Formalization and Verification of a Mail Server in Coq

Reynald Affeldt[a] and Naoki Kobayashi[b]

[a]Department of Computer Science, University of Tokyo
[b]Department of Computer Science, Tokyo Institute of Technology

Verification of System Software

- Most critical systems rely on software
  (traffic control, financial transactions, etc.)
- Software errors may result in disasters
  (Ariane 5, Therac-25, etc.)
- Testing cannot guarantee the absence of errors

$\Rightarrow$ Formal verification is necessary

Verification of a Mail Server

- Motivation :
  Verification for **midsize** system softwares
- Case study: Electronic mail
  - Widely used in business
  - Costly security holes:
    CodeRed / IIS Server → US$2.6 billions [a]

[a] source: Computer Economics, Inc.

Our Approach

1. Pick up the AnZenMail mail server [Shibayama, Taura et al. 2002]
2. Write reliability specifications
3. Prove the implementation meets them

IOW, **Proof** that a **program** has certain **properties**

$\Rightarrow$ Coq (logical framework + proof assistant)

Contributions

- Formal verification of (a part of) the AnZenMail mail server
- Demonstrate usefulness and feasibility of our approach
- Show techniques for narrowing the "implementation-model" gap

"Implementation-model" gap?

Goal of verification: **Implementation** in Java

Means of verification: **Model** in Coq

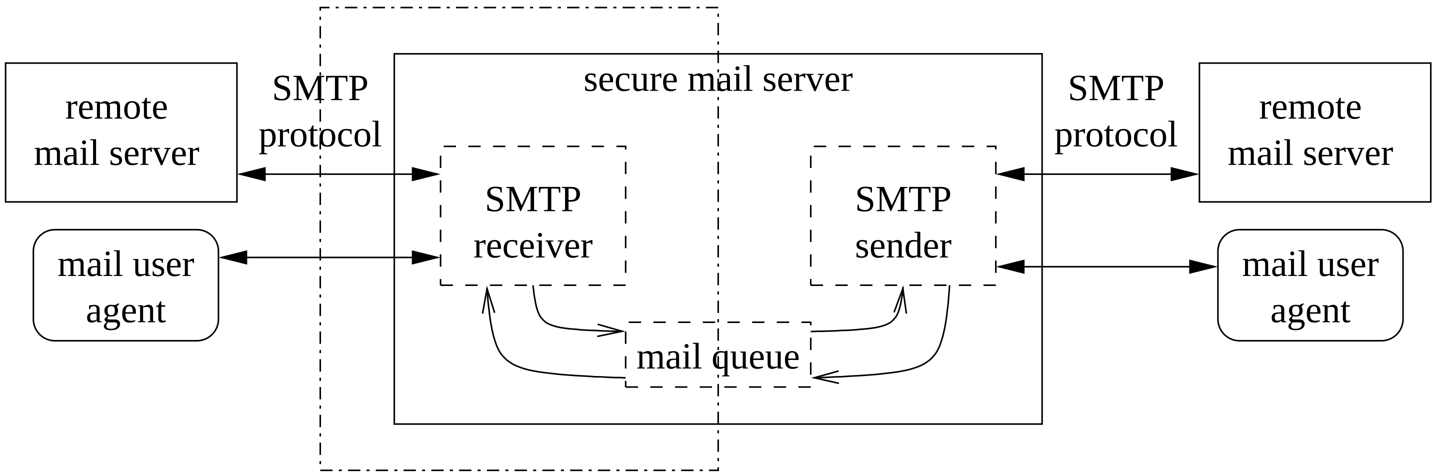
Outline

1. **Introduction to SMTP**
2. Modelization
3. Specifications
4. Results
5. Conclusion

A Client/Server Protocol

Mail system:

