Modeling A Certified Email Protocol using I/O Automata

C. Blundo, S. Cimato, R. De Prisco, A. L. Ferrara

Dipartimento di Informatica ed Applicazioni
Università di Salerno
84081 Baronissi (SA)

Abstract

Describing and reasoning about asynchronous distributed systems is often a difficult and error prone task. In this paper we experiment the Input/Output Automata framework as a tool to describe and reason about cryptographic protocols running in an asynchronous distributed system. We examine a simple certified email protocol [5], give its formalization using the IOA model, and prove that some security properties are satisfied during the execution of the protocol.

1 Introduction

With the spreading diffusion of the Internet and the World Wide Web, our society is becoming more and more dependent on communication data which are transmitted over computer networks. A large number of transactions involving a growing number of people has been actually replaced by their digital analogues, in which electronic "objects" are exchanged among two or more parties. An example comes from the diffusion of the electronic mail service which allows users to exchange messages containing text or multimedia files.

Because of its features, such as low cost, rapidity and accessibility, the email service is increasingly used in place of ordinary mail. In many cases, email messages are recognized as receipts or evidences of online transactions.
such as buying airline tickets, or submissions of papers for publications in conferences or journals, and so on. However the use of email poses some problems, since in its simplest form the email service does not have many features that are usually required in such cases. The standard email service is based on the Simple Mail Transfer Protocol [12] and Post Office Protocol [10], which do not offer guarantees on the delivery and the integrity of the messages. Messages are usually stored and transmitted in plain text allowing a malicious adversary to tap the connection during the transfer and making him able to access sensible data.

In order to provide some form of protection, cryptographic techniques have been employed to obtain additional guarantees on the email service. A number of certified email protocols has been presented in literature, ensuring that the message exchange procedure provides the participants with different security properties. Usually such protocols involve a trusted third party (TTP for brief) which controls the behavior of the participants, helping them in the message exchange, and resolving any dispute if necessary. According to the role played by the TTP, protocols have been classified as inline or optimistic. In inline protocols [3,5,14,15], the TTP is actively involved in each message exchange. In optimistic protocols [1,2,9], the sender and the receiver perform the message exchange without the intervention of the TTP but they can invoke the TTP to resolve any dispute, caused for example by a cheating attempt from one of the party.

In this paper we analyze Deng’s certified email protocol [5], and present its formal model relying on the Input/Output Automaton [7], (IOA for brief), framework. IOA provides a framework allowing both a precise description of the code and the possibility of very detailed proofs [6,13]. The aim of the work is to use the IOA as a tool to describe and to reason about cryptographic protocols running in an asynchronous distributed system. In this perspective, to perform the analysis, we consider a scenario in which the participants to the protocol are modeled as interacting nodes in a distributed system. Their behaviour is then described through IOA automata. The IOA model is then used to prove that some security properties are satisfied during the execution of the protocol.

The IOA formalism has been previously employed for the modeling and the analysis of security protocols in [8], where the correctness of a simple shared key communication protocol and the Diffie-Hellmann key distribution protocol has been proved. The security of Asokan’s certified email protocol [1] has been analyzed in [11], where a formal model relying on simulatability and probabilistic state-transition machines is employed.

This paper is organized as follows. In the next section, we introduce the
framework we consider to analyze the protocol, presenting the setting and
cryptographic primitives used during the execution of the protocol. In Sec-
tion 3 we describe Deng’s email protocol, and in Section 4 we present its IOA
formalization. Finally, the correctness of the protocol is provided in Section
5, where non repudiation properties for the origin and destination and fair-
ness properties are shown to hold with the help of invariant assertion proofs.
Conclusions are drawn in Section 6.

2 The Framework

We consider a distributed system consisting of \( n \) nodes (processors) \( P = \{1, 2, \ldots, n\} \) and a special node, namely the Trusted Third Party (\( \text{ttp} \) for brief) which is delegated by the participants to control the behavior of the
parties, assist them during the exchange of messages and resolve any dispute
if necessary. The \( \text{ttp} \) is a fully trusted party, meaning that the senders and
the receivers have complete trust in it. Moreover, there is a communication
channel between each node of the set \( P \cup \{\text{ttp}\} \).

2.1 Cryptographic Primitives

The cryptographic primitives used in this paper are:
- \( \text{Sig}_A(m) \): denotes the digital signature of the message \( m \) using the private
  key of user \( A \) under a public-key signature algorithm;
- \( h(m) \): indicates the hash of message \( m \) using some collision resistant hashing
  scheme. A collision resistant hash function maps arbitrary length messages
to constant size messages such that it is computationally infeasible to find
any two distinct messages hashing to the same value.
- \( \text{PK}_B(m) \): denotes the encryption of message \( m \) using the public key of user
  \( B \) in some public-key encryption algorithm. The algorithm should provide
non-malleability, i.e., given a ciphertext it is impossible to generate another
ciphertext such that the respective plaintexts are related.
- \( E_k(m) \): denotes the encryption of message \( m \) using the key \( k \) under some
  symmetric encryption algorithm.

2.2 IOA Automata

In order to help the reader not familiar with IOA to understand the code we
briefly explain how to read IOA code using a simple example. An IOA is a simple type of state machine in which transitions are associated with named actions. Figure 1 shows an automaton that models a channel
for communication from node $i$ to node $j$. The state is a list of all the variables that describe the state of the automaton. For this channel the state is completely described by a variable that contains the messages still in transit on the channel.

The channel has an input action $\text{Send}(m)_{i,j}$ which is controlled by another (unspecified in the example) automaton $A$, modeling node $i$, which has the same action $\text{Send}(m)_{i,j}$ as an output action. Whenever automaton $A$ executes this action also the channel executes the action (at the same time), we will say that the action $\text{Send}$ of $A$ controls the action $\text{Send}$ of channel $i,j$. In this case the effect of the action, in the channel automaton, is to add a message in the set of in transit messages.

The channel has an output action $\text{Receive}(m)_{i,j}$ which has a precondition (a boolean condition) specifying when the action is enabled, that is when the action can be executed. An output action can be executed whenever it is enabled. Moreover, all other automata that have such an action as input will execute it. There will be an automaton $B$, modeling node $j$, that has $\text{Receive}(m)_{i,j}$ as an input action.

There are also internal actions that are similar to output actions (i.e., have a precondition and an effect) with the difference that they do not interact with other automata (i.e., several automaton may have internal actions with the same name and they are all independent). We use the notation $\text{name.var}$ to indicate variable $\text{var}$ of automaton $\text{name}$, for example $\text{channel.Msgs}$ refers to variable $\text{Msgs}$ of automaton $\text{channel.i,j}$.

