Formalization and Verification of a Mail Server in Coq

Reynald Affeldt[1] and Naoki Kobayashi[2]

[1] Department of Computer Science, University of Tokyo
affeldt@yl.is.s.u-tokyo.ac.jp
[2] Department of Computer Science, Tokyo Institute of Technology
kobayasi@kb.cs.titech.ac.jp

Abstract. This paper reports on the formalization and verification of a mail server (SMTP server) in Coq. The correctness of a mail server is very important: bugs of the mail server may be abused for eavesdropping mail contents, spreading virus, sending spam messages, etc. We have verified a part of a mail server written in Java, by manually translating the Java program into a Coq function as faithfully as possible, and verifying properties of the Coq function. The results of this experiment indicate the feasibility and usefulness of verification of middle-sized system softwares in this style. The verification has been carried out in a few months, and a few bugs in the mail server have been indeed found during the verification process.

1 Introduction

The *AnZenMail* project [13] consists in implementing a secure facility for electronic mail. This is a large system with many subparts and among them the mail server is a central one. In this paper, we report an experiment of formalization and verification of this mail server.

From the security point of view, it is very important to verify that a mail server is correctly implemented because bugs may be responsible for loss of information and flaws may also be abused for eavesdropping mail contents, spreading viruses, sending spam messages, etc.

Concretely, we check by means of a theorem prover (namely Coq [14]) that the implementation of the mail server is correct with respect to Internet standards and reliable. Our approach is to create a model as faithful as possible of the actual mail server by translating the code base written in the Java language into a Coq function and verify the properties of the Coq function.

There are other approaches using a theorem prover such as extraction of an implementation directly from the proofs (in an adequate theorem prover) or direct proofs of the actual code base (without any need for a model). In comparison, our approach is convenient in many respects. First, it does not depend on the implementation language. It is thus possible to use an implementation language with appealing security and efficiency properties such as Java. Second, the proof development can be managed in size and time like the source code development; proofs done in parallel with the implementation increase confidence in the code base and it is possible to go back and forth between the specifications and the implementation for adjustments.

We claim that our experiment makes the following contributions:

– it is a large application of theorem proving (large in terms of the size of the original code and proofs),
– it illustrates several techniques to model in a functional interface a reactive system with infinitely many states, and
– it deals exhaustively with system errors.

The rest of this paper is organized as follows. Sect. 2 explains what part of the mail system we formalize. Sect. 3 explains the formalization of the program and its specifications. Sect. 4 presents the results of our verification. Sect. 5 discusses lessons learned from our experiment. Sect. 6 reviews related work. Sect. 7 concludes and lists future work.

2 Mail Server

2.1 A Secure Mail System

The *AnZenMail* project is a subproject of the *secure computing project*, which aims at enhancing the security of computer systems by using three levels of safety assurance mechanisms —formal verification/analysis techniques, compile-time code insertion for dynamic-checking of safety properties, and OS-level protection (see the project home page `http://anzen.is.titech.ac.jp/index-e.html` for details). The *AnZenMail* system consists of mail servers, which deliver mails, and mail clients, which interact with users and send/receive mails.

The key features of the *AnZenMail* system are as follows:

1. Use of a high-level programming language: both the server code and the client code are written in a safe language (namely Java) to minimize security holes (that are often caused by buffer overflows).
2. Protection against forging or modification of mails: secure protocols based on digital signatures are used to protect the contents of mails and the identity of senders. Correctness of the security protocols are proved in a formal manner.
3. Fault-conscious design of the server code: the server code is carefully designed in such a way that received mails are not lost even if the server crashes.
4. Verification of the server code: the core part of the server code is verified in a formal manner.
5. Safe execution of mail attachments: on the mail client side, safe execution of code received as a mail attachment is enforced by static analysis of the code, compile-time code insertion for dynamically checking security properties, and OS-level protection mechanisms.

The present paper is concerned with the fourth feature above. Verification of the server code is also related to the second and third features: the second and third features guarantee certain properties only at algorithm or design level, and give no guarantee that the properties are indeed satisfied by the actual code. By verifying the code, we can check whether the implementation is correct with respect to the algorithm and the design.

