Title	[課題研究報告書] 証明スコアを用いた認証プロトコルの形式的検証の検討と新規事例調査
Author(s)	藤井, 柊歩
Citation	
Issue Date	2021-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/17121
Rights	
Description	Supervisor: 緒方 和博, 先端科学技術研究科, 修士(情報科学)

Master's Research Project Report

An investigation of formal verification of authentication protocols with proof score and a new case study

Shuho FUJII

Supervisor Kazuhiro OGATA

Graduate School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology
(Information Science)

Month 3, Year 2021

Abstract

With the rapid spread and development of the Internet, security protocols that guarantee safe and secure communication on the Internet are becoming more and more popular. Although these security protocols have been carefully designed by security experts, it was not uncommon for security attacks such as interception, tampering and impersonation to happen, leading to lots of serious damages. Ensuring the reliability of security protocols is thus absolutely important. Many approaches have been proposed against the unexpected flaws in these protocols. In formal method, some techniques for formally verifying the correctness of security protocols have been extensively studied.

This research focuses on formal verification of the correctness of authentication protocols and We survey case studies conducted in the past as well as to conduct new case studies. Authentication is the process of verifying the identity of a person, an object, a computer, a program, etc. It is an indispensable technology for preventing unauthorized operations in network systems (also known as access control). Protocols are communication conventions that are necessary to communicate with each other. Thus, an authentication protocol is a communication convention to achieve authentication. Computers, printers, and programs are used and participated in by an unspecified number of entities, and only encoded information is exchanged. Therefore, there is a high possibility of eavesdropping, falsification, and impersonation of communication. There- fore, authentication protocols are intended to realize authentication for secure communication in such insecure communication channels.

This research focuses on two case studies of authentication protocols are presented with the Identify-Friend-or-Foe-System protocol (IFF protocol or just IFF) and the Needham-Schroeder-Lowe Public-Key protocol (NSLPK protocol or just NSLPK). NSLPK can be regarded as an advanced authentication protocol of IFF. We study the specification of two protocols in CafeOBJ, which is a formal specification language, and understand the "proof scores" to prove that they enjoy some desired properties. We present two more ways of verification that IFF enjoys some properties by using CafeInMaude Proof Assistant (CiMPA), and CafeInMaude Proof Generator (CiMPG). By achieving the objectives of this research, we will be able to acquire techniques to mitigate the number of authentication protocol failures, which can contribute to safer and more secure shopping on e-commerce sites and safer and more secure communication on the Internet.

Keywords : CafeOBJ, CiMPG, CiMPA, proof score, algebraic specification language, authentication protocol

Contents

List of Figures

Chapter 1

Introduction

In Chapter 1, the background and purpose of this research project and the structure of the subsequent chapters are presented.

1.1 Overview

With the rapid spread and development of the Internet, security protocols that guarantee safe and secure communication on the Internet are becoming more and more popular. Although these security protocols have been carefully designed by security experts, it was not uncommon for security attacks such as interception, tampering and impersonation to happen, leading to lots of serious damages. Ensuring the reliability of security protocols is thus absolutely important. Many approaches have been proposed against the unexpected flaws in these protocols. In formal method, some techniques for formally verifying the correctness of security protocols have been extensively studied.

1.2 Aims

This research focuses on formal verification of the correctness of authentication protocols and We survey case studies conducted in the past as well as to conduct new case studies. This research focuses on two case studies of authentication protocols are presented with the Identify-Friend-or-Foe-System protocol (IFF protocol or just IFF) and the Needham-Schroeder-Lowe Public-Key protocol (NSLPK protocol or just NSLPK). NSLPK can be regarded as an advanced authentication protocol of IFF. We study the specification of two protocols in CafeOBJ, which is a formal specification language, and understand the ”proof scores” to prove that they enjoy some desired properties. We present two more ways of verification that IFF enjoys some properties by using CafeInMaude Proof Assistant (CiMPA), and CafeInMaude Proof Generator (CiMPG). By achieving the objectives of this research, we will be able to acquire techniques to mitigate the number of authentication protocol failures, which can contribute to safer and more secure shopping on e-commerce sites and safer and more secure communication on the Internet.

1.3 The structure of the subsequent chapters

The remainder of this report is organized as follows:
- Chapter 2 - Preliminaries gives some common notions and background knowledge which are requirements for the rest of the report.
- Chapter 3 - Formal Verification of IFF Authentication Protocol with Proof Scores presents the formal verification that IFF protocol enjoys some desired properties by writing proof scores.
- Chapter 4 - Formal Verification of NSLPK Authentication Protocol with Proof Scores presents the formal verification that NSLPK protocol enjoys some desired properties by writing proof scores.
- Chapter 5 - Formal Verification of IFF Authentication Protocol with CiMPA and CiMPG presents two more ways of the formal verification with IFF protocol.
- Chapter 6 - Lessons Learned describes what we learned through the research project.
- Chapter 7 - Conclusion summarizes the report and gives some pieces of our future work.

Chapter 2

Preliminaries

This Chapter gives some common notions and background knowledge which are requirements for the rest of the report.

2.1 Authentication protocol

2.1.1 What is authentication

Authentication is the process of verifying the identity of a person, an object, a computer, a program, etc. It is an indispensable technology for preventing unauthorized operations in network systems (also known as access control).

For example, in the authority management of a server in a network system, after authenticating the identifiers of the users registered in the server (who have accessed the server), a list called ACL is used to describe what authority the subject has, and operations other than those listed in the list are not allowed. In general, this kind of technology is used to manage computer resources. In general, this technology is used to achieve secure access control in network systems via the Internet.

2.1.2 What is Authentication protocol

Protocols are communication conventions that are necessary to communicate with each other. Thus, an authentication protocol is a communication convention to achieve authentication. Computers, printers, and programs are used and participated in by an unspecified number of entities, and only encoded information is exchanged. Therefore, there is a high possibility of eavesdropping, falsification, and impersonation of communication. Therefore, authentication protocols are intended to realize authentication for secure communication in such insecure communication channels.

2.1.3 Authentication implementation method

Authentication is the process of verifying the identity of a person, an object, a computer, a program, etc. Mutual authentication is when two parties authenticate each other. There are two main ways to achieve mutual authentication using computers.

Method using shared information

This is a method of verification in which secret information is shared between the verifier and the subject, and the verifier confirms whether or not the subject possesses the secret information. Examples include password authentication, shared key authentication, and biometric authentication.

Method using public key

It is a method of authentication using a public key and possession of the corresponding private key. Examples include challenge-response authentication and authentication using digital signatures.

In both of these two methods, the information for authentication passes through an insecure communication channel at least once, because the authentication is ultimately done using a call from the subject to the verifier. Therefore, the following is necessary for the implementation of these methods.

- **Confidential information should not flow through the communication channel in plain text.**
- **That the call for authentication is different every time.**
- **Easy to change secret information for authentication.**
- **Easy to manage confidential information for authentication.**

A report is an encrypted version of an identifier, key, nonce (random number, etc., a report generated for each session), etc.

It is also necessary that the entity that is authenticating and the entity that is using the authentication must match. Take postal service as an example. In the postal service, when subject A sends a letter to subject B, the flow is as shown in Figure 2.1. Subject A takes the mail to the nearest post office (1), the mail is transferred between post offices (2), and subject B receives the mail from the nearest post office (3). In this case, authentication between post offices in (2) is only a secure communication in the transfer between post offices, and does not guarantee the safety of the user. Even if the authentication between post offices is done, if there is another person who pretends to be Subject B, the mail that should have been sent to Subject B may end up in the hands of another person. To ensure that the mail sent by Subject A passes through the secure communication channel and reaches Subject B, it is necessary to authenticate and encrypt the communication between Subject A and B.

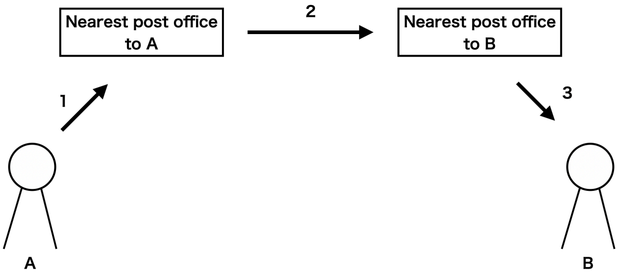


Figure 2.1: Example of a post office

2.1.4 Threats in Communications

The threats that may be posed on communication paths without authentication and encryption between communicating entities are as follows
1.Interception:Unauthorized entities (attackers) intercepting calls.
2.Falsification:An attacker takes a call, falsifies it, and sends it as if it were a legitimate call.
3.Hoax:Sending a report generated by an attacker to cause a malicious effect.
4.Masquerade:The attacker pretends to be a different entity.
5.Replay:An attacker uses the report, or a portion of it, to send it to an unauthorized effect.