Each IOA comes equipped with a partition of its locally controlled actions (output and internal actions); each equivalence class in the partition represents some task that the automaton is supposed to perform. In order for the input/output interaction to happen automata describing a system have to be composed together. The composition of several IOA is one single IOA. The execution of an IOA consists of a sequence of alternating states and transitions, beginning from a starting state. An execution is called fair if each task
gets infinitely many opportunities to perform one of its actions. Formally, an
execution fragment $\alpha$ of an IOA $A$ is said to be fair if the following conditions
hold for each class $C$ of tasks of $A$:

(i) If $\alpha$ is finite, then $C$ is not enabled in the final state of $\alpha$.

(ii) If $\alpha$ is infinite, then $\alpha$ contains either infinitely many events from $C$ or
infinitely many occurrences of states in which $C$ is not enabled.

We refer the reader to [7], Chapters 8 and 23, for more information about
the IOA models.

3 The CMPl Protocol

We now describe the CMP1 protocol for certified mail presented by Deng et
al. in [5]. A concise representation of protocol message flow is provided in
Figure 2.

$$
\begin{align*}
M_1 &= \langle S, R, \text{ttp}, h(m), PK_\text{ttp}(k), E_k(Sig_S(S, R, \text{ttp}, m)) \rangle \\
M_2 &= \langle Sig_R(S, R, \text{ttp}, h(m)), PK_\text{ttp}(k), E_k(Sig_S(S, R, \text{ttp}, m)) \rangle \\
M_3 &= \langle Sig_\text{ttp}(Sig_R(S, R, \text{ttp}, h(m))), R, m \rangle
\end{align*}
$$

Fig. 2. A concise representation of protocol message flows in CMPl.

To send a mail message containing $m$ to the Receiver $R$, the Sender $S$
first digitally signs $(S, R, \text{ttp}, m)$ with his private key to produce $Sig_S(S, R, \text{ttp}, m)$. Then, $S$ generates a session key $k$ and encrypts the signed data under $k$ using
a symmetric key cryptosystem. Finally, $S$ computes $h(m)$ and sends the message $M_1 = \langle S, R, \text{ttp}, h(m), PK_\text{ttp}(k), E_k(Sig_S(S, R, \text{ttp}, m)) \rangle$ to $R$. The clear
text part (i.e., $S, R, \text{ttp}, h(m)$) in this message serves as the mail identifier.
This message informs $R$ that there is a certified mail from $S$ to him. After
receiving this message, $R$ has two choices. He may ignore the message. In this
case, the protocol is aborted. He may choose to receive the message. In this
case, the protocol is aborted. He may choose to receive the message. In this
case, he signs $(S, R, \text{ttp}, h(m))$ using his private key and sends the message
$M_2 = \langle Sig_R(S, R, \text{ttp}, h(m)), PK_\text{ttp}(k), E_k(Sig_S(S, R, \text{ttp}, m)) \rangle$ to $\text{ttp}$. Upon
receiving this message, the $\text{ttp}$ first checks the validity of $Sig_R(S, R, \text{ttp}, h(m))$
using public key of $R$. Then, it decrypts $PK_\text{ttp}(k)$ using its private key, and
decrypts $E_k(Sig_S(S, R, \text{ttp}, m))$ using $k$. Next, the $\text{ttp}$ checks the validity of
$Sig_S(S, R, \text{ttp}, m)$ using $S$’s public key, computes $h(m)$, and compares this
$h(m)$ with the one received in $Sig_R(S, R, \text{ttp}, h(m))$. If the two values match,
the $\text{ttp}$ knows that $m$ is the mail content that $S$ wanted to send to $R$, and that
$R$ is willing to receive $m$. In this case, the $\text{ttp}$ is able to compute the messages
\[ M_3 = \langle \text{Sig}_{\text{ttp}}(\text{Sig}_R(S, R, \text{ttp}, h(m))), R, m \rangle \] corresponding to the proof-of-origin and \[ M_4 = \langle \text{Sig}_{\text{ttp}}(\text{Sig}_S(S, R, \text{ttp}, m)) \rangle \] corresponding to the proof-of-delivery and sends them to \( R \) and to \( S \), respectively.

In the next sections, by using the IOA model, we show that the protocol CMP1 meets the following requirements:

- **Non-repudiation of origin.** The protocol provides the recipient of an email with an irrefutable proof that the mail content received was the same as the one sent by the originator. This proof-of-origin can protect against any attempt by the originator to falsely deny sending that message.

- **Non-repudiation of delivery.** The protocol provides the mail originator with an irrevocable proof that the mail content received by the recipient was the same as the one sent by the originator. This proof-of-delivery can protect against any attempt by the recipient to falsely deny receiving the message.

- **Fairness.** Proper execution of the protocol ensures that the proof-of-delivery from the mail recipient and the proof-of-origin from the mail originator are available to the mail originator and recipient, respectively. Moreover, the protocol must be fail-safe. That is, incomplete execution of the protocol will not result in a situation where the proof-of-delivery is available to the originator but the proof-of-origin is not available to the recipient, or vice versa.

## 4 Description of CMP1 using IOA Model

In this section we provide a detailed description of CMP1 protocol by using the IOA model. We use an automaton \text{sender}_i to model the sender part on node \( i \) and an automaton \text{receiver}_i to model the receiver part on node \( i \). Hence, each node \( i \in \mathcal{P} \) is modeled with the composition of automata: \text{sender}_i and \text{receiver}_i. The \text{ttp} is modeled with a single automaton and, for each \( i, j \in \mathcal{P} \cup \{\text{ttp}\} \) there is an automaton which models the channel between the node \( i \) and the node \( j \).

We assume that the channel from the \text{ttp} to any node \( i \in \mathcal{P} \) is reliable, i.e., we assume that these channels do not lose or alter in transit messages. Therefore, we distinguish two type of channels, a reliable one: \text{channel}_{\text{ttp},i}, and an unreliable one: \text{UNREL CHANNEL}_{i,j}, for any \( i \in \mathcal{P} \) and \( j \in \mathcal{P} \cup \{\text{ttp}\} \). The overall system is described by the composition of all the above automata. Figure 3 gives an overview of the automata that compose the system.
4.1 IOA Code for the Sender

The code of sender$_i$ is shown in Figure 4. For each session, the sender keeps the following information: the StatusSnd is the “program counter” that goes through the steps of the normal protocol; variables $M1$ and $M4$ are used to store the corresponding messages of the protocol.

![Fig. 3. Overview of the system modeled as IOA.](image)

![Fig. 4. Automaton sender$_i$.](image)

We can now start with the description of the automaton actions, and will proceed by looking at each of them in the order they appear in the code from top to bottom, left column first. This order corresponds to the logical order in which the actions are executed. Notice the use of the unique identifier $id$: it is attached to all the messages concerning a particular email: this is just to avoid interference with possible delayed messages from other sessions.