2.2 SMTP Model

The overall structure of the mail server is shown in Fig. 1. It consists of two parts: the SMTP receiver, which receives mails from other mail servers and mail clients using the SMTP protocol and stores received mails in a mail queue, and the SMTP sender, which extracts mails from the mail queue and sends them to other mail servers and mail clients using the SMTP protocol.

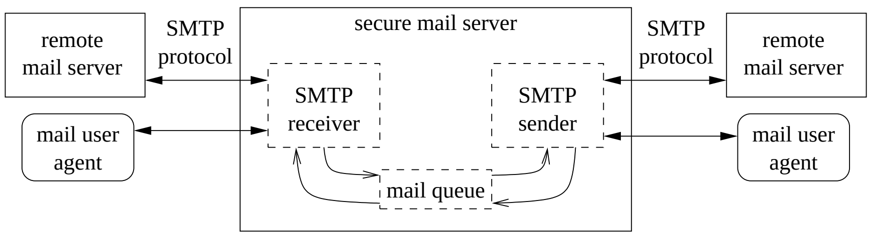

Fig. 1. The structure of the mail server

Fig. 2 illustrates most typical SMTP commands with an automaton. An SMTP protocol session consists of *commands* and some mail contents sent by a mail sender to the SMTP receiver that sends back *replies*. Before a session begins, the SMTP receiver waits for incoming connections. The mail client connects using the HELO command; the server replies with its identity and creates an empty *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 server may reply with error messages to the MAIL and RCPT commands for various reasons (malformed addresses, MAIL and RCPT commands not ordered, etc.) The client eventually sends the DATA command, followed by the mail contents and terminated by a line containing only a period. If the server accepts the data, it delivers the mail contents to recipients and notifies the mail sender if delivery fails. Finally, the client closes the connection with the QUIT command to which the server replies with some greetings. There are a few other commands: the RSET command clears the envelope, the NOOP command just causes the server to reply (it has no effect on the state of the server), etc.

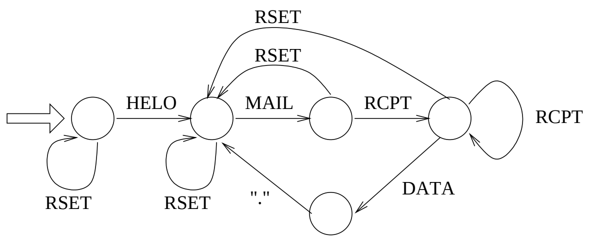

Fig. 2. Abstract state transition

2.3 Target of Formalization and Verification

Our goal is to verify that the implementation of the mail server as a whole satisfies the critical responsibilities highlighted above. Those properties are mainly adequacy of the code with the SMTP protocol and reliability of the implementation, i.e. the fact that accepted mails are eventually delivered.

In this paper, we are concerned with the SMTP receiver part of the mail server. Formalization and verification are therefore restricted to that part of above properties which are relevant to the SMTP receiver. Adequacy with the SMTP protocol amounts to the following properties: the SMTP receiver must handle all correct sessions, reject erroneous commands and send back correct replies. Reliability amounts to the following property: the SMTP receiver must save accepted mail contents in the mail queue for later processing by the SMTP sender.

In the next section, we explain how we formalize the SMTP receiver (hereafter referred to as the mail server) and formulate above properties.

3 Formalization and Verification

The task of formalization can be divided into construction of a model for the execution of the program and specification of the properties to be verified. In the following, we assume some familiarity with the syntax of Java. Other code samples are explained in such a way that no a priori knowledge of Coq is required.

3.1 Model

A model for the execution of the mail server is a program written in the Coq language such that its verification substitutes for the verification of the actual mail server written in Java.

Construction of such a model essentially amounts to conversion of the original program from Java to the Coq language. There are two difficulties. First, Java is an imperative language whereas the Coq language is functional. Second, relevant hardware aspects of the execution environment must be made explicit. For instance, errors due to the network or the host computer may in practice cause the mail server to behave unexpectedly. Therefore, our model must take into account such *system errors* in order for the properties we verify to hold under normal conditions of use. In the following, we first give an overview of code conversion and then explain how we overcome those difficulties.