In order to achieve communication that eliminates these threats, it is necessary to satisfy confidentiality (the property that only an appropriately authorized entity can read the report in communication without an unintended third party being able to decipher it) and integrity (the property that the report in communication is genuine and has not been tampered with, or can be detected if it has been tampered with) between two mutually determined communication entities. In other words, it is enough to satisfy the following requirements In other words, it is enough to satisfy the following requirements. In other words, it is necessary to prepare a tunnel-like communication channel between two determined parties, as shown in Fig. 2.2, which is not subject to any observation by others, and to use that channel for all communication between the two parties, and the authentication protocol must ensure such a communication channel.

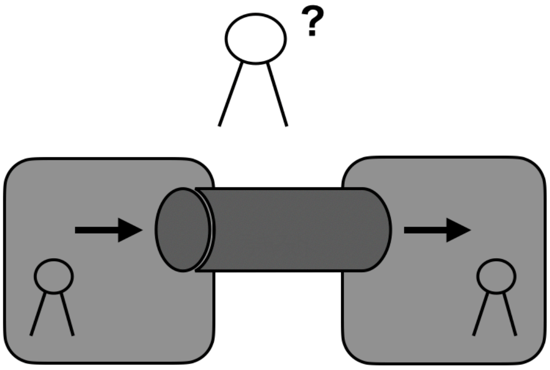


Figure 2.2: Secure communication channel

2.2 CafeOBJ

2.2.1 What is CafeOBJ

CafeOBJ is a language for verification (formal specification) and validation of formal models, designed to support formal methods. CafeOBJ is a formal specification language classified as an algebraic specification language, and can be executed by interpreting the equations that make up the specification as rewrite rules[4][5][6]. The flow of verification is as follows.
1.Understanding the problem and modeling it
2.Creating a CafeOBJ specification and formulating the properties to be verified
3.Verification by CafeOBJ system

Each of 1 3 will be explained using a mutual exclusion protocol called Qlock.

2.2.2 What is Qlock

When there is a resource shared by multiple processes, it is sometimes required that the resource be exclusively specified in the sense that at any given time there is at most one process using the resource. Thus, a mutual exclusion protocol is a mechanism for exclusive use of a shared resource, and a mutual exclusion protocol realized by using an atomic queue is called a Qlock. An atomic queue is a queue in which elements can be added, deleted, etc. as indivisible operations.

2.2.3 Understanding the problem and modeling it

Each process in Qlock behaves as follows. Each process i is in the other region when it does not use the shared resource, and in the sensitive region when it does. When each process i wants to specify a shared resource, it adds the process identifier i to the end of the queue (put(queue,i)), waits until i comes to the top of the queue (top(queue) = i), and then enters the intervening region. When it finishes using the shared resource, it removes the top of the queue (get(queue)) and returns to the rest of the region. Each process i repeats this.

rm, wt, and cs are labels. When a process is in the other region, we say it is in label rm. When it's waiting to enter a sensitive area, say it's on label wt. When it is in a close region, we say it is in label cs. Initially, we assume that all processes are in label rm and that queue is empty.

One of the properties of Qlock to be satisfied is mutual exclusivity, that is, there is always at most one process in the intervening region, and this is the property to be verified.

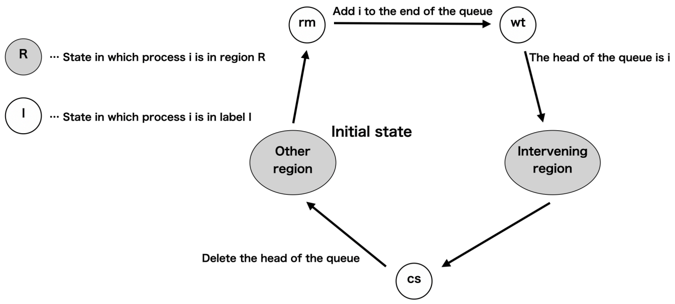


Figure 2.3: Behavior of each process

Modeling of states in observations

The behavior of Qlock is modeled by specifying the change in observable values. The behavior of Qlock is modeled as an observation transition system $\mathcal{S}_{Qlock}$, where the state transition system represents the state by a collection of observable values.

Since the observables that characterize the state of Qlock are the value of queue and the position of each process, we prepare observation functions queue and pc that take the state of $\mathcal{S}_{Qlock}$ as arguments and return these values to represent their observables. That is, given the state s of $\mathcal{S}_{Qlock}$ and the process identifier i, queue(s) and pc(s,i) represent the value of queue and the position of process i in the snapshot of the execution of Qlock represented by the state s, respectively. The set of observation functions of $\mathcal{S}_{Qlock}$ is defined as follows $\mathcal{O}_{Qlock}$

$\mathcal{O}_{Qlock}$\{queue : Sys → Queue,
pc : Sys Pid → Label\}

Sys, Queue, Pid, and Label represent the state, process identifier queue, process identifier, and label type, respectively. $\mathcal{O}_{Qlock}$ is described as follows in cafeOBJ.

op pc : Sys Pid -> Label
op queue : Sys -> Queue

op stands for operator, which declares an operation that takes the state of the system as an argument; in the case of pc and queue, the return value is a data type, so we declare an observation function.

Modeling of state transitions

The behavior of Qlock is represented as state transitions, and Qlock has the following three execution units.

(1)Execution of put(queue,i)
(2)Execution of top(queue) = i
(3)Execution of get(queue)

These are represented by the transition functions want, try, and exit, respectively. Given a state s and a process identifier i in $\mathcal{S}_{Qlock}$, want(s,i), try(s,i), and exit(s,i) represent the state after process i executes put(queue,i) in state s, the state after repeating top(queue) = i once, and the state after executing get(queue), respectively. and the state after executing get(queue). The set $\mathcal{T}_{Qlock}$ of transition functions in $\mathcal{S}_{Qlock}$ is represented as follows.

$\mathcal{T}_{Qlock}$\{want : Sys Pid → Sys,
try : Sys Pid → Sys,
exit : Sys Pid → Sys\}

$\mathcal{T}_{Qlock}$ is written as follows in CafeOBJ.

op want : Sys Pid -> Sys {constr}
op try : Sys Pid -> Sys {constr}
op exit : Sys Pid -> Sys {constr}

{constr} indicates that want, try, and exit are Sys terms.

Modeling the initial state

In the initial state init of $\mathcal{S}_{Qlock}$, queue(init) returns the empty queue, and for any process identifier i, pc(init,i) returns the label rm. The initial state can be modeled as a set of states $\mathcal{L}_{Qlock}$ that satisfy this condition as follows.

$\mathcal{L}_{Qlock}\{\text{init} \parallel queue(init) = empty \wedge$
$pc(init, i) = rm\}$

The initial state and the conditions it must satisfy are described in CafeOBJ as follows

```
op init : -> Sys {constr}
eq pc(init,I) = rs .
eq queue(init) = empty .
```

The two equations declared in eq declare the conditions that init must satisfy. what I, rs, and empty are is defined elsewhere.

2.2.4 Creating a CafeOBJ specification and formulating the properties to be verified

In describing the observation transition system $\mathcal{S}_{Qlock}$ in CafeOBJ, we first define the data types LABEL, PID, and QUEUE in CafeOBJ. the description unit of CafeOBJ is a module, and the CafeOBJ specification is expressed in modules.

Embedded Modules: BOOL

Some of the basic data types are provided as built-in modules. One of them is the module BOOL. It declares a visible sort Bool that returns a Boolean value, two arguments true and false that represent truth and falsity, and basic operations on Boolean values, and defines their meanings in equations. The module BOOL is automatically imported into the user-defined module, since it is the basis of logical computation for inference and verification.

Specifications of Label : LABEL

```
mod! LABEL {
[Label]
ops rs ws cs : -> Label {constr}
eq (rs = ws) = false .
eq (rs = cs) = false .
eq (ws = cs) = false .
}
```

The three constants rm, ws, and cs, declared together in ops(operators), correspond to the labels rm, ws, and cs, respectively.

process identifier : PID

```
mod* PID {
[ErrPid Pid < PidErr]
op none : -> ErrPid
var I : Pid
var EI : ErrPid
eq (I = EI) = false .
}
```

CafeOBJ can order inclusion relations between sorts, and can handle partial functions and error handling. pidErr is the upper sort for ErrPid and Pid. none is a constant for ErrPid, and is provided as different from the process identifier.