We assume that the environment tells the automaton when to send an email $m$ to a recipient $j$; this is modeled by the input action Deliver$(m,j)_i$. A new session $id$ is created for this email by means of the function Getuniqid and this id is used to identify all the communication related to this request. The...
first step in the processing of a request for an email $m$ is simply to construct the first message of the protocol $M_1 = \langle S, R, ttp, h(m), PK_{tp}(k), E_k(Sig_S(S, R, ttp, m)) \rangle$ where $k$ is a session key, by using the function $Constr_{M_1}$. Variable $Status_{Snd}$ is set to send so that the only (non-input) action that is enabled is the Send action. This action interacts with the channel to the recipient $j$ and sends the message stored in $M1$. The program counter goes into a wait state wait. All the non input actions are not enabled now. The execution proceeds when a message is received from the ttp. When this message is received, it is stored into variable $M4$. The program counter is updated to done. At this point the protocol has terminated successfully and nothing else has to be done. The output action Send is in a task, so in a fair execution it has infinitely many opportunities to be performed.

4.2 IOA Code for the Receiver

The code of $receiver_i$ is shown in Figure 5. As for the sender, state variables are indexed by a session id. Again, the state variable $Status_{Rcv}$ is the “program counter”. Variables $M1$, $M2$ and $M3$ are used to store the corresponding messages of the protocol.

RECEIVER$_i$

Let $S = \{idle, received, wait, discarded, done\}$

State:
- for each $id \in \mathbb{N}$
  - $Status_{Rcv}(id) \in S$, initially idle
  - $M1(id) \in \mathcal{M}$, initially nil
  - $M2(id) \in \mathcal{M}$, initially nil
  - $M3(id) \in \mathcal{M}$, initially nil

Actions:

input Receive($m, id$)$_{j,i}$

Eff: $M1(id) := m$

Status$_{Rcv}(id) :=$ received

output Send($M2(id), id$)$_{i,ttp}$

Pre: $Status_{Rcv}(id) = received$

$M2(id) := Constr_{M2}(M1(id), id)$

Eff: $Status_{Rcv}(id) := wait$

Tasks: \{Send($m, id$)$_{i,ttp}$, Discard(id)$_i$\}

output Discard(id)$_i$

Pre: $Status_{Rcv}(id) = received$

Eff: $Status_{Rcv}(id) :=$ discarded

input Receive($m, id$)$_{ttp,i}$

Eff: if $(Status_{Rcv}(id) = wait)$

$M3(id) := m$

$Status_{Rcv}(id) :=$ done

Fig. 5. Automaton RECEIVER$_i$
Variable $M1$ is used to store the message itself. The program counter $StatusRcv$ is set to received so that the enabled actions are Send and Discard. The automaton non-deterministically executes one of these actions. If it executes the Discard action the program counter $StatusRcv$ is set to discarded and nothing else has to be done. Otherwise, using the function $Constr\_M2$ the message $M2 = \langle \text{Sig}_t (S, R, ttp, h(m)), PK_{ttp}(k), E_k (\text{Sig}_s (S, R, ttp, m)) \rangle$ is constructed and it is sent to $ttp$. The automaton goes into a waiting state (no internal or output action is enabled) by setting $StatusRcv$ to wait. The automaton exits from this waiting state upon reception of a message from $ttp$. When this message is received, it is stored into variable $M3$. The program counter is updated to done. At this point the protocol has terminated successfully. The done state for this session, means that the receiver has the original email. The Send and Discard actions are in the same task, hence, in a fair execution this task gets infinitely many opportunities to perform one of these actions.

4.3 IOA Code for the Trusted Third Party

TTP
Let $S = \{idle, received, send_rcv, send_snd, corrupt, done\}$

State:
for each $id \in N$

- $StatusTtp(id) \in S$, initially idle
- $M2(id) \in M$, initially nil
- $M3(id) \in M$, initially nil
- $M4(id) \in M$, initially nil
- $Rcv(id) \in P$, initially nil
- $Snd(id) \in P$, initially nil
- $Hcheck(id) \in B$, initially no
- $HttpToRcv(id) \in B$, initially no

Actions:

input Receive$(m, id)i,ttp$
Eff: if($StatusTtp(id) = idle$)
  $M2(id) := m$
  $Rcv(id) := \text{ExtractRcv}(m)$
  $Snd(id) := \text{ExtractSnd}(m)$
  $StatusTtp(id) := received$

internal Check$(M2(id), id)_{ttp}$
Pre: $StatusTtp(id) = received$
Eff: if($\text{CheckSignHash}(M2(id), id)$)
  $StatusTtp(id) := send_rcv$
  $M3(id) := Constr\_M3(M2(id), id)$
  $M4(id) := Constr\_M4(M2(id), id)$
  $Hcheck(id) := yes$
else $StatusTtp(id) := corrupt$

output Send$(M3(id))_{ttp,Rcv(id)}$
Pre: $StatusTtp(id) = send_rcv$
Eff: $StatusTtp(id) := send_snd$
  $HttpToRcv(id) := yes$

output Send$(M4(id))_{ttp,Snd(id)}$
Pre: $StatusTtp(id) = send_snd$
Eff: $StatusTtp(id) := done$

Tasks: $\{\text{Check}(M2(id), id)_{ttp}\}, \{\text{Send}(M3(id))_{ttp,Rcv(id)}\}, \{\text{Send}(M4(id))_{ttp,Snd(id)}\}$

Fig. 6. Automaton TTP
The code of the ttp is shown in Figure 6. For each session, the ttp keeps the following information: the StatusTtp is the “program counter”; variables Snd and Rcv store the sender and the receiver for the session; variables $M_2$, $M_3$ and $M_4$ are used to store the corresponding messages of the protocol. By using the CheckSignHash($m$, id) function the ttp first checks the validity of $\text{Sig}_R(S, R, \text{ttp}, h(m))$ using public key of $R$, then it decrypts $PK_{\text{ttp}}(k)$ using its private key, and decrypts $E_k(\text{Sig}_S(S, R, \text{ttp}, m))$ using $k$. Next, the ttp checks the validity of $\text{Sig}_S(S, R, \text{ttp}, h(m))$ using $S$’s public key, computes $h(m)$, and compares this $h(m)$ with the one received in $\text{Sig}_R(S, R, \text{ttp}, h(m))$. If the two values match, the ttp knows that $m$ is the mail content that $S$ wanted to send to $R$ and that $R$ is willing to receive $m$. In this case the function CheckSignHash($m$, id) returns true and the ttp is able to construct the proof-of-origin and the proof-of-delivery by using the functions Constr$_{M_3}$ and Constr$_{M_4}$, respectively. Moreover, we also use two history variables $^2$ $\text{Hcheck}$ and $\text{HTtpToRcv}$. The variable $\text{Hcheck}$ is set to yes if the ttp is able to construct the message $M_3$ and $M_4$ corresponding to the proof-of-origin and to the proof-of-delivery, respectively, whereas, the value of the history variable $\text{HTtpToRcv}$ is yes if the ttp has sent the message $M_3$ to the receiver. We are now ready to describe the actions of automaton ttp top to bottom, left to right. The Receive($m$, id)$_{\text{ttpp}}$ action takes a message from the channel and stores it into variable $M_2$. The program counter StatusTtp is set to received so that the enabled action is the internal action Check($M_2(id)$)$_{\text{ttpp}}$. The Check($M_2(id)$)$_{\text{ttpp}}$ action checks whether it may construct the proofs of delivery and origin with the CheckSignHash($m$, id) function. If this is not possible, the program counter StatusTtp is set to corrupt and the protocol is aborted. Otherwise, the ttp constructs the messages $M_3 = \langle \text{Sig}_{\text{ttp}}(\text{Sig}_R(S, R, \text{ttp}, h(m))), R, m \rangle$ and $M_4 = \langle \text{Sig}_{\text{ttp}}(\text{Sig}_S(S, R, \text{ttp}, m)) \rangle$ and the program counter StatusTtp is set to send_rcv so that the enabled action is the Send to the receiver. Finally, the Send($M_3(id)$)$_{\text{ttpp}, \text{Rcv}(id)}$ action sets the program counter StatusTtp to send_snd and the Send action to the sender can be executed. The message $M_4$ is sent to the sender so that the program counter StatusTtp is update to done. The done state for this session means that this session has completed and nothing else has to be done. Actions Check($M_2(id)$)$_{\text{ttpp}}$, Send($M_3(id)$)$_{\text{ttpp}, \text{Rcv}(id)}$ and Send($M_4(id)$)$_{\text{ttpp}, \text{Snd}(id)}$ are in three different tasks, hence, in a fair execution they get infinitely many opportunities to be executed.