Code Conversion Overview We manually convert the original program in such a way that its structure is preserved, i.e. any language construct can easily be associated with its counterpart in the model. Conceptually, this task can be divided into conversion of datatypes and conversion of control structures.

Java datatypes are converted to corresponding Coq types. Java data structures (or object fields) are converted to Coq records or tuples, union or enumeration types are converted to Coq *inductive types* (similar to datatypes in ML-like languages), etc. Those objects belong to the type *Set* of program specifications in Coq. For instance, we represent the global state of the mail server by specifying it as a record:

```
Record STATE: Set := smtp_state{to_client: OutputStream;
                                server_name: String;
                                queue_dir: File;
                                buf: Buffer;
                                from_domain: String;
                                rev_path: Rfc821_path;
                                fwd_paths: Rfc821_pathlist;
                                quit: bool;
                                files: FileSystem;
                                oracles: Oracles}.
```

The field *to_client* contains replies from the server, the field *rev_path* contains the return path of the envelope, the field *fwd_paths* contains the list of recipients, etc. (Other relevant fields are explained later.) For Coq records to be manipulated as Java data structures, we also provide mutator functions. For instance, the function *enque_reply*, given a *STATE* and an SMTP reply, updates the list of replies sent by the server so far.

Java control structures are converted to corresponding Coq control structures. Java `switch` statements are converted to Coq *Cases* statements, method calls are converted to function calls, Java sequences `a;b` are converted to Coq function calls *(seq a b)* (where the function *seq* is to be defined later), etc. Conversion of control structures is illustrated later in this section.

Monads for Exception Handling The first difficulty is the conversion of imperative operations in Java to the Coq language. In the case of exception handling, we use monadic style programming [15].

The actual mail server can circumvent some system errors (e.g., network failures) by using Java exceptions. In Coq, we represent all system errors, including *fatal errors* (e.g., host computer failure), by an inductive type:

```
Inductive Exception: Set :=
          IOException: Exception
        | parse_error_exception: Exception
        | Smail_implementation_exception: Exception
        | empty_stream_exception: Exception
        | system_failure: Exception.
```

The *IOException* constructor represents network failures, the *system_failure* constructor represents host computer failures, etc.

Using the above definitions for the state of the mail server and for exceptions, we can mimic exception handling by the following monad-like inductive type (bracketed variables are here parameters and the arrow-like notation indicates constructors arguments):

```
Inductive Except [A: Set]: Set :=
          Succ: A → STATE → (Except A)
        | Fail: Exception → STATE → (Except A).
Definition Result : Set := (Except unit).
```

Result represents the result of a computation which can be either successful (constructor *Succ*) or a failure (constructor *Fail*).

Test Oracles for Non-deterministic Failures The second difficulty is the simulation by the model of non-deterministic system errors. For that purpose, we use in Coq a *coinductive type* that represents test oracles in the form of an infinite sequence of booleans:

```
CoInductive Set Oracles := flip : bool → Oracles → Oracles.
```

Test oracles appear as a part of the state of the model (field *oracles* in the record *STATE* above) and are used at any point where a system error may occur. Concretely, the value of the leading test oracle determines whether a system error occurs or not, and whenever a test oracle is used, test oracles are updated (by applying the function *update_coin* to the current *STATE*). For instance, a network failure may occur whenever the mail server sends a reply, hence the following definition of the function that sends replies:

```
Definition reply [r:ReplyMsg; st:STATE]: Result :=
          (Cases (oracles st) of
                    (flip true coin) ⇒
                              (Succ unit tt (update_coin (enque_reply st r) coin))
               | (flip false coin) ⇒
                              (Fail unit IOException (update_coin st coin))
          end).
```

Similarly, a host computer failure may occur during the execution of any two successive Java statements. In Coq, we represent the sequence of Java statements by the following function (bracketed variables are here function arguments):

```
Definition seq: Result → (STATE→Result) → Result :=
          [x: Result][f: STATE→Result]
```

```
(Cases x of
      (Succ _ st) ⇒
          Cases (oracles st) of
                 (flip true coin) ⇒ (f (update_coin st coin))
              | (flip false coin) ⇒ (Fail unit system_failure st)
          end
    | (Fail e st) ⇒ (Fail unit e st)
  end).
```

Example of Code Conversion Equipped with above objects, we show how we convert a method taken from the original Java program to its Coq counterpart.

