Formalization and Verification of a Mail Server in Coq

Reynald Affeldt\textsuperscript{a} and Naoki Kobayashi\textsuperscript{b}

\textsuperscript{a}Department of Computer Science, University of Tokyo
\textsuperscript{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

⇒ 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 $\rightarrow$ US$2.6$ billions

\(^a\)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

⇒ 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:

- SMTP commands:
  - HELO
  - RCPT
  - DATA
  - RSET

- SMTP replies:
  - Acknowledgments
  - Error messages

(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

⇒ Narrow the “implementation-model” gap
⇒ 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 \( SMTP\_cmd : Set \) :=

- \( cmd\_helo : String \rightarrow SMTP\_cmd \)
- \( cmd\_mail\_from : String \rightarrow SMTP\_cmd \)
- \( cmd\_rcpt\_to : String \rightarrow SMTP\_cmd \)
- \( cmd\_data : String \rightarrow SMTP\_cmd \)
- \( cmd\_noop : SMTP\_cmd \)
- \( cmd\_rset : SMTP\_cmd \)
- \( cmd\_quit : SMTP\_cmd \)
- \( cmd\_abort : SMTP\_cmd \)
- \( cmd\_unknown : SMTP\_cmd \).
Java control structures → Coq control structures

For instance, switch statements:

\[
\text{switch (cmd) } \left\{ \\
\text{case cmd\textunderscore unknown: } / * \ldots * / \quad \text{cmd\textunderscore unknown ⇒}(* \ldots *) \\
\text{case cmd\textunderscore abort: } / * \ldots * / \quad \text{cmd\textunderscore abort ⇒}(* \ldots *) \\
\text{case cmd\textunderscore quit: } / * \ldots * / \quad \text{cmd\textunderscore quit ⇒}(* \ldots *) \\
\text{case cmd\textunderscore rset: } / * \ldots * / \quad \text{cmd\textunderscore rset ⇒}(* \ldots *) \\
\text{case cmd\textunderscore noop: } / * \ldots * / \quad \text{cmd\textunderscore noop ⇒}(* \ldots *) \\
\text{case cmd\textunderscore helo: } / * \ldots * / \quad (\text{cmd\textunderscore helo arg}) ⇒(* \ldots *) \\
\text{case cmd\textunderscore rcpt\textunderscore to: } / * \ldots * / \quad (\text{cmd\textunderscore rcpt\textunderscore to b}) ⇒(* \ldots *) \\
\text{default: } / * \ldots * / \quad \_ ⇒(* \ldots *) \\
\right. \\
\text{end}
\]
Modeling System Errors

• **Several kinds** (recoverable network errors, fatal host computer failures, etc.)

  ⇒ 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**

  ⇒ Representation as test oracles:

  CoInductive *Set Oracles := flip : bool → Oracles → 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: Set]: Set :=
    Succ: A → STATE → (Except A)
    | Fail: Exception → STATE → (Except A).

- A function for sequential execution:
  Definition seq: Result → (STATE→Result) → Result := ...

⇒ Application to code conversion:

\[ a; b \rightarrow (seq\ a\ b) \]
Put It All Together (2/2)

Concretely\(^a\):

Definition \(\text{seq}: \text{Result} \rightarrow (\text{STATE} \rightarrow \text{Result}) \rightarrow \text{Result} := \)

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

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

\((\text{Cases } x \text{ of} \)

\((\text{Succ } st) \Rightarrow \)

(* the host computer may fail *)

\(\text{Cases } (\text{oracles st}) \text{ of} \)

\((\text{flip true coin}) \Rightarrow (f (\text{update_coin st coin})))

| \((\text{flip false coin}) \Rightarrow (\text{Fail unit system_failure st}) \)

end

| \((\text{Fail e st}) \Rightarrow (\text{Fail unit e st}) \)

end).

\(^a\text{see the paper for detailed explanations}\)
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 unless a fatal error occurs:

Theorem accept_SMTP:
\[
(s : \text{InputStream})(\text{st : STATE})
\]
\[
(\text{valid\_protocol } s) \rightarrow (\text{is\_succ\_or\_fatal } (\text{work } s \text{ st})).
\]

Basic definitions:

- \((\text{valid\_protocol } s)\): SMTP commands \(s\) are correct\(^a\)
- \((\text{is\_succ\_or\_fatal } r)\): 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:

\[(s: \text{InputStream})(st: \text{STATE})(st': \text{STATE})(exn: \text{Exception})
\]
\[\vdash ((\text{work } s \text{ st}) = (\text{succ } st') \lor (\text{work } s \text{ st}) = (\text{fail } exn \text{ st'})) \rightarrow
\]
\[(\text{all-mails-saved-in-file}
\]
\[\quad (\text{received-mails } s \ (\text{to-client } st')) \ (\text{files } st) \ (\text{files } st')).\]

Basic definitions:

- \((\text{received-mails } s \ r)\): accepted mails
- \((\text{all-mails-saved-in-file } m \ fs' \ fs)\): 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 $\approx$ 4050 lines Specifications in Coq $\approx$ 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