$^2$ An history variable is a variable that is used only for the proofs but it is not necessary in the real code.
4.4 IOA Code for Channels

The code for UNREL\textsc{Channel}_\{i,j\} is shown in Figure 7. The state is described by variable \texttt{Msgs} that contains the messages still in transit on the channel. It has an \texttt{input} action \texttt{Send}(m, id)_{i,j} whose effect is to add a message in the set of in transit messages. Non-deterministically the automaton can execute one of the two actions in the task: \{\texttt{Receive}(m, id)_{i,j}, \texttt{Lose}(m, id)_{i,j}\}. The \texttt{Receive}(m, id)_{i,j} action models the delivery of the message, whereas the \texttt{Lose}(m, id)_{i,j} action models the loss or the alteration of a message in transit on the channel.

\begin{verbatim}
UNREL\textsc{CHANNEL}_{i,j}
State: \texttt{Msgs}, a set of elements of \(\mathcal{M}\), initially empty

Actions:
\begin{itemize}
  \item \texttt{input Send}(m, id)_{i,j}
    \hspace{1em} Eff: add \((m, id)\) to \texttt{Msgs}
  \item \texttt{internal Lose}(m, id)_{i,j}
    \hspace{1em} Pre: \((m, id)\) is in \texttt{Msgs}
    \hspace{1em} Eff: remove \((m, id)\) from \texttt{Msgs}
  \item \texttt{output Receive}(m, id)_{i,j}
    \hspace{1em} Pre: \((m, id)\) is in \texttt{Msgs}
    \hspace{1em} Eff: remove \((m, id)\) from \texttt{Msgs}
  \item \texttt{internal Lose}(m, id)_{i,j}
    \hspace{1em} Pre: \((m, id)\) is in \texttt{Msgs}
    \hspace{1em} Eff: remove \((m, id)\) from \texttt{Msgs}
\end{itemize}

Task: \{\texttt{Receive}(m, id)_{i,j}, \texttt{Lose}(m, id)_{i,j}\}
\end{verbatim}

Fig. 7. Automaton UNREL\textsc{CHANNEL}_{i,j},

The code for \textsc{channel}\texttt{ttp,}\texttt{i} is shown in Figure 8. The automaton is described in section 2.2. We have only added the two history variables: \texttt{HChanSnd}(id) and \texttt{HChanRcv}(id). The history variable \texttt{HChanSnd}(id) models the mailing of a message from the ttp to \texttt{i} whereas, the variable \texttt{HChanRcv}(id) models the delivery of the message.

\begin{verbatim}
CHANNEL_{\texttt{ttp,}\texttt{i}}
State:
\begin{itemize}
  \item for each \(id \in \mathcal{N}\)
    \hspace{1em} \texttt{Msgs}, a set of elements of \(\mathcal{M}\), initially empty
    \hspace{1em} \texttt{HChanSnd}(id) \in \mathbb{B}, initially \texttt{no}
    \hspace{1em} \texttt{HChanRcv}(id) \in \mathbb{B}, initially \texttt{no}
\end{itemize}

Actions:
\begin{itemize}
  \item \texttt{input Send}(m, id)_{\texttt{ttp,}\texttt{i}}
    \hspace{1em} Eff: add \((m, id)\) to \texttt{Msgs}
    \hspace{1em} \texttt{HChanSnd}(id)=\texttt{YES}
  \item \texttt{output Receive}(m, id)_{\texttt{ttp,}\texttt{i}}
    \hspace{1em} Pre: \((m, id)\) is in \texttt{Msgs}
    \hspace{1em} Eff: remove \((m, id)\) from \texttt{Msgs}
    \hspace{1em} \texttt{HChanRcv}(id)=\texttt{YES}
\end{itemize}

Tasks: \{\texttt{Receive}(m, id)_{\texttt{ttp,}\texttt{i}}\}
\end{verbatim}

Fig. 8. Automaton CHANNEL_{\texttt{ttp,}\texttt{i}}
5 Correctness of CMP1 Protocol

In this section we analyze the CMP1 protocol by using the IOA model, in particular we prove that the protocol satisfies the properties shown in section 3. In the following we denote by $S$ and $R$ the indices corresponding to the processes which represent the sender and the receiver, respectively.

During the $\text{Check}(m, id)$ action, the TTP executes the function $\text{CheckSignHash}(m, id)$ which returns yes if the TTP is able to construct the proof-of-origin and the proof-of-delivery corresponding to the messages $M_3$ and $M_4$, respectively. Hence, in order to show that the protocol CMP1 satisfies the properties in section 3, we have to prove the following three informal assertions:

- The sender eventually receives the message $M_4$ corresponding to the proof-of-delivery constructed by the TTP.
- The receiver eventually receives the message $M_3$ corresponding to the proof-of-origin constructed by the TTP.
- The sender eventually receives the proof-of-delivery if and only if the receiver eventually receives the proof-of-origin.

In the following we will prove several invariants that will be used to prove the above statements.

5.1 Invariants

The first invariant shows that if $\text{StatusSnd}(id) = \text{done}$ the message $M_4$ of the protocol has been delivered to the sender.

**Invariant 5.1** In any reachable state $s$, if $s.\text{StatusSnd}(id) = \text{done}$ then $s.\text{channel}_{\text{ttp}, S}.\text{HChanRec}(id) = \text{yes}$.