The `get_helo` method (Fig. 3) is essentially a loop that fetches SMTP commands until it receives the HELO command. In case of unexpected commands, it replies with some error message; in case of termination commands, it terminates the SMTP protocol session. Note that the execution of `get_helo` may also be interrupted by network failures if the socket gets closed or broken.

```
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;
        }
    }
}
```

Fig. 3. Java `get_helo` method

The context in which the `get_helo` method is called is the mail server main loop (Fig. 4). Intuitively, upon reception of the HELO command, a loop is entered in which the following MAIL, RCPT and DATA commands are to be processed.

```
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++);
    }
    ...
}
```

Fig. 4. Mail system main loop

The first step of code conversion is to convert Java datatypes to Coq types. For instance, SMTP commands represented by the integers `cmd_helo`, `cmd_quit`, etc. in the actual Java code base are converted to the inductive type *SMTP_cmd* (Fig. 5).

```
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.
```

Fig. 5. *SMTP_cmd* inductive type

The second step of code conversion is to convert Java control structures to Coq control structures. Let us directly comment on the result of code conversion (Fig. 6). In the resulting *get_helo* Coq function, we see that Java's `switch` is replaced by Coq's *Cases*, that sequences of statements are replaced by calls to functions *seq*, *comp* and *seq_b* (*comp* and *seq_b* are slight variations of *seq*), and that calls to third-party Java methods are replaced by calls to their respective Coq counterparts (after adequate code conversion). Observe that, as a result of our code conversion, statements that used to appear syntactically after the call to method `get_helo` in method `work` now appear inside the function *get_helo*.

```
Fixpoint get_helo [in_stream: InputStream]: STATE → Result :=
    (comp get_cmd
    (comp get_arg
        [st: STATE]
        Cases in_stream of
          nil ⇒ (fail empty_stream_exception st)
        | (cons m in_stream') ⇒
            (Cases m of
                cmd_unknown ⇒
                    (seq (reply_unknown_cmd st) (get_helo in_stream'))
              | cmd_abort ⇒
                    (seq (reply_ok_quit st) succ)
              | cmd_quit ⇒
                    (seq (reply_ok_quit st) succ)
              | cmd_rset ⇒
                    (seq (reply_ok_rset st) (get_helo in_stream'))
              | cmd_noop ⇒
                    (seq (reply_ok_noop st) (get_helo in_stream'))
              | (cmd_helo arg) ⇒
                    (seqb (do_helo arg st)
                          [x:bool]if x then (get_mail_from in_stream')
                                    else (get_helo in_stream'))
              | (cmd_rcpt_to b) ⇒
                    (seq (reply_no_mail_from st) (get_helo in_stream'))
              | _ ⇒
                    (seq (reply_no_helo st) (get_helo in_stream'))
              end)
        end))
```

Fig. 6. Coq *get_helo* function

Not all methods are converted as faithfully as our example. Indeed, the method `get_helo` is specific to the problem at hand and requires careful conversion for the verification to be meaningful. In contrast, methods such as `get_cmd` and `get_arg` here are of more general use and can be seen as trusted utility methods. From the viewpoint of our verification, the interesting point about such methods is rather that their execution may cause some errors. As a consequence, *get_cmd* and *get_arg* are simply converted as generators of exceptions in a non-deterministic manner.

Difficulties in Automation Although the code conversion described here can be applied to many programs, it seems difficult to automate it because it requires human intervention. The latter is for instance necessary to properly model the execution environment and in particular to insert non-determinism at relevant places. In the case of theorem provers such as Coq, human intervention is also required to build a tractable model. As an illustration, the fact that the input stream of SMTP commands is made a parameter of the function *get_helo* is linked to the fact that Coq allows recursive functions to be defined only by structural induction.

3.2 Specification

We have already stated informally the properties we verify in Sect. 2.3. In this section, we formally write those properties in Coq using *inductive predicates* (similar to predicates in logic). Those objects belong to the type *Prop* of logical propositions in Coq. Before, we explain how we formalize the correctness of SMTP protocol sessions and acknowledged and saved mails.