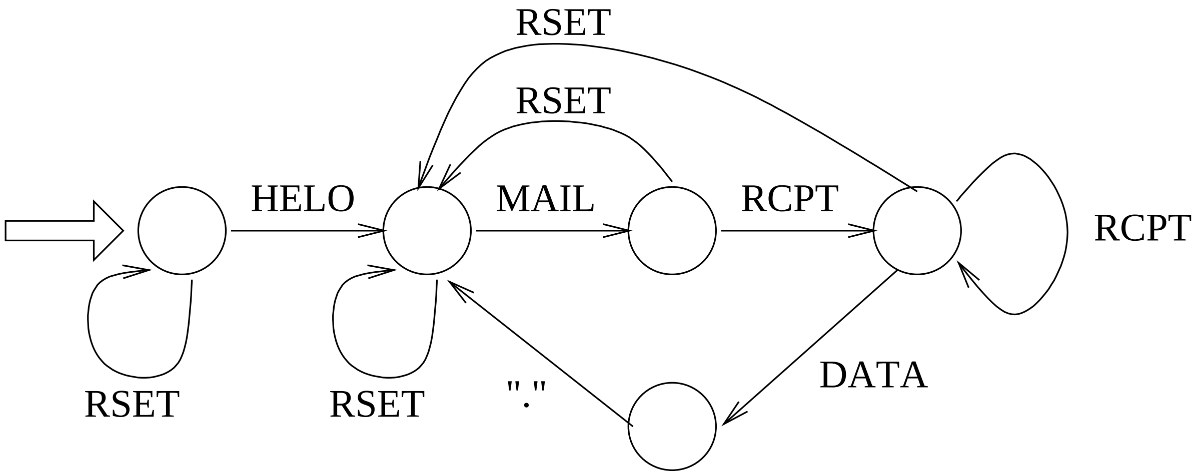
- Mail servers:
  - **SMTP receiver**
  - SMTP sender
- Mail clients



SMTP Protocol Sessions

SMTP session[a]:

- SMTP commands:



- SMTP replies:
  - Acknowledgments
  - Error messages

[a] full specification: RFC 821

Outline

1. Introduction to SMTP
2. **Modelization**
3. Specifications
4. Results
5. Conclusion

Modelization Overview

- From Java to Coq
- Useful verification

$\Rightarrow$ Narrow the "implementation-model" gap

$\Rightarrow$ **Faithful code conversion**

Difficulties:

1. Java is imperative whereas Coq is functional
2. Explicit relevant non-software specific aspects (e.g., non-deterministic system errors)

Code Conversion Basis (1/2)

Java datatypes → Coq types

For instance, SMTP commands:

```
int cmd_helo = 0;
int cmd_mail_from = 1;
int cmd_rcpt_to = 2;
int cmd_data = 3;
int cmd_noop = 4;
int cmd_rset = 5;
int cmd_quit = 6;
int cmd_abort = 100;
int cmd_unknown = 101;
```

→

Inductive $SMTP\_cmd$: Set :=

cmd_helo: $String \rightarrow SMTP\_cmd$

| cmd_mail_from: $String \rightarrow SMTP\_cmd$

| cmd_rcpt_to: $String \rightarrow SMTP\_cmd$

| cmd_data: $String \rightarrow SMTP\_cmd$

| cmd_noop: $SMTP\_cmd$

| cmd_rset: $SMTP\_cmd$

| cmd_quit: $SMTP\_cmd$

| cmd_abort: $SMTP\_cmd$

| $cmd\_unknown$: $SMTP\_cmd$.

Code Conversion Basis (2/2)

Java control structures → Coq control structures

For instance, switch statements:

```
switch(cmd) {
  case cmd_unknown: /* ... */
  case cmd_abort: /* ... */
  case cmd_quit: /* ... */
  case cmd_rset: /* ... */
  case cmd_noop: /* ... */
  case cmd_helo: /* ... */
  case cmd_rcpt_to: /* ... */
  default: /* ... */
}
```

⟶

```
(Cases m of
  cmd_unknown ⇒(* ... *)
  | cmd_abort ⇒(* ... *)
  | cmd_quit ⇒(* ... *)
  | cmd_rset ⇒(* ... *)
  | cmd_noop ⇒(* ... *)
  | (cmd_helo arg) ⇒(* ... *)
  | (cmd_rcpt_to b) ⇒(* ... *)
  | _ ⇒(* ... *)
end)
```

Modeling System Errors

- **Several kinds** (recoverable network errors, fatal host computer failures, etc.)
  $\Rightarrow$ Representation as exceptions:

  Inductive $Exception$: Set :=
  $IOException$: $Exception$
  | $parse\_error\_exception$: $Exception$
  | $Smail\_implementation\_exception$: $Exception$
  | $empty\_stream\_exception$: $Exception$
  | $system\_failure$: $Exception$.