**Proof:** By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. Initially $\text{StatusSnd}(id)$ is $\text{idle}$. Hence, the invariant is true.

For the inductive step, assume that the invariant is true in a reachable state $s'$. We need to prove that it is true in $s$ for any possible step $(s', \pi, s)$. If $s.\text{StatusSnd}(id) \neq \text{done}$, the invariant is true. Thus, assume that $s.\text{StatusSnd}(id) = \text{done}$. We have to distinguish the following two cases:

- $s'.\text{StatusSnd}(id) = \text{done}$. From the inductive hypothesis it holds that $s'.\text{channel}_{\text{ttp}, S}.\text{HChanRec}(id) = \text{yes}$. Since $\text{HChanRec}(id)$ once set to yes, never changes any longer, it holds that $s.\text{channel}_{\text{ttp}, S}.\text{HChanRec}(id) = \text{yes}$. 

• $s'.\text{StatusSnd}(id) \neq \text{done}$. There exists only one enabled action that sets $\text{StatusSnd}(id)$ to $\text{done}$: $\pi$ is the Receive$(m, id)_{\text{ttp}, s}$ action of the $\text{sender}_S$ automaton. This input action is controlled by the output action Receive$(m, id)_{\text{ttp}, s}$ of the $\text{channel}_{\text{ttp}, s}$ automaton.

Since this action sets $\text{channel}_{\text{ttp}, s}$.HChanRec$(id)$ to $\text{yes}$, it follows that $s.\text{channel}_{\text{ttp}, s}$.HChanRec$(id) = \text{yes}$. □

The next invariant states that if $i$ receives a message from the ttp the message has been sent by the ttp.

**Invariant 5.2** In any reachable state $s$, if $s.\text{channel}_{\text{ttp}, i}.\text{HChanRec}_{\text{ttp}, i}(id) = \text{yes}$ then $s.\text{channel}_{\text{ttp}, i}.\text{HChanSnd}(id) = \text{yes}$.

**Proof:** By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. Initially we have that $\text{channel}_{\text{ttp}, i}.\text{HChanRec}_{\text{ttp}, i}(id) = \text{no}$. Hence, the invariant is true.

For the inductive step, assume that the invariant is true in a reachable state $s'$. We need to prove it is true in $s$ for any possible step $(s', \pi, s)$. If we have that $s.\text{channel}_{\text{ttp}, i}.\text{HChanRec}(id) = \text{no}$, then the invariant is true. Thus, assume that $s.\text{channel}_{\text{ttp}, i}.\text{HChanRec}(id) = \text{yes}$. We have to distinguish the following two cases:

- **$s'.\text{channel}_{\text{ttp}, i}.\text{HChanRec}(id) = \text{yes}$**.
  
  From the inductive hypothesis it holds that $s'.\text{channel}_{\text{ttp}, i}.\text{HChanSnd}(id) = \text{yes}$. Since $\text{channel}_{\text{ttp}, i}.\text{HChanSnd}(id)$ once set to $\text{yes}$, never changes any longer, it holds that $s.\text{channel}_{\text{ttp}, i}.\text{HChanSnd}(id) = \text{yes}$.

- **$\text{channel}_{\text{ttp}, i}.\text{HChanRec}(id) = \text{no}$**.
  
  There exists only one enabled action that sets $s.\text{channel}_{\text{ttp}, i}.\text{HChanRec}(id)$ to $\text{yes}$: $\pi$ is the output action Receive$(m, id)_{\text{ttp}, i}$ of the $\text{channel}_{\text{ttp}, i}$ automaton. The precondition of this action states that the message $m$ is in $\text{channel}_{\text{ttp}, i}.\text{Msgs}$. There is only one action that inserts a message in $\text{channel}_{\text{ttp}, i}.\text{Msgs}$: the input action Send$(m, id)_{\text{ttp}, i}$ of the $\text{channel}_{\text{ttp}, i}$ automaton. This action also sets $\text{channel}_{\text{ttp}, i}.\text{HChanSnd}(id)$ to $\text{yes}$.

Since $\text{channel}_{\text{ttp}, i}.\text{HChanSnd}(id)$ once set to $\text{yes}$, never changes any longer, it follows that $s.\text{channel}_{\text{ttp}, i}.\text{HChanSnd}(id) = \text{yes}$. □

Invariant 5.3 states that if the ttp has sent a message to the sender it has completed the protocol.

**Invariant 5.3** In any reachable state $s$, if $s.\text{channel}_{\text{ttp}, s}.\text{HChanSnd}(id) = \text{yes}$ then $s.\text{StatusTtp}(id) = \text{done}$.
Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. Initially we have that \( \text{channel}_{ttp,s}.\text{HChanSnd}(id) = \text{no} \). Hence, the invariant is true.

For the inductive step, assume that the invariant is true in a reachable state \( s' \). We need to prove it is true in \( s' \) for any possible step \((s', \pi, s)\). If \( s.\text{channel}_{ttp,s}.\text{HChanSnd}(id) = \text{no} \), the invariant is true. Thus, assume that \( s.\text{channel}_{ttp,s}.\text{HChanSnd}(id) = \text{yes} \). We have to distinguish the following two cases:

- \( s'.\text{channel}_{ttp,s}.\text{HChanSnd}(id) = \text{yes} \).
  From the inductive hypothesis \( s'.\text{StatusTtp}(id) = \text{done} \).
  Since \( \text{StatusTtp}(id) \) once set to \( \text{done} \) never changes any longer, it holds that \( s.\text{StatusTtp}(id) = \text{done} \).

- \( s'.\text{channel}_{ttp,s}.\text{HChanSnd}(id) = \text{yes} \). There exists only one enabled action that sets \( s.\text{channel}_{ttp,s}.\text{HChanSnd}(id) \) to \text{yes}: \( \pi \) is the the input action \( \text{Send}(m, id)_{ttp,s} \) of the \( \text{channel}_{ttp,s} \) automaton. This input action is controlled by the output action \( \text{Send}(m, id)_{ttp,s} \) of the \( \text{ttp} \) automaton. Since this action sets \( \text{StatusTtp}(id) \) to \text{done}, it follows that \( s.\text{StatusTtp}(id) = \text{done} \).

The next invariant shows that if the receiver has completed the protocol, it received the message \( M_3 \).

Invariant 5.4 In any reachable state \( s \), if \( s.\text{StatusRcv}(id) = \text{done} \) then we have \( s.\text{channel}_{ttp,R}.\text{HChanRec}(id) = \text{yes} \).

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. Initially we have that \( \text{StatusRcv}(id) \) is \text{idle}. Hence, the invariant is true.