Specifications ofQueueing : QUEUE

```
mod! QUEUE(E :: TRIVerr) {
[EQueue NeQueue < Queue]
op empty : -> EQueue {constr}
op __ : Elt.E Queue -> NeQueue {constr}
op enq : Queue Elt.E -> NeQueue
op deq : Queue -> Queue
op top : EQueue -> ErrElt.E
op top : NeQueue -> Elt.E
op top : Queue -> EltErr.E
op _\in_ : Elt Queue -> Bool
op del : Queue Elt.E -> Queue
var Q : Queue
vars X Y : Elt.E
eq enq(empty,X) = X empty .
eq enq(Y Q,X) = Y enq(Q,X) .
eq deq(empty) = empty .
eq deq(X Q) = Q .
eq top(empty) = err.E .
eq top(X Q) = X .
eq X \in empty = false .
eq X \textbackslash in (Y Q) = (if X = Y then true else X \in Q fi) .
eq del(empty,Y) = empty .
eq del(X Q,Y) = (if X = Y then Q else X del(Q,Y) fi) .
eq X \in enq(Q,Y) = (if X = Y then true else X \in Q fi) .
ceq X \in del(enq(Q,X),X) = false if not X \in Q .
ceq X \in del(enq(Q,Y),X) = X \in del(Q,X) if not X = Y .
ceq X \in del(Q,X) = false if not X \in Q .
}
```

E is a temporary argument of the parameterization module QUEUE, whose requirements are specified by the temporary argument module TRIVerr. The module TRIVerr is declared as follows: var Q: Queue declares that the identifier Q is to be used as a variable of the sort Queue with equality to be declared in this module.

```
mod* TRIVerr {
[ErrElt Elt < EltErr]
op err : -> ErrElt
```

```
}
```

The module TRIVerr specifies the existence of a set, indicated by the sort Elt, and one element that does not belong to that set, indicated by the constant none.

The constant empty in module QUEUE indicates an empty queue, and the operation __ indicates that the non-empty queue construct {constr} is a construct attribute. Elt.E refers to the visible sort Elt of the temporary argument E. The operations enq, dep, top, and del represent the usual functions of a queue, and their definitions are given by the equations.

Embodiment of the parameter module

The real arguments used when embodying a parameterization module must meet the requirements specified in the temporary argument module TRIVerr. In other words, the model of the real argument must be the model of the temporary argument. The realization of the parameterization module is done by mapping the elements of real arguments to the elements of temporary arguments. The language element of CafePBJ that defines this mapping is the view. The view for embodying a provisional argument E, whose requirements are specified in the module TRIVerr, with the process identifier PID is as follows.

```
view TRIVerr2PID from TRIVerr to PID
sort Elt -> Pid,
sort ErrElt -> ErrPid,
sort EltErr -> PidErr,
op err -> none,
```

CafeOBJ Specification of Observation Transition System

Now that we have created the CafeOBJ specifications for the required data types, we can use them to write the CafeOBJ specification for $\mathcal{S}_{Qlock}$ as follows

```
mod* QLOCK {
pr(LABEL + PID)
pr(QUEUE(E <= TRIVerr2PID))
[Sys]
op init : -> Sys {constr}
op want : Sys Pid -> Sys {constr}
op try : Sys Pid -> Sys {constr}
op exit : Sys Pid -> Sys {constr}
op pc : Sys Pid -> Label
op queue : Sys -> Queue
var S : Sys
vars I J : Pid
var Q : Queue
```

```
eq pc(init,I) = rs .
eq queue(init) = empty .
op c-want : Sys Pid -> Bool
eq c-want(S,I) = (pc(S,I) = rs) .
ceq pc(want(S,I),J) = (if I = J then ws else pc(S,J) fi) if c-want(S,I) .
ceq queue(want(S,I)) = enq(queue(S),I) if c-want(S,I) .
ceq want(S,I) = S if not c-want(S,I) .
op c-try : Sys Pid -> Bool
eq c-try(S,I) = (pc(S,I) = ws and top(queue(S)) = I) .
ceq pc(try(S,I),J) = (if I = J then cs else pc(S,J) fi) if c-try(S,I) .
eq queue(try(S,I)) = queue(S) .
ceq try(S,I) = S if not c-try(S,I) .
op c-exit : Sys Pid -> Bool
eq c-exit(S,I) = (pc(S,I) = cs) .
ceq pc(exit(S,I),J) = (if I = J then rs else pc(S,J) fi) if c-exit(S,I) .
ceq queue(exit(S,I)) = deq(queue(S)) if c-exit(S,I) .
ceq exit(S,I) = S if not c-exit(S,I) .
op inv1 : Sys Pid Pid -> Bool
op inv2 : Sys Pid -> Bool
op inv3 : Sys Pid -> Bool
op inv4 : Sys Pid -> Bool
op inv5 : Sys Pid -> Bool
op inv6 : Sys Pid -> Bool
op inv7 : Sys Pid -> Bool
eq inv1(S,I,J) = ((pc(S,I) = cs and pc(S,J) = cs) implies I = J) .
eq inv2(S,I) = (pc(S,I) = cs implies top(queue(S)) = I) .
eq inv3(S,I) = (pc(S,I) = rs implies (not I \in queue(S))) .
eq inv4(S,I) = ((not I \in queue(S)) implies pc(S,I) = rs) .
eq inv5(S,I) = (pc(S,I) = ws or pc(S,I) = cs implies I \in queue(S)) .
eq inv6(S,I) = (I \in queue(S) implies pc(S,I) = ws or pc(S,I) = cs) .
eq inv7(S,I) = (not I \in del(queue(S),I)) .
}
```

pr stands for protecting, which declares the module to be imported in protected mode. Module PLOCK explicitly imports LABEL, PID, and QUEUE (E <= TRIVerr2PID) and implicitly imports one module BOOL, where Sys is a hidden sort and represents the state space of $\mathcal{S}_{Qlock}$. . The constant init represents an arbitrary initial state of $\mathcal{S}_{Qlock}$. The operations pc and queue correspond to pc and queue, respectively, and are called observation functions. The operations try, want, and exit correspond to try, want, and exit, respectively, and are called transition functions. After these declarations, we define the initial state and the transition functions. inv1 inv7 formulate the properties that Qlock must have in order to satisfy mutual exclusivity.

2.2.5 Verification by CafeOBJ system

In order to verify that Qlock satisfies mutual exclusivity, we modeled Qlock as an observation transition machine $\mathcal{S}_{Qlock}$ and created its CafeOBJ specification QLOCK. Finally, we formulate the proof methods and try to realize them in the CafeOBJ system[7][8][9]. The proof score for inv1 is expressed as follows.

```
– I) Base case
open QLOCK .
– fresh constants
ops i j : -> Pid .
– ‖–
red inv1(init,i,j) .
close
– II) Induction cases
– 1) want(s,k)
open QLOCK .
– fresh constants
op s : -> Sys .
ops i j k : -> Pid .
– IH
eq [:nonexec] : inv1(s,I:Pid,J:Pid) = true .
– assumptions
eq pc(s,k) = rs .
eq i = k .
– ‖–
red inv1(s,i,j) implies inv1(want(s,k),i,j) .
close
open QLOCK .
– fresh constants
op s : -> Sys .
ops i j k : -> Pid .
– IH
eq [:nonexec] : inv1(s,I:Pid,J:Pid) = true .
– assumptions
eq pc(s,k) = rs .
eq (i = k) = false .
eq j = k .
```

```
    – ∥–
    red inv1(s,i,j) implies inv1(want(s,k),i,j) .
    close
    open QLOCK .
    – fresh constants
    op s : -> Sys .
    ops i j k : -> Pid .
    – IH
    eq [:nonexec] : inv1(s,I:Pid,J:Pid) = true .
    – assumptions
    eq pc(s,k) = rs .
    eq (i = k) = false .
    eq (j = k) = false .
    –
—-
    red inv1(s,i,j) implies inv1(want(s,k),i,j) .
    close
    open QLOCK .
    – fresh constants
    op s : -> Sys .
    ops i j k : -> Pid .
    – IH
    eq [:nonexec] : inv1(s,I:Pid,J:Pid) = true .
    – assumptions
    eq (pc(s,k) = rs) = false .
    – ∥–
    red inv1(s,i,j) implies inv1(want(s,k),i,j) .
    close
    – 2) try(s,k)
    open QLOCK .
    – fresh constants
    op s : -> Sys .
    ops i j k : -> Pid .
    – IH
    eq [:nonexec] : inv1(s,I:Pid,J:Pid) = true .
    – assumptions
    eq pc(s,k) = ws .
    eq top(queue(s)) = k .
    eq i = k .
    eq j = k .
    – ∥–
```

```
red inv1(s,i,j) implies inv1(try(s,k),i,j) .
close
open QLOCK .
– fresh constants
op s : -> Sys .
ops i j k : -> Pid .
– IH
eq [:nonexec] : inv1(s,I:Pid,J:Pid) = true .
eq [:nonexec] : inv2(s,I:Pid) = true .
– assumptions
eq pc(s,k) = ws .
eq top(queue(s)) = k .
eq i = k .
eq (j = k) = false .
eq pc(s,j) = cs .
– ‖–
red inv2(s,j) implies inv1(s,i,j) implies inv1(try(s,k),i,j) .
close
open QLOCK .
– fresh constants
op s : -> Sys .
ops i j k : -> Pid
. – IH
eq [:nonexec] : inv1(s,I:Pid,J:Pid) = true .
eq [:nonexec] : inv2(s,I:Pid) = true .
– assumptions
eq pc(s,k) = ws .
eq top(queue(s)) = k .
eq i = k .
eq (j = k) = false .
eq (pc(s,j) = cs) = false .
– ‖–
red inv1(s,i,j) implies inv1(try(s,k),i,j) .
close
open QLOCK .
– fresh constants
op s : -> Sys .
ops i j k : -> Pid .
– IH
eq [:nonexec] : inv1(s,I:Pid,J:Pid) = true .
eq [:nonexec] : inv2(s,I:Pid) = true .
```

```
– assumptions
eq pc(s,k) = ws .
eq top(queue(s)) = k .
eq (i = k) = false .
eq j = k .
eq pc(s,i) = cs .
– ‖–
red inv2(s,i) implies inv1(s,i,j) implies inv1(try(s,k),i,j) .
close
···
```