Correct SMTP Protocol Sessions The relevant Internet standard is the RFC 821[12]. It is a prose description that explains the SMTP protocol, including definitions of correct commands and replies.

By way of example, we show below how correct SMTP protocol sessions are represented in Coq. Let us assume that we are given simple predicates that state the correctness of individual SMTP commands. For instance, *(valid_cmd_helo s)* states that s is a correct HELO command. Correct SMTP protocol sessions satisfy the following inductive predicate (not displayed entirely by lack of space):

```
Inductive valid_protocol: InputStream → Prop :=
        rcv_helo: (s: InputStream)(c: SMTP_cmd)
                      (valid_cmd_helo c)
                      → (valid_after_helo s)
                      →(valid_protocol (cons_stream c s))
      | rcv_quit:
                (s: InputStream)(valid_protocol (cons_stream cmd_quit s))
      | rcv_skip: (s: InputStream)(c: SMTP_cmd)
                      ¬(valid_cmd_helo c)
                      → ¬c=cmd_quit → ¬c=cmd_abort
                      → (valid_protocol s)
                      → (valid_protocol (cons_stream c s))
with valid_after_helo: InputStream → Prop :=
  ...
```

Intuitively, it can be read as follows. There are three cases distinguished by the constructors: (1) an SMTP protocol session that starts with a correct HELO command such that the rest of the session is also correct is itself correct, (2) an SMTP protocol session that starts with a QUIT command is always correct and (3) an SMTP protocol session that starts with any other command such that the rest of the session is also correct is correct.

Similarly, correct replies are represented by an inductive predicate called *correct_reply*.

Messages Stored in the Mail Queue To specify the reliability property, we need to represent on the one hand mails for which reception is acknowledged by the mail server and on the other hand mails that are indeed saved in the mail queue.

Acknowledged mails are represented by the function *received_mails*.

Saved mails are represented by the following inductive predicate:

```
Inductive all_mails_saved_in_file: (list Mail)→FileSystem→FileSystem→Prop:=
     all_saved_none: (fs1:FileSystem)(fs2:FileSystem)
                              (eq_fs_except_garbage fs1 fs2)
                                 → (all_mails_saved_in_file (nil Mail) fs1 fs2)
  | all_saved_some: (m:Mail)(mails: (list Mail))
                              (fs1:FileSystem)(fs2:FileSystem)(fs3:FileSystem)
                              (mail_saved_in_file m fs1 fs2) →
                              (all_mails_saved_in_file mails fs2 fs3) →
                              (all_mails_saved_in_file (cons Mail m mails) fs1 fs3).
```

This requires some explanations. *FileSystem* represents the file system through which the mail queue is actually implemented (the field *files* in the record *STATE* represents the file system in our model). The predicate *eq_fs_except_garbage* is true for file systems whose non-empty files have the same contents and the predicate *mail_saved_in_files* is true if the envelop and the mail contents have been saved in the file system.

Verified Properties Equipped with above predicates, we can now write the formal specification of the properties we want to verify. They appear as Coq theorems. In the following, parenthesized parameters and the *EX* constructor represent respectively universally and existentially quantified variables. Other logical symbols (→, =, ∧, ∨) have their usual meaning.

- Adequacy with the SMTP protocol:
  - The server handles correct SMTP protocol sessions unless a fatal error occurs:

```
Theorem valid_protocol1:
   (s: InputStream)(st:STATE)
         (valid_protocol s) → (is_succ_or_fatal (work s st)).
```

  - The server rejects erroneous SMTP protocol sessions:

```
Theorem valid_protocol2:
         (s: InputStream)(st:STATE)(st':STATE)
            (work s st)=(succ st')→ (valid_protocol s).
```

  - The server sends correct replies:

```
Theorem correct_reply1:
   (s: InputStream)(st: STATE)(st': STATE)
     ((work s st)=(succ st')∨
        (work s st)=(fail empty_stream_exception st'))→
         (correct_reply s (to_client st')).
```

- The server sends correct replies up to failure:

  Theorem *correct_reply2*:

  $(s$: *InputStream*$)(st$: *STATE*$)(st'$: *STATE*$)(exn$: *Exception*$)$

  $($*work s st*$)=($*fail exn st'*$) \rightarrow$

  $($*EX s'*: *InputStream* $|$

  $($*is_prefix SMTP_cmd s' s*$) \wedge$

  $($*correct_reply s'* $($*to_client st'*$)))$.