- **Non-deterministic**
  $\Rightarrow$ Representation as test oracles:

  CoInductive $Set\ Oracles := flip : bool \rightarrow Oracles \rightarrow Oracles$.

Put It All Together (1/2)

Exceptions + test oracles + **global state**

⇒ **Monadic style programming**:

- A type for computation results:

  Definition $Result : Set := (Except\ unit)$.

  Inductive $Except\ [A\colon Set]\colon Set :=$

  $\quad Succ\colon A \rightarrow STATE \rightarrow (Except\ A)$

  $\quad \mid Fail\colon Exception \rightarrow STATE \rightarrow (Except\ A)$.

- A function for sequential execution:

  Definition $seq\colon Result \rightarrow (STATE{\rightarrow}Result) \rightarrow Result :=$ ...

⇒ Application to code conversion:

`a;b` → *(seq a b)*

Put It All Together (2/2)

Concretely[a]:

Definition seq: $Result \rightarrow (STATE \rightarrow Result) \rightarrow Result :=$

$[x{:}\ Result][f{:}STATE \rightarrow Result]$

(* **the first statement may be a success or a failure** *)

$(Cases\ x\ of$

$(Succ\ \_\ st) \Rightarrow$

(* **the host computer may fail** *)

$Cases\ (oracles\ st)\ of$

$(flip\ true\ coin) \Rightarrow (f\ (update\_coin\ st\ coin))$

$|\ (flip\ false\ coin) \Rightarrow (Fail\ unit\ system\_failure\ st)$

end

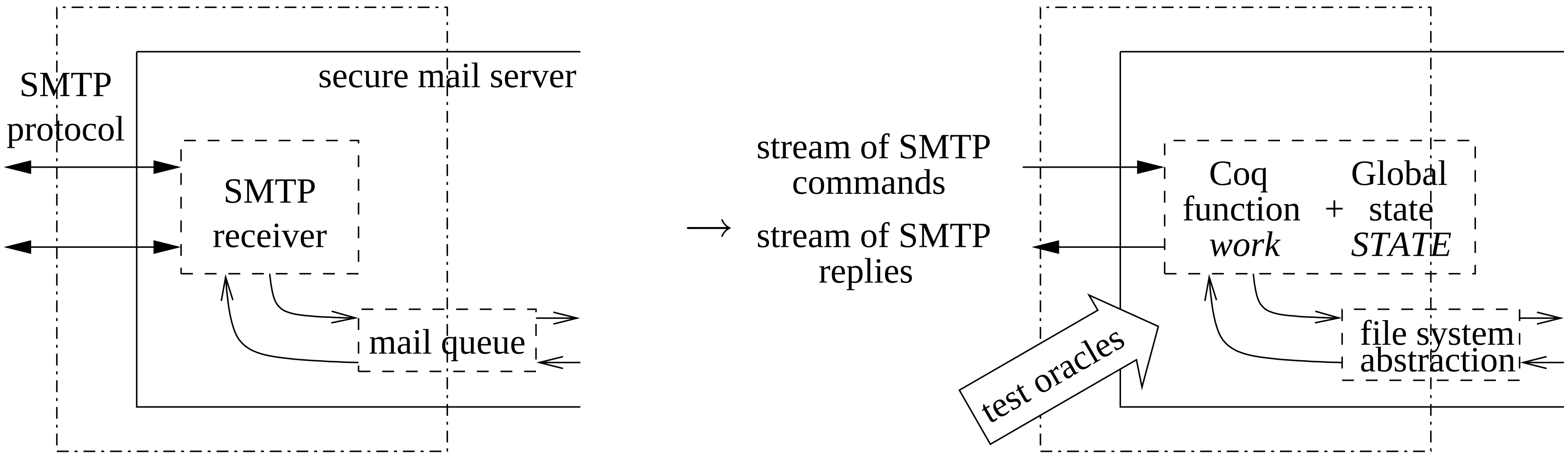
$|\ (Fail\ e\ st) \Rightarrow (Fail\ unit\ e\ st)$