The CafeOBJ command open makes the module given as its argument available; the CafeOBJ command red returns the value of the given term converted to the simplest possible term, using all the equations defined in the module as a left-to-right rewrite rule. The CafeOBJ command close is a command to discard and terminate the temporary module created in this way. through the end of the line is a comment.

Proof scores can be broadly divided into an inductive basis and an inductive stage. The inductive stage is further divided corresponding to the transition functions want, try, and exit, each of which is divided into sufficient statements to be proved by running the proof clauses through the CafeOBJ system.

Each proof clause in the inductive stage consists of the following four parts

(1)Declaring a variable that represents an arbitrary value
(2)Equation declarations for assumptions
(3)Definition of post-event condition
(4)Check that the logical formula to be proved is valid under the assumption (2).

In addition, if it is necessary to divide the case within each transition function, the case is divided and further subdivided. If the return values of all reds are true after these subdivisions, the proof is successful, but this does not mean that all reds in inv1 are true. If we assume that inv2 is true, then the return value of red will be true in all cases, which means that the proof is successful.

If we can prove all the cases from inv1 to inv7 as described above, the safeOBJ system has completed the verification that Qlock satisfies mutual exclusivity.

2.3 CiMPG and CiMPA

2.3.1 What is CiMPG and CiMPA

The method of verification using proof scores in CafeOBJ may lead to incorrect proofs because of the addition of unnecessary equations or the wrong way of doing case spliting. CiMPA (CafeInMaude Proof Assistant) is a proof assistant for inductive properties of CafeOBJ specification. CiMPG (CafeInMaude Proof Generator) is a proof assistant that identifies proof scores and provides a minimal set of annotations to generate proof scripts for these proof scores[10][11][12][13]. The advantages of using these are twofold
(1)If a proof script is successfully generated from a proof score using CiMPG, its properties shall be preserved.
(2)If no proof scripts are generated, valuable feedback on the proofs underlying the proof scores can be obtained.

Both proof scores and proof scripts can be written by hand by humans, but writing proof scripts is often more difficult than writing proof scores. Therefore, rather than writing the proof script by hand, it is better to use CiMPG to generate the proof script, and if the proof does not complete correctly, to modify the proof score.

2.3.2 Proof of Qlock using CiMPG and CiMPA

Using QCiMPG and CiMPA, we will generate the proof script for Qlock as explained earlier, and perform the proof. In the proof, we first rewrite inv1 inv7, which formulate the properties that Qlock must have to satisfy mutual exclusivity, as follows.

```
op inv1 : Sys Pid Pid -> Bool
op inv2 : Sys Pid -> Bool
eq inv1(S:Sys,I:Pid,J:Pid) = (((pc(S,I) = cs) and pc(S,J) = cs) implies I = J) .
eq inv2(S:Sys,I:Pid) = (pc(S,I) = cs implies top(queue(S)) = I) .
op inv2-0 : Sys Pid Pid Pid -> Bool
eq inv2-0(S:Sys,I:Pid,J:Pid,K:Pid) = not((pc(S,K) = ws) and (top(queue(S)) = K) and (I = K) and (not (J = K)) and (pc(S,J) = cs)) .
```

Since we have rewritten the properties that Qlock must have as described above, we also need to rewrite the proof score as follows.

```
– I) Base case
open QLOCK .
:id(qlock)
ops i j : -> Pid .
```

```
red inv1(init,i,j) .
close
– II) Induction cases
– 1) want(s,k)
open QLOCK .
:id(qlock)
op s : -> Sys .
ops i j k : -> Pid .
eq [:nonexec] : inv1(s,I:Pid,J:Pid) = true .
eq pc(s,k) = rs .
eq i = k .
red inv1(s,i,j) implies inv1(want(s,k),i,j) .
close
open QLOCK .
:id(qlock)
op s : -> Sys .
ops i j k : -> Pid .
eq [:nonexec] : inv1(s,I:Pid,J:Pid) = true .
eq pc(s,k) = rs .
eq (i = k) = false .
eq j = k .
red inv1(s,i,j) implies inv1(want(s,k),i,j) .
close
open QLOCK .
:id(qlock)
op s : -> Sys .
ops i j k : -> Pid .
eq [:nonexec] : inv1(s,I:Pid,J:Pid) = true .
eq pc(s,k) = rs .
eq (i = k) = false .
eq (j = k) = false .
red inv1(s,i,j) implies inv1(want(s,k),i,j) .
close
open QLOCK .
:id(qlock)
op s : -> Sys .
ops i j k : -¿ Pid .
eq [:nonexec] : inv1(s,I:Pid,J:Pid) = true .
eq (pc(s,k) = rs) = false .
red inv1(s,i,j) implies inv1(want(s,k),i,j) .
close
```

```
...
– I) Base case
open QLOCK .
:id(qlock)
– fresh constants
op i : -> Pid .
– —-
red inv2(init,i) .
close
–
– II) Induction cases
– 1) want(s,k)
open QLOCK .
:id(qlock)
op s : -> Sys .
ops i k : -> Pid .
eq [:nonexec] : inv1(s,I:Pid,J:Pid) = true .
eq [:nonexec] : inv2(s,I:Pid) = true .
eq pc(s,k) = rs .
eq i = k .
red inv2(s,i) implies inv2(want(s,k),i) .
close
open QLOCK .
:id(qlock)
op s : -> Sys .
ops i k : -> Pid .
eq [:nonexec] : inv1(s,I:Pid,J:Pid) = true .
eq [:nonexec] : inv2(s,I:Pid) = true .
eq pc(s,k) = rs .
eq (i = k) = false .
eq queue(s) = empty .
red inv2(s,i) implies inv2(want(s,k),i) .
close
open QLOCK .
:id(qlock)
op s : -> Sys .
ops i k j : -> Pid .
op q : -> Queue .
eq [:nonexec] : inv1(s,I:Pid,J:Pid) = true .
eq [:nonexec] : inv2(s,I:Pid) = true .
eq pc(s,k) = rs .
```

```
eq (i = k) = false .
eq queue(s) = j — q .
red inv2(s,i) implies inv2(want(s,k),i) .
close
open QLOCK .
:id(qlock)
op s : -> Sys .
ops i k : -> Pid .
eq [:nonexec] : inv1(s,I:Pid,J:Pid) = true .
eq [:nonexec] : inv2(s,I:Pid) = true .
eq (pc(s,k) = rs) = false .
red inv2(s,i) implies inv2(want(s,k),i) .
close
···
open QLOCK .
:proof(qlock)
close
```