– Reliability: Once the server acknowledges reception of a message, the latter is saved in the mail queue and is not lost:

  Theorem *reliability*:

  $(s$: *InputStream*$)(st$: *STATE*$)(st'$: *STATE*$)(exn$: *Exception*$)$

  $(($*work s st*$)=($*succ st'*$) \vee ($*work s st*$)=($*fail exn st'*$)) \rightarrow$

  $($*all_mails_saved_in_file*

  $($*received_mails s* $($*to_client st'*$))$ $($*files st*$)$ $($*files st'*$))$.

4 Verification Results

In this section, we present the results of the verification in Coq of the specification and the model presented above.

The SMTP receiver part of the secure mail system roughly consists of 700 lines of Java, excluding utility code such as parsing which is taken for granted. Our model only accounts for the core part of the SMTP receiver and also roughly consists of 700 lines in the Coq language. We believe that the code and its model grow in size similarly.

The official documentation for the SMTP protocol [12] is 4050 lines long. Our specification only accounts for that part of the documentation describing a simple SMTP receiver and is roughly 500 lines long. Although it is hard to compare prose documentation with formal specifications, we believe that the latter is a concise alternative to the former.

The size of proof scripts are given below (there is a base of common lemmas used throughout the other proofs):

Files	Size (lines)
Lemmas	2324
valid_protocol1	960
valid_protocol2	2842
correct_reply1	5278
correct_reply2	4116
reliability	2071
Total	17591

Although we tried hard to limit the size of proofs, much experience is required to write short proofs in Coq. Sizes we report here should therefore be seen as upper bounds. Yet, it is likely that proof scripts grow quickly with the size of involved inductive predicates.

The model, the specification and the proofs are available at `http://web.yl.is.s.u-tokyo.ac.jp/~affeldt/mail-system.tar.gz`.

The authors (who are not experienced Coq users) have carried out the verification described in this paper in a few months of sparse work. We believe that one person working in optimal conditions may need roughly 150 hours.

Coq 7.1 requires 7.3 minutes (according to Coq's `Time` command) and 157MB of memory (according to Unix's `top` command) to check the proofs (operating system: Solaris 8, architecture: UltraSparc 400MHz)

The main result is that verification has proved to be effective. Indeed, errors were found when building proofs. They appeared as contradictory hypotheses indicating some inconsistency in the state of the mail server. Once the offending operation is identified in the Coq model, the Java code can be immediately corrected accordingly thanks to the program transformation described in the previous section. To be more precise, verification of the mail server allowed us to find the following errors in the Java code base:

– the state of the mail system was not reset upon mail reception and wrongly reset upon reception of the RSET command, and
– a wrong number of reply messages were sent back to the client.

Those errors are only revealed by specific sequences of SMTP commands, that is the reason why they slip through a non-exhaustive testing procedure.

5 Discussion

5.1 Limitations

A first limitation of our approach is that there is a small gap between the implementation and its model. However, alternative approaches also have drawbacks. One can implement the mail server directly in the theorem prover to prove its properties and eventually use an extraction facility to generate a runnable program. Yet, (1) the extracted code may not be directly runnable (because the model contains non-software aspects such as a model of the program environment) and (2) the extracted code is unlikely to be efficient (because handling code optimizations complicates the proof and because extraction facilities are non-optimizing code generators). One can choose a radical approach by formalizing the semantics of an efficient programming language chosen for the implementation as a preliminary step. Yet, (1) it results in a long and intricate development and (2) it may not even be possible since most languages are only described in prose. Another argument in favor of our approach is that it is still possible to defer proof that the model is faithful to the implementation to a later stage of software development.

A second limitation of our approach in general is that it does not deal with threads. However, it is not a major problem here because the processing operated by the SMTP receiver part of the mail server is confined to a single thread.