For the inductive step, assume that the invariant is true in a reachable state \( s' \). We need to prove it is true in \( s \) for any possible step \((s', \pi, s)\). If it holds that \( s.\text{StatusRcv}(id) \neq \text{done} \), the invariant is true. Thus, assume that \( s.\text{StatusRcv}(id) = \text{done} \). We have to distinguish the following two cases:

- \( s'.\text{StatusRcv}(id) = \text{done} \). From the inductive hypothesis it holds that \( s'.\text{channel}_{ttp,R}.\text{HChanRec}(id) = \text{yes} \).
  Since \( \text{channel}_{ttp,R}.\text{HChanRec}(id) \) once set to \text{yes}, never changes any longer, it holds that \( s.\text{channel}_{ttp,R}.\text{HChanRec}(id) = \text{yes} \).

- \( s'.\text{StatusRcv}(id) \neq \text{done} \). There exists only one enabled action that sets \( \text{StatusRcv}(id) \) to \text{done}: \( \pi \) is the input action \( \text{Receive}(m, id)_{ttp,R} \) of the automaton \( \text{receiver}_R \). This input action is controlled by the output action \( \text{Receive}(m, id)_{ttp,R} \) of the \( \text{channel}_{ttp,R} \).
Since this action sets $\text{channel}_{\text{ttp},R}.\text{HChanRec}(id)$ to yes, it follows that $s.\text{channel}_{\text{ttp},R}.\text{HChanRec}(id) = \text{yes}$. 

The next invariant states that if the message is in transit on the channel from the ttp to the receiver, the message was sent by the ttp.

**Invariant 5.5** *In any reachable state $s$, if $s.\text{channel}_{\text{ttp},R}.\text{HChanSnd}(id) = \text{yes}$ then $s.\text{HTtpToRcv}(id) = \text{yes}$.**

**Proof:** By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. Initially we have that $\text{channel}_{\text{ttp},R}.\text{HChanSnd}(id) = \text{no}$. Hence, the invariant is true.

For the inductive step, assume that the invariant is true in a reachable state $s'$. We need to prove it is true in $s$ for any possible step $(s', \pi, s)$. If we have that $s.\text{channel}_{\text{ttp},R}.\text{HChanSnd}(id) = \text{no}$, then the invariant is true. Thus, assume that $s.\text{channel}_{\text{ttp},R}.\text{HChanSnd}(id) = \text{yes}$. We have to distinguish the following two cases:

- $s'.\text{channel}_{\text{ttp},R}.\text{HChanSnd}(id) = \text{yes}$. From inductive hypothesis it holds that $s'.\text{HTtpToRcv}(id) = \text{yes}$. Since HTtpToRcv(id) once set to yes never changes any longer, it holds that $s.\text{HTtpToRcv}(id) = \text{yes}$.

- $s'.\text{channel}_{\text{ttp},R}.\text{HChanSnd}(id) = \text{no}$. There exists only one enabled action that sets $s.\text{channel}_{\text{ttp},R}.\text{HChanSnd}(id)$ to yes:

  $\pi$ is the input action $\text{Send}(m, id)_{\text{ttp},R}$ of $\text{channel}_{\text{ttp},R}$. This input action is controlled by the output action $\text{Send}(m, id)_{\text{ttp},R}$ of the ttp automaton. Since this action sets HTtpToRcv(id) to yes, it follows that $s.\text{HTtpToRcv}(id) = \text{yes}$. 

Invariant 5.6 shows that if the ttp completed the protocol, then it has sent message $M_3$ to the receiver.

**Invariant 5.6** *In any reachable state $s$, if $s.\text{StatusTtp}(id) = \text{done}$ then we have $s.\text{HTtpToRcv}(id) = \text{yes}$.**

**Proof:** By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. $\text{StatusTtp}(id) = \text{idle}$. Hence, the invariant is true.

For the inductive step, assume that the invariant is true in a reachable state $s'$. We need to prove it is true in $s$ for any possible step $(s', \pi, s)$. If $s.\text{StatusTtp}(id) \neq \text{done}$, the invariant is true.

Thus, assume that $s.\text{StatusTtp}(id) = \text{done}$. We have to distinguish the following two cases:
\begin{itemize}
\item $s'.\text{StatusTtp}(id) = \text{done}$. From the inductive hypothesis, it follows that $s'.\text{HTtpToRcv}(id) = \text{yes}$. Since HTtpToRcv(id) once set to yes never changes any longer, it holds that $s.\text{HTtpToRcv}(id) = \text{yes}$.
\item $s'.\text{StatusTtp}(id) \neq \text{done}$. There exists only one action enabled that sets $s.\text{StatusTtp}(id)$ to done: $\pi$ is the output action $\text{Send}(m(id))_{\text{ttp,Snd}(id)}$ of TTP automaton. The precondition of this action claims: $\text{StatusTtp}(id) = \text{send-snd}$. The only action that sets $\text{StatusTtp}(id)$ to send-snd is the output action $\text{Send}(m(id))_{\text{ttp,Rcv}(id)}$ of TTP automaton. This action also sets HTtpToRcv(id) to yes. Since HTtpToRcv(id) once set to yes never changes any longer, it holds that $s.\text{HTtpToRcv}(id) = \text{yes}$.
\end{itemize}

The next invariant shows that the TTP executes the internal action Check before executing its Send actions.

\textbf{Invariant 5.7} In any reachable state $s$, if $s.\text{StatusTtp}(id) \in \{\text{send-rcv, send-snd, done}\}$ then $s.\text{Hcheck}(id) = \text{yes}$.

\textbf{Proof:} By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. Initially $\text{StatusTtp}(id)$ is idle. Hence, the invariant is true.

For the inductive step, assume that the invariant is true in a reachable state $s'$. We need to prove it is true in $s$ for any possible step ($s', \pi, s$). If $s.\text{StatusTtp}(id) \not\in \{\text{send-rcv, send-snd, done}\}$, the invariant is true. Otherwise, we have to distinguish the following two cases:

\begin{itemize}
\item $s'.\text{StatusTtp}(id) \in \{\text{send-rcv, send-snd, done}\}$. From the inductive hypothesis $s'.\text{Hcheck}(id) = \text{yes}$. Since, Hcheck(id) once set to yes, never changes any longer, it follows that $s.\text{Hcheck}(id) = \text{yes}$.
\item $s'.\text{StatusTtp}(id) \not\in \{\text{send-rcv, send-snd, done}\}$. There exists only one enabled action that sets StatusTtp(id) to a value in $\{\text{send-rcv, send-snd, done}\}$: $\pi$ is the internal action $\text{Check}(m,id)_{\text{ttp}}$ of the TTP automaton. This action also sets Hcheck(id) to yes. Therefore, $s.\text{Hcheck}(id) = \text{yes}$. \hfill \Box
\end{itemize}

Finally, the following invariant states that once the TTP has sent message $M_3$ to the receiver, in order to complete the protocol it only needs to send message $M_4$ to the sender.