By rewriting it in this way, it becomes a statement that CiMPG can read, and by writing :proof(qlock) at the end, it generates a proof script. If you give CiMPG the above rewritten specification and proof score, it will return a proof script as shown below.

```
open QLOCK .
:goal{
eq [qlock :nonexec] : inv2(S:Sys,P:Pid) = true .
eq [qlock1 :nonexec] : inv1(S:Sys,P:Pid,P0:Pid) = true .
}
:ind on (S:Sys)
:apply(si)
:apply(tc)
:def csb1 = :ctf {eq pc(SSys,PPid) = cs .}
:apply(csb1)
:def csb2 = :ctf {eq P@Pid = PPid .}
:apply(csb2)
:imp [qlock1] by {P0:Pid <- P0@Pid ; P:Pid <- P@Pid ;}
:apply (rd)
:def csb3 = :ctf {eq P0@Pid = PPid .}
:apply(csb3)
:imp [qlock1] by {P0:Pid <- P0@Pid ; P:Pid <- P@Pid ;}
:apply (rd)
:imp [qlock1] by {P0:Pid <- P0@Pid ; P:Pid <- P@Pid ;}
:apply (rd)
```

```
:imp [qlock1] by {P0:Pid <- P0@Pid ; P:Pid <- P@Pid ;}
:apply (rd)
:def csb4 = :ctf {eq pc(SSys,PPid) = cs .}
:apply(csb4)
:def csb5 = :ctf {eq P@Pid = PPid .}
:apply(csb5)
:imp [qlock] by {P:Pid ¡- P@Pid ;}
:apply (rd)
:def csb6 = :ctf {eq pc(SSys,P@Pid) = cs .}
:apply(csb6)
:imp [qlock] by {P:Pid <- P@Pid ;}
:imp [qlock1] by {P0:Pid <- PPid ; P:Pid <- P@Pid ;}
:apply (rd)
:imp [qlock] by {P:Pid <- P@Pid ;}
:apply (rd)
:imp [qlock] by {P:Pid <- P@Pid ;}
:apply (rd)
:apply(tc)
:apply (rd)
:apply (rd)
:apply(tc)
:def csb7 = :ctf {eq pc(SSys,PPid) = ws .}
:apply(csb7)
:def csb8 = :ctf {eq top(queue(SSys)) = PPid .}
:apply(csb8)
:def csb9 = :ctf {eq P@Pid = PPid .}
:apply(csb9)
:def csb10 = :ctf {eq P0@Pid = PPid .}
:apply(csb10)
:imp [qlock1] by {P0:Pid <- P0@Pid ; P:Pid <- P@Pid ;}
:apply (rd)
:def csb11 = :ctf {eq pc(SSys,P0@Pid) = cs .}
:apply(csb11)
:imp [qlock1] by {P0:Pid <- P0@Pid ; P:Pid <- P@Pid ;}
:imp [qlock] by {P:Pid <- P0@Pid ;}
:apply (rd)
:imp [qlock1] by {P0:Pid <- P0@Pid ; P:Pid <- P@Pid ;}
:apply (rd)
:def csb12 = :ctf {eq P0@Pid = PPid .}
:apply(csb12)
:def csb13 = :ctf {eq pc(SSys,P@Pid) = cs .}
```

```
:apply(csb13)
:imp [qlock1] by {P0:Pid <- P0@Pid ; P:Pid <- P@Pid ;}
:imp [qlock] by {P:Pid <- P@Pid ;}
:apply (rd)
:imp [qlock1] by {P0:Pid <- P0@Pid ; P:Pid <- P@Pid ;}
:apply (rd)
:imp [qlock1] by {P0:Pid <- P0@Pid ; P:Pid <- P@Pid ;}
:apply (rd)
:imp [qlock1] by {P0:Pid <- P0@Pid ; P:Pid <- P@Pid ;}
:apply (rd)
:imp [qlock1] by {P0:Pid <- P0@Pid ; P:Pid <- P@Pid ;}
:apply (rd)
:def csb14 = :ctf {eq pc(SSys,PPid) = ws .}
:apply(csb14)
:def csb15 = :ctf {eq top(queue(SSys)) = PPid .}
:apply(csb15)
:def csb16 = :ctf {eq P@Pid = PPid .}
:apply(csb16)
:imp [qlock] by {P:Pid <- P@Pid ;}
:apply (rd)
:imp [qlock] by {P:Pid <- P@Pid ;}
:apply (rd)
:imp [qlock] by {P:Pid <- P@Pid ;}
:apply (rd)
:imp [qlock] by {P:Pid <- P@Pid ;}
:apply (rd)
:apply(tc)
:def csb17 = :ctf {eq pc(SSys,PPid) = rs .}
:apply(csb17)
:def csb18 = :ctf {eq P@Pid = PPid .}
:apply(csb18)
:imp [qlock1] by {P0:Pid <- P0@Pid ; P:Pid ¡- P@Pid ;}
:apply (rd)
:def csb19 = :ctf {eq P0@Pid = PPid .}
:apply(csb19)
:imp [qlock1] by {P0:Pid <- P0@Pid ; P:Pid <- P@Pid ;}
:apply (rd)
:imp [qlock1] by {P0:Pid <- P0@Pid ; P:Pid <- P@Pid ;}
:apply (rd)
:imp [qlock1] by {P0:Pid <- P0@Pid ; P:Pid <- P@Pid ;}
:apply (rd)
```

```
:def csb20 = :ctf {eq pc(SSys,PPid) = rs .}
:apply(csb20)
:def csb21 = :ctf {eq P@Pid = PPid .}
:apply(csb21)
:imp [qlock] by {P:Pid <- P@Pid ;}
:apply (rd)
:def csb22 = :ctf [queue(SSys) .]
:apply(csb22)
:imp [qlock] by {P:Pid <- P@Pid ;}
:apply (rd)
:imp [qlock] by {P:Pid <- P@Pid ;}
:apply (rd)
:imp [qlock] by {P:Pid <- P@Pid ;}
:apply (rd)
close
```

This is the proof script for Qlock generated by CiMPG, and when it is loaded into CiMPA, it returns ture for all terms, which means the proof is complete.

Chapter 3

Formal Verification of IFF Authentication Protocol with Proof Scores

This Chapter presents the formal verification that IFF protocol enjoys some desired properties by writing proof scores.

3.1 IFF authentication protocol

Use the Identify-Friend-or-Foe System (IFF), which is simpler than NSLPK, to illustrate that.

- CafeOBJ
- Observation transition system
- Proof score
- Consideration of intruders when creating and verifying formal specifications of authentication protocols.

An authentication protocol is a mechanism for entities to authenticate each other over a network[14]. IFF authentication protocol can be described as follows.

Msg1 P → Q : R
Msg2 Q → P : E_K(R,Q)

It is assumed that the members of the group share the private key and never leak it to a third party. R is a random number, and E_K (R, Q) is a ciphertext in which the random number R and its own ID are encrypted with a private key.

3.2 Assumption of the existence of an intruder

Assume the existence of an intruder pretending to be a member. The intruder does not know the private key shared by the group, but can obtain all the messages flowing through the network such as random numbers and ciphertexts.

3.3 Basic behavior of the protocol

f the subject P wants to authenticate the subject Q, which is a member of the group, P randomly generates a number R and send it to Q. Q which receives the random number R sends a ciphertext in which the received random number and its own ID are encrypted with the private key to P. The intruder cannot be visually identified, and all messages flowing through the network such as random numbers and ciphertexts can be acquired. Therefore, if the key used in the ciphertext received by P at this time is the private key shared by the members of the group, Q is a true member of the group and P can authenticate Q.

3.4 Creating a model of IFF

Since IFF assumes the existence of an intruder, the model creation also assumes the existence of an intruder. Create the observation transition system $\mathcal{S}_{iff}$ as a model of IFF.

$\mathcal{S}_{iff}$ consists of the set $\mathcal{O}_{iff}$ of observation functions, the set $\mathcal{L}_{iff}$ of the initial state, and the set $\mathcal{T}_{iff}$ of transition functions. Each is as follows.

$\mathcal{O}_{iff}\{nw : Field \rightarrow Network,$
$ur : Field \rightarrow URands\}$
$\mathrm{L}_{iff}\{init \mid nw(init) = void \wedge$
$ur(init) = empty\}$
$\mathrm{T}_{iff}\{sdcm : Field\ Agent\ Agent\ Rand \rightarrow Field,$
$sdrm : Field\ Agent\ Msg \rightarrow Field,$
$fkcm1 : Field\ Agent\ Agent\ Rand \rightarrow Field,$
$fkrm1 : Field\ Agent\ Agent\ Cipher \rightarrow Field,$
$fkrm2 : Field\ Agent\ Agent\ Rand \rightarrow Field\}$

Field, Network, URands, Agent, Rand, Msg and Cipher are state of IFF, type of network, type of random number multiset, type of subject, type of random number, type of message and type of ciphertext.