A last limitation that applies to formal verification in general is that there may be bugs in the specifications we write and in the tools we use (although here we only need to trust the smaller proof checker of Coq).

5.2 Lessons Learned

The main lesson is taught by results in Sect. 4: formal verification of system softwares is feasible (although proofs may become tedious) and useful (since it may uncover errors).

Another lesson is the practical importance of proof modularity because it decreases the size of proofs and facilitates their maintenance. The following two examples illustrate those points.

Reusable lemmas allow for shorter proofs. In the case of the model discussed in Sect. 3.1, when we know that the execution of a sequence of two statements ends up with a success, we may need to show that there is an intermediate successful state. It is possible to unfold objects of the formalization to directly exhibit this state, but it is shorter to use the following lemma:

```
Lemma lem_seq_succ1:
        (v:Result)(g:STATE→Result)(st:STATE)
          (seq v g)=(succ st)→
          (EX st1:STATE |
            (EX st2:STATE |
                v=(succ st1) ∧ (g st2)=(succ st) ∧ (eqstate st1 st2))).
```

Changes in the actual code base must be reflected in the model; by encapsulating details that are susceptible to change in lemmas, we facilitate maintenance of proofs. For instance, when we know that a call to the *do_helo* function (the Coq counterpart of the Java `do_helo` method, see Fig. 3 and 6) is successful, we may need to show that the HELO command is well-formed or that addresses of recipients are unchanged. It is possible to unfold objects to exhibit directly those properties but the proof would have to be modified whenever the body of the *do_helo* is changed. In contrast, the proof may not need to be changed if we use the following lemma:

```
Lemma lem_do_helo:
        (s:String)(st:STATE)(b:bool)(st':STATE)
          (do_helo s st)=(Succ bool b st')→
            ((is_nullstr s)=(negb b) ∧ (fwd_paths st)=(fwd_paths st')).
```

Another lesson is the practical importance of support for dealing with redundancies in proofs. Indeed, the proofs made here involve many similar case analyses which are unfortunately difficult to automate with Coq.

A last lesson is that our difficulty to deal with threads emphasizes the importance of proof systems for concurrent programs.

6 Related Work

There are few experiments of verification of existing programs using theorem proving. Most work targets idealized subparts: verification of stripped down algorithms (e.g., [10]), verification of properties of network protocols (e.g., [8]), verification of compilers for languages subsets (e.g., [2]), etc.

Pierce and Vouillon specify a file synchronizer and prove its properties [11] with Coq. The model for file synchronization is a function that takes two possibly inconsistent trees of files as input and outputs a single, 'synchronized' one. In comparison, a mail server is more difficult to model because it is a reactive program dealing with multiple entities communicating by means of messages. Intents are also different: we write our model in order to match the actual code base whereas they do it for the purpose of generating an idealized implementation.

Thttpd is a freely available http daemon that puts an emphasis on security [7]. Black and Windley discuss inference rules of the axiomatic semantics of C with the aim of mechanically verifying a small version (100 lines of C source code) of thttpd [4] with a theorem prover. Verification has been eventually achieved by Black in his Ph.D. thesis [3]. In contrast, we avoid the burden of formalizing such a semantics thanks to a program transformation that allows us to handle a larger program, independently of the implementation language.

Filliâtre [9] studies certification of imperative programs in type theory. One of his achievement is a Coq tactic that generates proof obligations given a program written in some imperative language. The underlying machinery actually makes use of a transformation to an intermediate functional representation using effects and monads. This is reminiscent of the manual transformation we perform here and indicates that it can be automated to a certain extent. However, full automation requires to overcome several difficulties such as non-determinism and general recursion as discussed at the end of section 3.1.

There are several implementation of mail servers. In particular, Dan Bernstein wrote qmail [1] with security and reliability in mind. For instance, qmail's so-called 'straight-paper path' philosophy ensures that accepted mails cannot be lost by design, which corresponds to our reliability property.

Model checking is another well-established approach to verification of system softwares [6]. In the case of a mail server, complex data manipulation is a source of state explosion that makes it not immediate to apply model checking techniques. Of course, it is possible to handle model checking of even infinite state systems by using, for instance, abstraction techniques (e.g., [5]). However, such an approach leads us away from the faithful code transformation we also think is important here. This is the reason that makes us prefer theorem proving to model checking in this paper.