\textbf{Invariant 5.8} In any reachable state $s$, if $s.\text{HTtpToRcv}(id) = \text{yes}$ then we have $s.\text{StatusTtp}(id) \in \{\text{send-snd, done}\}$.

\textbf{Proof:} By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. Initially $s.\text{HTtpToRcv}(id)$ is idle. Hence, the invariant is true.

For the inductive step, assume that the invariant is true in a reachable state $s'$. We need to prove it is true in $s$ for any possible step ($s', \pi, s$). If $s.\text{HTtpToRcv}(id) \not\in \{\text{send-rcv, send-snd, done}\}$, the invariant is true. Otherwise, we have to distinguish the following two cases:

\begin{itemize}
\item $s'.\text{HTtpToRcv}(id) \in \{\text{send-rcv, send-snd, done}\}$. From the inductive hypothesis $s'.\text{Hcheck}(id) = \text{yes}$. Since, Hcheck(id) once set to yes, never changes any longer, it follows that $s.\text{Hcheck}(id) = \text{yes}$.
\item $s'.\text{HTtpToRcv}(id) \not\in \{\text{send-rcv, send-snd, done}\}$. There exists only one enabled action that sets HTtpToRcv(id) to a value in $\{\text{send-rcv, send-snd, done}\}$: $\pi$ is the internal action $\text{Check}(m,id)_{\text{ttp}}$ of the TTP automaton. This action also sets Hcheck(id) to yes. Therefore, $s.\text{Hcheck}(id) = \text{yes}$. \hfill \Box
\end{itemize}
= no. Hence, the invariant is true.

For the inductive step, assume that the invariant is true in a reachable state \( s' \). We need to prove it is true in \( s \) for any possible step \((s', \pi, s)\). If \( s.\text{HTTPToRcv}(\text{id}) = \text{no} \), the invariant is true. Thus, assume that \( s.\text{HTTPToRcv}(\text{id}) = \text{yes} \). We have to distinguish the following two cases:

- \( s'.\text{HTTPToRcv}(\text{id}) = \text{yes} \). From the inductive hypothesis, it holds that \( s'.\text{StatusTtp}(\text{id}) \in \{\text{send-snd}, \text{done}\} \). If \( s'.\text{StatusTtp}(\text{id}) = \text{send-snd} \) there exists only one enabled action that modifies the value of \( \text{StatusTtp}(\text{id}) \): \( \pi \) is the output action \( \text{Send}(m, \text{id})_{\text{ttp}, \text{S}} \) of the \texttt{ttp} automaton. This action sets \( \text{StatusTtp}(\text{id}) \) to \text{done}. Moreover, \( \text{StatusTtp}(\text{id}) \) once set to \text{done} never changes any longer. It follows that \( s.\text{StatusTtp}(\text{id}) \in \{\text{send-snd}, \text{done}\} \).

- \( s'.\text{HTTPToRcv}(\text{id}) = \text{no} \). There exists only one enabled action that sets \( s.\text{HTTPToRcv}(\text{id}) \) to \text{yes}: \( \pi \) is the output action \( \text{Send}(m, \text{id})_{\text{ttp}, \text{S}} \) of the \texttt{ttp} automaton. This action also sets \( \text{StatusTtp}(\text{id}) \) to \text{send-snd}. Therefore, \( s.\text{StatusTtp}(\text{id}) \in \{\text{send-snd}, \text{done}\} \).

5.2 Non Repudiation of Destination Property

The variable \( M_4 \) of \texttt{sender}_S automaton contains a message received by the \texttt{ttp}. We will prove that the \texttt{ttp} sends this message after that the controls made by the \texttt{CheckSignHash}(m, \text{id}) function has been executed and the \texttt{ttp} is able to construct the proof of delivery for the sender by using the \texttt{Constr}_M function. Recall that the value of the history variable \( \text{Hcheck}(\text{id}) \) is \text{yes} only if the \texttt{ttp} may send the proof of delivery to the sender. Hence, we have to prove the following lemma:

**Lemma 5.1** In any reachable state \( s \), if \( s.\text{StatusSnd}(\text{id}) = \text{done} \) then we have that \( s.\text{Hcheck}(\text{id}) = \text{yes} \).

**Proof.** If \( s.\text{StatusSnd}(\text{id}) = \text{done} \), from Invariant 5.1 it holds that \( s.\text{HChanRec}_{\text{ttp}, \text{S}} = \text{yes} \). From Invariant 5.2 it holds that \( s.\text{HChanSnd}_{\text{ttp}, \text{S}} = \text{yes} \). Moreover, from Invariant 5.3 it holds that \( s.\text{StatusTtp}(\text{id}) = \text{done} \). Finally, from Invariant 5.7 \( s.\text{Hcheck}(\text{id}) = \text{yes} \).

If the history variable \( \text{Hcheck}(\text{id}) \) is set to \text{yes}, then the \texttt{ttp} is able to send the proof-of-delivery corresponding to the message \( M_4 \) to \( S \). The next lemma says that if the \texttt{ttp} sends the proof-of-delivery to \( S \), eventually \( S \) receives it.

**Lemma 5.2** In any fair execution, if there exists a state \( s' \) for which it holds that \( s'.\text{Hcheck}(\text{id}) = \text{yes} \), then there exists a reachable state \( s \) such that \( s.\text{StatusSnd}(\text{id}) = \text{done} \).
Proof. The variable $Hcheck(id)$, initially is equal to no. There exists only one action that sets it to yes: $Check(m, id)_{ttp}$. Since the execution is fair, the output actions $Send(m, id)_{ttp, r}$ and $Send(m, id)_{ttp, s}$ of the ttp will be executed. Moreover, the $Send(m, id)_{ttp, s}$ action controls the input action of the channel between ttp and the sender $S$. From the fair property also the output action of the channel will be executed. Finally, the $Receive$ action of the channel controls the $Receive$ action of the automaton $SENDER_s$. Since, this action sets $StatusSnd(id)$ to done, it follows that there exists a state $s$ such that $s.StatusSnd(id) = done$. 

The variable $StatusSnd$ is set to done after that the sender received the message $M_4$ from the ttp. Therefore, the next theorem easily follows from Lemma 5.1 and Lemma 5.2.

**Theorem 5.3** The CMP1 protocol satisfies the non repudiation of destination property.

### 5.3 Non Repudiation of Origin Property

The variable $M_3$ of the $RECEIVER_R$ automaton contains a message received by the ttp. We will prove that the ttp sends this message after that the controls made by the $CheckSignHash(m, id)$ function has been executed and the ttp is able to construct the proof of delivery for the sender by using the $Constr_M3$ function. Recall that the value of the history variable $Hcheck(id)$ is yes only if the ttp may send the proof of origin to the sender. Hence, we have to prove the following lemma:

**Lemma 5.4** In any reachable state $s$, if $s.StatusRcv(id) = done$ then we have that $s.Hcheck(id) = yes$.

**Proof:** If $s.StatusRcv(id) = done$, from Invariant 5.4 it holds that $s.HChanRec_{ttp, r} = yes$. From Invariant 5.2 it holds that $s.HChanSnd_{ttp, r} = yes$. Moreover, from Invariant 5.5 and Invariant 5.8 it holds that $s.StatusTtp(id) \in \{send-snd, done\}$. Finally, from Invariant 5.7 $s.Hcheck(id) = yes$. 