The network is also modeled as a multiset of messages. Furthermore, due to the presence of intruders, messages once sent will remain on the network.This is because a message once sent can be sent many times by an intruder. Void represents an empty multiset for networks, and empty represents an empty multiset for random numbers.

3.5 Observation function and transition function

Given state F, the observation functions nw, ur return the random numbers available in that state (ur (F)) and the multiset of messages sent up to that state (nw (F)).

The transition functions sdcm and sdrm correspond to the subject sending Msg1 and Msg2 according to the protocol, respectively. On the other hand, the transition functions fkcm1, fkrm1, and fkrm2 correspond to spoofing Msg1 and Msg2 using random numbers and ciphertexts collected by the intruder, respectively.

3.6 Create CafeOBJ specification for IFF

We will create a CafeOBJ specification of $\mathcal{S}_{ifff}$. Before that, create the CafeOBJ specifications for Network, URands, Agent, Rand, Msg, and Cipher. iff messages are ciphertext, so the network will be a set of multiple ciphertext.

3.6.1 Creating a CafeOBJ specification for a data type

Specifications of agent : AGENT

```
mod* AGENT {
  [Agent]
  op enemy : -> Agent
  eq (P:Agent = P) = true .
}
```

The constant enemy represents a generic intruder.

Specifications of key : KEY

```
mod! KEY {
  pr(AGENT)
  [Key]
  op k : Agent -> Key
  op p : Key -> Agent
  var P : Agent
  vars K1 K2 : Key
```

```
    eq p(k(P)) = P .
    ceq (K1 = K2) = true if not(p(K1) = enemy) and not(p(K2) = enemy) .
    ceq (K1 = K2) = false if not(p(K1) = enemy) and p(K2) = enemy .
}
```

The operation k is a function that returns the key of a given subject based on that subject. Similarly, the operation p is a function that returns the subject holding the key based on the given key.

Specifications of random number : RAND

```
mod* RAND {
    [Rand]
    op _=_ : Rand Rand -> Bool {comm}
}
```

The operation _=_ is a predicate that determines whether the terms representing two random numbers are equal.

Specifications of cipher text : CIPHER

```
mod! CIPHER principal-sort Cipher {
    pr(AGENT + KEY + RAND)
    [Cipher]
    op enc : Key Rand Agent -> Cipher
    op k : Cipher -> Key
    op r : Cipher -> Rand
    op p : Cipher -> Agent
    var K : Key
    var R : Rand
    var P : Agent
    vars C1 C2 : Cipher
    eq k(enc(K,R,P)) = K .
    eq r(enc(K,R,P)) = R .
    eq p(enc(K,R,P)) = P .
    eq (C1 = C2) = (k(C1) = k(C2) and r(C1) = r(C2) and p(C1)
        = p(C2)) .
}
```

The operation enc is a component of the ciphertext. Given the key K, the random number R, and the subject P, it represents the ciphertext E_K (R, P). The operations k, r, and p return the first, second, and third arguments of the operation enc, respectively.

Specifications of message : MSG

```
mod! MSG principal-sort Msg {
    pr(AGENT + RAND + CIPHER)
    [Msg]
    op cm : Agent Agent Agent Rand -> Msg
```

```
    op rm : Agent Agent Agent Cipher -> Msg
    op cm? : Msg -> Bool
    op rm? : Msg -> Bool
    op crt : Msg -> Agent
    op src : Msg -> Agent
    op dst : Msg -> Agent
    op r : Msg -> Rand
    op c : Msg -> Cipher
    vars P1 P2 P3 : Agent
    var R : Rand
    var C : Cipher
    vars M1 M2 : Msg
    eq cm?(cm(P1,P2,P3,R)) = true .
    eq cm?(rm(P1,P2,P3,C)) = false .
    eq rm?(cm(P1,P2,P3,R)) = false .
    eq rm?(rm(P1,P2,P3,C)) = true .
    eq crt(cm(P1,P2,P3,R)) = P1 .
    eq crt(rm(P1,P2,P3,C)) = P1 .
    eq src(cm(P1,P2,P3,R)) = P2 .
    eq src(rm(P1,P2,P3,C)) = P2 .
    eq dst(cm(P1,P2,P3,R)) = P3 .
    eq dst(rm(P1,P2,P3,C)) = P3 .
    eq r(cm(P1,P2,P3,R)) = R .
    eq c(rm(P1,P2,P3,C)) = C .
    ceq (M1 = M2) = (cm?(M1) and crt(M1) = crt(M2) and src(M1)
        = src(M2) and dst(M1) = dst(M2) and r(M1) = r(M2)) if cm?(M2) .
    ceq (M1 = M2) = (rm?(M1) and crt(M1) = crt(M2) and src(M1)
        = src(M2) and dst(M1) = dst(M2) and c(M1) = c(M2)) if rm?(M2) .
}
```

The operation cm takes three subjects and a random number as arguments and returns a message. Each of the three entities represents the true author, sender, and recipient of the message, and this operation corresponds to Msg1. The operation rm takes three subjects and a ciphertext as arguments and returns a message. Similar to the operation cm, each of the three entities represents the true creator, sender, and recipient of the message, which corresponds to Msg2.

Calculation cm?, rm? is a predicate that determines whether the given message is Msg1 or Msg2, and the operations crt, srt, and dst are predicates that determine the true creator, sender, and recipient from the given message, respectively.

The operations r and s return a random number and a ciphertext from the given message, respectively.

Specifications of general-purpose multiset for message : BAG

```
mod! BAG (D :: TRIV) {
    [Elt.D < Bag]
    op void : -> Bag
    op _,_ : Bag Bag -> Bag { assoc comm id: void }
    op _\in_ : Elt.D Bag -> Bool
    var B : Bag
    vars E1 E2 : Elt.D
    eq E1 \in void = false .
    ceq E1 \in (E2,B) = true if E1 = E2 .
    ceq E1 \in (E2,B) = E1 \in B if not(E1 = E2) .
}
```

Specifications of general-purpose multiset for random number : SET

```
mod! SET (D :: TRIV) {
    [Elt.D < Set]
    op empty : -> Set
    op __ : Set Set -> Set { assoc comm idem id: empty }
    op _\in_ : Elt.D Set -> Bool
    var S : Set
    vars E1 E2 : Elt.D
    eq E1 \in empty = false .
    ceq E1 \in (E2 S) = true if E1 = E2 .
    ceq E1 \in (E2 S) = E1 \in S if not(E1 = E2) .
}
```

Specify formal argument D : COLLECTION

```
mod* COLLECTION(D :: TRIV) {
    [Elt.D < Col]
    op _\in_ : Elt.D Col -> Bool
}
```

Sort ELt.D is declared as a subsort of sort Bag and sort Col. This means that each element of a multiset can be regarded as a multiset having only that element. The constants void and empty represent an empty multiset, and the operations _,_ and __ are constituents of a non-empty multiset. Also, the operations _,_ and __ are declared to satisfy the commutative law (comm) and the associative law (assoc).

The operation _\in_ is a predicate that determines whether a given element is included in a given multiset.

Specifications of general-purpose multiset for cipher : NETWORK

```
mod! NETWORK {
    pr(BAG(MSG)*{sort Bag -> Network})
    pr(COLLECTION(RAND)*{sort Col -> ColRands})
    pr(COLLECTION(CIPHER)*{sort Col -> ColCiphers})
    op rands : Network -> ColRands
    op ciphers : Network -> ColCiphers
    var NW : Network
    var M : Msg
    var R : Rand
    var C : Cipher
    eq R \in rands(void) = false .
    ceq R \in rands(M,NW) = true if cm?(M) and R = r(M) .
    ceq R \in rands(M,NW) = true if rm?(M) and k(enemy) = k(c(M)) and
        R = r(c(M)) .
    ceq R \in rands(M,NW) = R ∈ rands(NW)
        if not(cm?(M) and R = r(M)) and
           not(rm?(M) and k(enemy) = k(c(M)) and R = r(c(M))) .
    eq C \in ciphers(void) = false .
    ceq C \in ciphers(M,NW) = true if rm?(M) and C = c(M) .
    ceq C \in ciphers(M,NW) = C ∈ ciphers(NW)ifnot(rm?(M)and
        C = c(M)) .
}
```

Create a multiset of ciphertext from a general-purpose multiset. The sort name has been changed from BAG to NETWORK and from Col to ColRands and ColCiphers.

The operations rands and ciphers return random numbers and ciphertexts in a given multiset, respectively.

3.6.2 Creation of CafeOBJ specifications for observation transition system

Since the CafeOBJ specification of the data type used in the observation transition system $\mathcal{S}_{iff}$ has been created, the CafeOBJ specification of $\mathcal{S}_{iff}$ is created next.

