Clarke, E.M. and Wing, J.M.: Formal methods: state of the art and future directions, *ACM Computing Surveys*, Vol.28, No.4, pp.626–643 (1996).

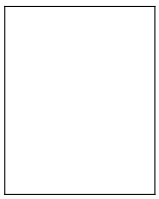
8) Heyd, B. and Crégut, P.: A modular coding of UNITY in Coq, von Wright, J., Grundy, J. and Harrison, J. (eds), *Theorem Proving in Higher Order Logics*, Vol.1125 of *Lecture Notes in Computer Science*, pp.251–266, Springer (Aug. 1996).

9) Hirschkoff, D.: *Mise en œuvre de preuves de bisimulation*, Ph.D. thesis, École Nationale des Ponts et Chaussées (1999).

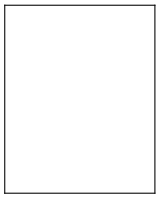
10) Honsell, F., Miculan, M. and Scagnetto, I.: π-calculus in (co)inductive type theory, *Theoretical Computer Science*, Vol.253, No.2, pp.239–285 (2001).

11) Milner, R.: *Communicating and Mobile Systems: the π-calculus*, Cambridge University Press (1999).
12) Paulin-Mohring, C.: Inductive definitions in the system Coq: Rules and properties, Bezem, M. and Groote, J.F. (eds), *1st International Conference on Typed Lambda Calculi and Applications (TLCA 1993)*, Utrecht, The Netherlands, March 16–18, 1993, Vol.664 of *Lecture Notes in Computer Science*, pp.328–345, Springer (1993).
13) Paulson, L.C.: Mechanizing a theory of program composition for UNITY, *ACM Transactions on Programming Languages and Systems*, Vol.23, No.5, pp.626–656 (2001).
14) Pierce, B.C. and Turner, D.N.: Pict: A programming language based on the pi-calculus, Plotkin, G., Stirling, C. and Tofte, M. (eds), *Proof, Language and Interaction: Essays in Honour of Robin Milner*, MIT Press (2000).
15) Postel, J.B.: Rfc 821: Simple mail transfer protocol, available at: `http://www.faqs.org/rfcs/rfc821.html` (Aug. 1982).
16) Röckl, C., Hirschkoff, D. and Berghofer, S.: Higher-order abstract syntax with induction in Isabelle/HOL: Formalizing the pi-calculus and mechanizing the theory of contexts, *4th International Conference on Foundations of Software Science and Computation Structures (FOSSACS 2001)*, Genova, Italy, April 2–6, 2001, Vol.2030 of *Lecture Notes in Computer Science*, Springer (2001).
17) Shibayama, E., Hagihara, S., Kobayashi, N., Nishizaki, S., Taura, K. and Watanabe, T.: AnZenMail: A secure and certified e-mail system, Okada, M., Pierce, B., Scedrov, A., Tokuda, H., and Yonezawa, A. (eds), *International Symposium on Software Security*, Keio University, Tokyo, Japan, November 8–10, 2002, Vol.2609 of *Lecture Notes in Computer Science*, Springer (Feb. 2003).

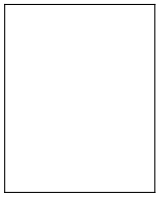
(Received July 2, 2004)
(Accepted September 21, 2004)



Reynald Affeldt was born in 1976, graduated in 2000 from the École Nationale Supérieure des Mines de Nancy (France), and received his M.S. and D.S. degrees in Computer Science from the University of Tokyo in 2001 and 2004 respectively. He is currently a researcher in the Graduate School of Information Science and Technology at the University of Tokyo. His main interests are partial evaluation, proof assistants, and process calculi. He is a member of the Japanese Society of Software Science and Technology.



Naoki Kobayashi was born in 1968, and received his B.S., M.S., and D.S. degrees in Computer Science from the University of Tokyo in 1991, 1993, and 1996, respectively. He was a research associate and a lecturer in the Graduate School of Science at the University of Tokyo from 1993 to 2001. In 2001, he became an associate professor in the Graduate School of Science at the Tokyo Institute of Technology. In October 2004, he became professor in the Graduate School of Information Sciences at Tohoku University. His current major research interests are in the area of principles of programming languages. In particular, he is interested in type systems and static analysis of functional and concurrent programming languages. He is a member of the Association for Computing Machinery, the Information Processing Society of Japan, and the Japanese Society of Software Science and Technology.



Akinori Yonezawa received his Ph.D. degree in Computer Science from the MIT in 1977. He is currently professor in the Department of Computer Science at the University of Tokyo. His current major research interests are in the areas of concurrent/parallel computation models, programming languages, distributed computing, and software security. He was a member of the Scientific Advisory Board of the German National Research Institute of Computer Science, served as an associate editor of the ACM Transactions of Programming Languages and Systems (TOPLAS), and was a member of the editorial boards of IEEE Computer and IEEE Concurrency. He also acted as the president of the Japanese Society of Software Science and Technology. In 2000, he was appointed by the Prime Minister to be a member of the Reformation and Deregulation Committee and the chairman of its Education Subcommittee for three years. He is a fellow of the ACM as well as the Japanese Society of Software Science and Technology.