7 Conclusion

In this paper, we report an experiment of verification of the SMTP receiver part of the secure mail system. Such a verification is very important from the viewpoint of security and challenging because of the size of the application. The verification is carried out by the Coq theorem prover. Our approach is to translate manually the actual code base written in Java into a model written in the Coq language. The techniques we use ensure that the verification of the model substitutes for the verification of the actual code base. Then we specify the correctness properties using inductive predicates and prove them mechanically in Coq. Although much effort and care is needed to write proofs, results attest that our approach is feasible and useful in practice, since errors in the actual code base were indeed found during verification.

The verification is in progress. We now have to verify the rest of the mail server (in particular the SMTP sender part, but also some procedures such as parsing whose correctness has been taken for granted). Eventually, we would like to prove formally the correspondence between the Java code and the Coq function, so as to overcome one major limitation of our approach. Concerning applicability to other system softwares, a solution should be devised to handle concurrency. More broadly speaking, the size of proofs in our experiment indicates that our approach to formal verification may become unreasonably tedious. In such situations, a combination of theorem proving and model checking may be worth exploring.

Acknowledgements The authors are grateful to Kenjiro Taura for providing the source code with detailed explanations and suggestions about verification.

References

1. D. Bernstein and various contributors. The qmail home page. `http://www.qmail.org`.
2. Y. Bertot. A certified compiler for an imperative language. Technical Report RR-3488, INRIA, Sep. 1998.
3. P. E. Black. *Axiomatic Semantics Verification of a Secure Web Server*. PhD thesis, Department of Computer Science, Brigham Young University, Feb. 1998.
4. P. E. Black and P. J. Windley. Inference rules for programming languages with side effects in expressions. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1125 of *Lecture Notes in Computer Science*, pages 51–60. Springer-Verlag, August 1996.
5. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.
6. E. M. Clarke and J. M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.
7. F. B. Cohen. Why is thttpd secure? Available at `http://www.all.net/journal/white/whitepaper.html`.
8. B. Dutertre and S. Schneider. Using a PVS embedding of CSP to verify authentication protocols. In E. L. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics*, volume 1275 of *Lecture Notes in Computer Science*, pages 121–136. Springer-Verlag, Aug. 1997.
9. J.-C. Filliâtre. *Preuve de programmes impératifs en théorie des types*. PhD thesis, Université Paris-Sud, Jul. 1999. Available at `http://www.lri.fr/~filliatr/ftp/publis/these.ps.gz`.
10. D. Nazareth and T. Nipkow. Formal verification of algorithm W: The monomorphic case. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1125 of *Lecture Notes in Computer Science*, pages 331–345. Springer-Verlag, Aug. 1996.

11. B. C. Pierce and J. Vouillon. Specifying a file synchronizer (full version). Draft, Mar. 2002.
12. J. B. Postel. Rfc 821: Simple mail transfer protocol. Available at `http://www.faqs.org/rfcs/rfc821.html`, Aug. 1982.
13. E. Shibayama, S. Hagihara, N. Kobayashi, S. Nishizaki, K. Taura, and T. Watanabe. AnZenMail: A secure and certified e-mail system. In M. Okada, B. Pierce, A. Scedrov, H. Tokuda, and A. Yonezawa, editors, *Proceedings of International Symposium on Software Security, Keio University, Tokyo, Japan (Nov. 2002)*, Lecture Notes in Computer Science. Springer-Verlag, Feb. 2003.
14. The Coq Development Team. The Coq proof assistant reference manual. Available at `http://coq.inria.fr/doc/main.html`, 2002.
15. P. Wadler. Monads for functional programming. In M. Broy, editor, *Marktoberdorf Summer School on Program Design Calculi*, volume 118 of *NATO ASI Series F: Computer and systems sciences.* Springer-Verlag, Aug. 1992. Also in J. Jeuring and E. Meijer, editors, Advanced Functional Programming, Springer Verlag, LNCS 925, 1995. Some errata fixed August 2001.