Specifications of observation transition system : IFF

```
mod* IFF {
    pr(NETWORK)
    pr(SET(RAND)*{sort Set -> URands})
    [Field]
    op init : -> Field {constr}
```

```
    op nw : Field -> Network
    op ur : Field -> URands
    op sdcm : Field Agent Agent Rand -> Field {constr}
    op sdrm : Field Agent Msg -> Field {constr}
    op fkcm1 : Field Agent Agent Rand -> Field {constr}
    op fkrm1 : Field Agent Agent Cipher -> Field {constr}
    op fkrm2 : Field Agent Agent Rand -> Field {constr}
    var F : Field
    vars P1 P2 : Agent
    vars M1 M2 : Msg
    var R : Rand
    var C : Cipher
    ···
}
```

The constant init represents any initial state of $\mathcal{S}_{iff}$. The operations nw and ur correspond to the observation functions of $\mathcal{S}_{iff}$, and the remaining operations correspond to the transition functions. In the place of ..., the equations that defines the initial state and behavior of $\mathcal{S}_{iff}$ declared. They will be described below.

Definition of initial state

```
    eq nw(init) = void .
    eq ur(init) = empty .
```

These equations correspond to $\mathcal{L}_{iff}$.

Definition of transition function sdcm

```
    eq c-sdcm(F,P1,P2,R) = not(R \in ur(F)) .
    ceq nw(sdcm(F,P1,P2,R)) = cm(P1,P1,P2,R) , nw(F) if
        c-sdcm(F,P1,P2,R) .
    ceq ur(sdcm(F,P1,P2,R)) = R ur(F) if c-sdcm(F,P1,P2,R) .
    ceq sdcm(F,P1,P2,R) = F if not c-sdcm(F,P1,P2,R) .
```

Each transition function has an effect condition, and the effect condition of this transition function is that R is not included in the multiset of random numbers. When there is a message cm(P1, P1, P2, R) generated by this transition function, the message is added to the network multiset nw(F) and the random number is added to the random number multiset ur(F).

Definition of transition function sdrm

```
    eq c-sdrm(F,P1,M1) = (M1 \in nw(F) and cm?(M1) and P1 = dst(M1))
    ceq nw(sdrm(F,P1,M1)) = rm(P1,P1,src(M1),enc(k(P1),r(M1),P1)) ,
        nw(F) if c-sdrm(F,P1,M1) .
    eq ur(sdrm(F,P1,M1)) = ur(F) .
    ceq sdrm(F,P1,M1) = F if not c-sdrm(F,P1,M1) .
```

The validity condition of this transition function is that M1 addressed to the subject P1 exists in the network. When there is a message rm(P1, P1, src(M1), enc(k (P1), r(M1), P1)) generated by this transition function, that message is put into the network multiset nw(F). In addition, a random number is added to the multiset ur(F) of random numbers.

Definition of transition function fkcm1

```
eq c-fkcm1(F,P1,P2,R) = R \in rands(nw(F)) .
ceq nw(fkcm1(F,P1,P2,R)) = cm(enemy,P1,P2,R) , nw(F)
    if c-fkcm1(F,P1,P2,R) .
eq ur(fkcm1(F,P1,P2,R)) = ur(F) .
ceq fkcm1(F,P1,P2,R) = F if not c-fkcm1(F,P1,P2,R) .
```

The validity condition of this transition function is that R is included in the multiset of random numbers. This means that random numbers may have been collected by the intruder. When there is a message cm(enemy, P1, P2, R) generated by this transition function, the message is added to the network multiset nw(F) and the random number is added to the random number multiset ur(F).

Definition of transition function fkrm1

```
eq c-fkrm1(F,P1,P2,C) = C \in ciphers(nw(F)) .
ceq nw(fkrm1(F,P1,P2,C)) = rm(enemy,P1,P2,C) , nw(F)
    if c-fkrm1(F,P1,P2,C) .
eq ur(fkrm1(F,P1,P2,C)) = ur(F) .
ceq fkrm1(F,P1,P2,C) = F if not c-fkrm1(F,P1,P2,C) .
```

The validity condition of this transition function is that C(ciphertext) is included in the multiset of the network. This means that the ciphertext may have been collected by an intruder. When there is a message rm(enemy, P1, P2, C) generated by this transition function, the message is added to the network multiset nw(F) and the random number is added to the random number multiset ur(F).

Definition of transition function fkrm2

```
eq c-fkrm2(F,P1,P2,R) = R \in rands(nw(F)) .
ceq nw(fkrm2(F,P1,P2,R)) = rm(enemy,P1,P2,enc(k(enemy),R,P1)) , nw
  (F) if c-fkrm2(F,P1,P2,R) .
eq ur(fkrm2(F,P1,P2,R)) = ur(F) .
ceq fkrm2(F,P1,P2,R) = F if not c-fkrm2(F,P1,P2,R) .
```

The validity condition of this transition function is that R is included in the multiset of random numbers. When there is a message rm(enemy, P1, P2, enc(k(enemy), R, P1)) generated by this transition function, that message is added to the multiset nw(F) of the network. Add random numbers to the multiset ur(F) of random numbers.

3.7 Verification of IFF

IFF assumes the existence of an intruder that cannot be visually identified. We verify that the IFF authentication protocol modeled as described above can identify such intruders. The proof by CafeOBJ is performed below.

3.7.1 Verification of intruder identification

First, declare the following modules.

```
mod* INV {
    pr(IFF)
    ops p1 p2 p3 : -> Agent
    op k : -> Key
    op r : -> Rand
    op inv1 : Field Agent Agent Agent Key Rand -> Bool
    op inv2 : Field Key Rand -> Bool
    var F : Field
    vars P1 P2 P3 : Agent
    var K : Key
    var R : Rand
    eq inv1(F,P1,P2,P3,K,R) = ((not(K = k(enemy)) and rm(P1,P2,P3,enc(K
      ,R,P2))\in nw(F)) implies not(P2 = enemy)) .
    eq inv2(F,K,R) = (enc(K,R,enemy) ∈ ciphers(nw(F))implies(K = k(ene
      my))).
}
```

The operation inv1 is the property of the IFF that we want to prove. rm(P1, P2, P3, enc(K, R, P2)) represents the cipher enc(K, R, P2) that appears to be transmitted from P2 to P3. P1 is the true sender. In other words, the property I want to prove is that if the message sent from P2 to P3 exists on the network and the key of that message is the private key of the group, P2 is a companion.

The operation inv2 is a lemma used to prove this property, and ciphers (nw(F)) represent the ciphertext obtained by an intruder. In other words, the meaning of this lemma is that if the ciphertext that third argument of enc is enemy, the key used is the intruder's key.

A module that describes the logical formula to be proved at each induction stage is declared as follows.

```
mod* ISTEP {
    pr(INV)
    ops f f' : -> Field
    op istep1 : Agent Agent Agent Key Rand -> Bool
```

```
    op istep2 : Key Rand -> Bool
    vars P1 P2 P3 : Agent
    var K : Key
    var R : Rand
    eq istep1(P1,P2,P3,K,R) = inv1(f,P1,P2,P3,K,R) implies
        inv1(f’,P1,P2,P3,K,R) .
    eq istep2(K,R) = inv2(f,K,R) implies inv2(f’,K,R) .
}
```

The constant f represents an arbitrary state, and the constant f’ represents the posterior state of the state f.

3.7.2 Proof of inv1

Proof clause of induction basis

```
open INV .
    red inv1(init,p1,p2,p3,k,r) .
close
```

CafeOBJ returns true for this proof clause. Since there were five transition functions this time, this proof clause is divided into five cases, and a proof clause is created for each case where the validity condition is satisfied and when it is not satisfied.

Proof clause about cdcm

When the validity condition is met

```
open ISTEP .
    ops q1 q2 : -> Agent .
    op r1 : -> Rand .
    – eq c-sdcm(f,q1,q2,r1) = true .
    eq r1 \in ur(f) = false .
    –
    eq f’ = sdcm(f,q1,q2,r1) .
    red istep1(p1,p2,p3,k,r) .
close
```

When the validity condition is not met

```
open ISTEP .
    ops q1 q2 : -> Agent .
    op r1 : -> Rand .
    eq c-sdcm(f,q1,q2,r1) = false .
    eq f’ = sdcm(f,q1,q2,r1) .
    red istep1(p1,p2,p3,k,r) .
close
```

We use the assumption that c-sdcm(f, q1, q2, r1) = true and the assumption that c-sdcm(f, q1, q2, r1) = false, respectively. CafeOBJ returns true for this proof clause.