$end).$

---

[a] see the paper for detailed explanations

Model Summary



Properties preserved by modelization:

- The structure of the source code
- Non-determinism for system errors

⇒ “Implementation-model” match

Outline

1. Introduction to SMTP
2. Modelization
3. **Specifications**
   (a) Verified Properties
   (b) Formal Statements
4. Results
5. Conclusion

Verified Properties

Program properties expressed modulo system errors:

- Compliance to **standard** protocols
  - The server accepts correct SMTP commands unless a fatal error occurs
  - The server sends back correct SMTP replies
  - The server rejects wrong SMTP commands
- **Reliability** of the provided service
  - Accepted mails are not lost even if a system error occurs

A Formal Statement

The server accepts correct SMTP commands <u>unless</u> a fatal error occurs:

Theorem $accept\_SMTP$:
$(s\colon InputStream)(st{:}STATE)$
$$(valid\_protocol\ s) \rightarrow (is\_succ\_or\_fatal\ (work\ s\ st)).$$

Basic definitions:

- $(valid\_protocol\ s)$: SMTP commands s are correct[a]
- $(is\_succ\_or\_fatal\ r)$: result r is a success or a fatal error

[a] as defined in RFC 821

Another Formal Statement

Accepted mails are not lost even if a system error occurs:

Theorem *reliability*:

$$(s\text{: } InputStream)(st\text{: } STATE)(st'\text{: } STATE)(exn\text{: } Exception)$$
$$((work\ s\ st)=(succ\ st') \lor (work\ s\ st)=(fail\ exn\ st')) \rightarrow$$
$$(all\_mails\_saved\_in\_file$$
$$(received\_mails\ s\ (to\_client\ st'))\ (files\ st)\ (files\ st')).$$

Basic definitions:

- *(received_mails s r)*: accepted mails
- *(all_mails_saved_in_file m fs' fs)*: saved mails

Outline

1. Introduction to SMTP
2. Modelization
3. Specifications
4. **Results**
5. Conclusion

Verification is Useful

- Bugs found in the implementation:
  - Resetting of the state of the mail server
  - Number of SMTP replies
- Formal specifications in themselves
  (Debatable comparison:
  SMTP RFC in prose $\simeq$ 4050 lines
  Specifications in Coq $\simeq$ 500 lines)

Verification is Feasible

- Size:
  - Java implementation $\simeq$ 700 lines
  - Coq model $\simeq$ 700 lines
  - Proofs scripts $\simeq$ 18,000 lines
- Time:
  - Full development $\simeq$ 150 hours for 1 person
  - Proof check $\simeq$ 7.3 minutes
    (Coq 7.1, UltraSparc 400MHz)

Application to Other System Softwares

- Any implementation language is ok
- Systematic (though manual) code conversion
- Proofs done in parallel with code development

Possible issues:

- No support for threads (not a problem here)
- Size of proofs (solutions: modularity, automation, libraries)
- There may be errors in specifications

Outline

1. Introduction to SMTP
2. Modelization
3. Specifications
4. Results
5. **Conclusion**

Related Work (1/2)

- Formal verification of **algorithms**:
  Many experiments
  (often tailored for formal verification)
- Formal verification of **implementations**:
  - Thttpd [Black 1998]
    Proofs of security for an http daemon
    About 100 lines of C code
  - Unison [Pierce and Vouillon 2002]
    Program for file synchronization
    Certified reference implementation in Coq

Related Work (2/2)

- Code conversion:
  - Correctness tactic in Coq [Filliatre 1999]
    Semi-automatic certification of imperative programs
- Secure electronic mail:
  - AnZenMail [Shibayama, Taura et al. 2002]
  - qmail [Bernstein et al.]
    Straight-paper-path philosophy

Conclusion

Verification for **midsize** system softwares in **Coq**:

- "Implementation-model" match:
  - Faithful code conversion
  - Failure-conscious modelization
- Useful and feasible in practice

Future work:

- Verification of the SMTP sender
- Modularity and redundancy in Coq proofs
- Support for concurrency