If the history variable $Hcheck(id)$ is set to yes, then the ttp is able to send the proof-of-origin corresponding to the message $M_3$ to $R$. The next lemma says that if the ttp sends the proof-of-origin to $R$, eventually $R$ receives it.

**Lemma 5.5** In any fair execution, if there exists a state $s'$ for which it holds that $s'.Hcheck(id) = yes$, then there exists a reachable state $s$ such that $s.StatusRcv(id) = done$. 

---

*Proof.* The variable $Hcheck(id)$, initially is equal to no. There exists only one action that sets it to yes: $Check(m, id)_{ttp}$. Since the execution is fair, the output actions $Send(m, id)_{ttp, r}$ and $Send(m, id)_{ttp, s}$ of the ttp will be executed. Moreover, the $Send(m, id)_{ttp, s}$ action controls the input action of the channel between ttp and the sender $S$. From the fair property also the output action of the channel will be executed. Finally, the $Receive$ action of the channel controls the $Receive$ action of the automaton $SENDER_s$. Since, this action sets $StatusSnd(id)$ to done, it follows that there exists a state $s$ such that $s.StatusSnd(id) = done$. 

The variable $StatusSnd$ is set to done after that the sender received the message $M_4$ from the ttp. Therefore, the next theorem easily follows from Lemma 5.1 and Lemma 5.2.

**Theorem 5.3** The CMP1 protocol satisfies the non repudiation of destination property.

### 5.3 Non Repudiation of Origin Property

The variable $M_3$ of the $RECEIVER_R$ automaton contains a message received by the ttp. We will prove that the ttp sends this message after that the controls made by the $CheckSignHash(m, id)$ function has been executed and the ttp is able to construct the proof of delivery for the sender by using the $Constr_M3$ function. Recall that the value of the history variable $Hcheck(id)$ is yes only if the ttp may send the proof of origin to the sender. Hence, we have to prove the following lemma:

**Lemma 5.4** In any reachable state $s$, if $s.StatusRcv(id) = done$ then we have that $s.Hcheck(id) = yes$.

**Proof:** If $s.StatusRcv(id) = done$, from Invariant 5.4 it holds that $s.HChanRec_{ttp, r} = yes$. From Invariant 5.2 it holds that $s.HChanSnd_{ttp, r} = yes$. Moreover, from Invariant 5.5 and Invariant 5.8 it holds that $s.StatusTtp(id) \in \{send-snd, done\}$. Finally, from Invariant 5.7 $s.Hcheck(id) = yes$. 

If the history variable $Hcheck(id)$ is set to yes, then the ttp is able to send the proof-of-origin corresponding to the message $M_3$ to $R$. The next lemma says that if the ttp sends the proof-of-origin to $R$, eventually $R$ receives it.

**Lemma 5.5** In any fair execution, if there exists a state $s'$ for which it holds that $s'.Hcheck(id) = yes$, then there exists a reachable state $s$ such that $s.StatusRcv(id) = done$. 

---

*Proof.*
Proof. The variable $Hcheck(id)$, initially is equal to $no$. There exists only one action that sets it to $yes$: $Check(m, id)_{ttp}$. Since the execution is fair, the output actions $Send(m, id)_{ttp,R}$ of the $ttp$ will be executed. Moreover, this action controls the input action of the channel between $ttp$ and the sender $S$. From the fair property also the output action of the channel will be executed. Finally, the $Receive$ action of the channel controls the $Receive$ action of the automaton $RECEIVER_R$. Since, this action sets $StatusRcv(id)$ to $done$, it follows that there exists a state $s$ such that $s.StatusSnd(id) = done$. 

The variable $StatusRcv$ is set to $done$ after that the receiver received the message $M_3$ from the $ttp$. Therefore, the next theorem easily follows from Lemma 5.4 and Lemma 5.5.

Theorem 5.6 The CMP1 protocol satisfies the non repudiation of origin property.

5.4 Fairness Property

The next lemma states that the sender receives the prof-of-delivery if and only if the receiver receives the proof-of-origin.

Lemma 5.7 In any fair execution, there exists a state $s$ for which it holds that $s.StatusRcv(id) = done$ if and only if there exists a state $s'$ such that $s'.StatusSnd(id) = done$.

Proof: Assume that there exists a state $s$ such that $s.StatusRcv(id) = done$. From Invariants 5.4, 5.2 and 5.5 it holds that $s.HttpToRcv(id) = yes$. This variable, initially is equal to $no$ and there exists only one action that sets it to $yes$: $Send(m, id)_{ttp,R}$. Since the execution is fair, the output action $Send(m, id)_{ttp,S}$ of the $ttp$ will be executed. Moreover, this action controls the input action of the channel between $ttp$ and the sender $S$. From the fair property also the output action of the channel will be executed. Finally, the $Receive$ action of the channel controls the $Receive$ action of the automaton $SENDER_S$. Since, this action sets $StatusSnd(id)$ to $done$, it follows that there exists a state $s'$ such that $s'.StatusSnd(id) = done$.

Conversely, assume that there exists a state $s'$ such that $s'.StatusSnd(id) = done$. From Invariant 5.1, Invariant 5.2 and Invariant 5.3, it holds that $s'.StatusTtp(id) = done$. From Invariant 5.6 it holds that $s'.HttpToRcv(id) = yes$. This variable, initially equal to $no$, is set to $yes$ in the $Send$ action of the $ttp$ to the receiver $R$. This action controls the input action of the channel between $ttp$ and the receiver $R$. Since the execution is fair also the output action of the channel will be executed. Finally, the $Receive$ action of the channel controls
the Receive action of the automaton RECEIVER<sub>R</sub>. Since this action sets Status<sub>Rcv</sub>(id) to done, there exists a state s such that s.Status<sub>Rcv</sub>(id) = done.

The next theorem easily follows from Lemma 5.7.

**Theorem 5.8** The CMP1 protocol satisfies the fairness property.

### 6 Conclusions

Describing and reasoning about asynchronous distributed systems is often a difficult and error prone task. The I/O Automaton [7] provides a framework allowing both a precise description of the code and the possibility of very detailed proofs. In this paper we carry out a simple experiment in using the IOA as a tool to describe and to reason about cryptographic protocols running in an asynchronous distributed system. We showed the feasibility of the approach by examining the security properties of the Deng’s certified email protocol and proving its correctness. We are planning to extend these ideas to the modeling of more complex protocols for certified email [4].

### References