Proof clause about cdrm

When the validity condition is met

```
open ISTEP .
   op q1 : -> Agent .
   op m1 : -> Msg .
   op nw1 : -> Network .
   eq nw(f) = m1 , nw1 .
   eq cm?(m1) = true .
   eq q1 = dst(m1) .
   –
   eq (rm(p1,p2,p3,enc(k,r,p2)) = rm(dst(m1),dst(m1),src(m1),enc(k(dst(m1))
     ,r(m1),dst(m1))))= false .
   eq f' = sdrm(f,q1,m1) .
   red istep1(p1,p2,p3,k,r) .
close
open ISTEP .
   op q1 : -> Agent .
   op m1 : -> Msg .
   op nw1 : -> Network .
   eq nw(f) = m1 , nw1 .
   eq cm?(m1) = true .
   eq q1 = dst(m1) .
   –
       dst(m1))) .
   eq p1 = dst(m1) .
   eq p3 = src(m1) .
   eq k = k(dst(m1)) .
   eq r = r(m1) .
   eq p2 = dst(m1) .
   –
   eq dst(m1) = enemy .
   eq f' = sdrm(f,q1,m1) .
   red istep1(p1,p2,p3,k,r) .
close
open ISTEP .
   op q1 : -> Agent .
   op m1 : -> Msg .
   op nw1 : -> Network .
```

```
    eq nw(f) = m1 , nw1 .
    eq cm?(m1) = true .
    eq q1 = dst(m1) .
    –
        dst(m1))) .
    eq p1 = dst(m1) .
    eq p3 = src(m1) .
    eq k = k(dst(m1)) .
    eq r = r(m1) .
    eq p2 = dst(m1) .
    –
    eq (dst(m1) = enemy) = false .
    eq f' = sdrm(f,q1,m1) .
    red istep1(p1,p2,p3,k,r) .
close
```

When the validity condition is not met

```
open ISTEP .
    op q1 : -> Agent .
    op m1 : -> Msg .
    eq c-sdrm(f,q1,m1) = false .
    eq f' = sdrm(f,q1,m1) .
    red istep1(p1,p2,p3,k,r) .
close
```

When the validity conditions are met, three additional cases are made.

(1) c-sdrm(f, q1, m1) = true and
(rm (p1, p2, p3, enc (k, r, p2)) = rm(dst (m1), dst(m1), src(m1), enc(k(dst(m1)),r(m1)),dst(m1))))))) = false

(2) c-sdrm(f, q1, m1) = true and
rm(p1, p2, p3, enc (k, r, p2)) = rm(dst(m1), dst(m1), src(m1), enc(k(dst(m1)),r(m1), dst(m1))) and dst (m1) = enemy

(3) c-sdrm(f, q1, m1) = true and
rm(p1, p2, p3, enc(k, r, p2)) = rm(dst(m1), dst(m1), src(m1), enc(k(dst(m1)),r(m1), dst(m1))) and Assuming
(dst (m1) = enemy) = false

If the validity condition is not satisfied, the assumption that c-sdrm(f, q1, m1) = false is used. CafeOBJ returns true for this proof clause.

Proof clause about fkcm1

When the validity condition is met

```
open ISTEP .
    ops q1 q2 : -> Agent .
    op r1 : -> Rand .
```

```
    – eq c-fkcm1(f,q1,q2,r1) = true .
    eq r1 \in rands(nw(f)) = true .
    –
    eq f’ = fkcm1(f,q1,q2,r1) .
    red istep1(p1,p2,p3,k,r) .
close
```

When the validity condition is not met

```
open ISTEP .
    ops q1 q2 : -> Agent .
    op r1 : -> Rand .
    eq c-fkcm1(f,q1,q2,r1) = false .
    eq f’ = fkcm1(f,q1,q2,r1) .
    red istep1(p1,p2,p3,k,r) .
close
```

We use the assumption that c-fkcm1(f, q1, q2, r1) = true and the assumption that c-fkcm1(f, q1, q2, r1) = false, respectively. CafeOBJ returns true for this proof clause.

Proof clause about fkrm1

When the validity condition is met

```
open ISTEP .
    ops q1 q2 : -> Agent .
    op c : -> Cipher .
    – eq c-fkrm1(f,q1,q2,c) = true .
    eq c \in ciphers(nw(f)) = true .
    –
    eq (rm(enemy,q1,q2,c) = rm(p1,p2,p3,enc(k,r,p2))) = false .
    eq f’ = fkrm1(f,q1,q2,c) .
    red istep1(p1,p2,p3,k,r) .
close
open ISTEP .
    ops q1 q2 : -> Agent .
    op c : -> Cipher .
    – eq c-fkrm1(f,q1,q2,c) = true .
    – eq c \in ciphers(nw(f)) = true .
    eq enc(k,r,p2) \in ciphers(nw(f)) = true .
    –
    – eq rm(enemy,q1,q2,c) = rm(p1,p2,p3,enc(k,r,p2)) .
    eq p1 = enemy .
    eq q1 = p2 .
    eq q2 = p3 .
    eq c = enc(k,r,p2) .
```

```
    –
    eq k = k(enemy) .
    eq f’ = fkrm1(f,q1,q2,c) .
    red istep1(p1,p2,p3,k,r) .
close
open ISTEP .
    ops q1 q2 : -> Agent .
    op c : -> Cipher .
    – eq c-fkrm1(f,q1,q2,c) = true .
    – eq c \in ciphers(nw(f)) = true .
    eq enc(k,r,p2) \in ciphers(nw(f)) = true .
    –
    – eq rm(enemy,q1,q2,c) = rm(p1,p2,p3,enc(k,r,p2)) .
    eq p1 = enemy .
    eq q1 = p2 .
    eq q2 = p3 .
    eq c = enc(k,r,p2) .
    –
    eq (k = k(enemy)) = false .
    eq (p2 = enemy) = false .
    eq f’ = fkrm1(f,q1,q2,c) .
    red istep1(p1,p2,p3,k,r) .
close
open ISTEP .
    ops q1 q2 : -> Agent .
    op c : -> Cipher .
    – eq c-fkrm1(f,q1,q2,c) = true .
    – eq c \in ciphers(nw(f)) = true .
    – eq enc(k,r,p2) \in ciphers(nw(f)) = true .
    eq enc(k,r,enemy) \in ciphers(nw(f)) = true .
    –
    – eq rm(enemy,q1,q2,c) = rm(p1,p2,p3,enc(k,r,p2)) .
    eq p1 = enemy .
    eq q1 = p2 .
    eq q2 = p3 .
    eq c = enc(k,r,p2) .
    –
    eq (k = k(enemy)) = false .
    eq p2 = enemy .
    eq f’ = fkrm1(f,q1,q2,c) .
    red inv2(f,k,r) implies istep1(p1,p2,p3,k,r) .
```

close

When the validity condition is not met

```
   open ISTEP .
    ops q1 q2 : -> Agent .
    op c : -> Cipher .
    eq c-fkrm1(f,q1,q2,c) = false .
    eq f’ = fkrm1(f,q1,q2,c) .
    red istep1(p1,p2,p3,k,r) .
   close
```

When the validity conditions are met, four additional cases are made.

(1) c-fkrm1(f, q1, q2, c) = true and
  (rm(enemy, q1, q2, c) = rm(p1, p2, p3, enc(k, r, p2)))) = false
(2) c-fkrm1(f, q1, q2, c) = true and
  c \in ciphers(nw(f)) = true and
  rm(enemy, q1, q2, c) = rm(p1, p2, p3, enc(k, r, p2)) and k = k(enemy)
(3) c-fkrm1(f, q1, q2, c) = true and
  c \in ciphers(nw (f)) = true and
  rm(enemy, q1, q2, c) = rm(p1, p2, p3, enc(k, r, p2)) and (k = k(enemy)) = false and
  (p2 = enemy) = false
(4) c-fkrm1(f, q1, q2, c) = true and
  c \in ciphers (nw(f)) = true and
  enc(k, r, p2) \in ciphers(nw(f)) = true and
  rm(enemy, q1, q2, c) = rm(p1, p2, p3, enc(k, r, p2)) and (k = k(enemy)) = false and
  Assuming p2 = enemy

Furthermore, as in v2(f, k, r) implies istep1(p1, p2, p3, k, r), this proof clause uses the prepared lemmas. If the validity condition is not met, the assumption c-fkrm1(f, q1, q2, c) = false is used. CafeOBJ returns true for this proof clause.

Proof clause about fkrm2

When the validity condition is met

```
open ISTEP .
    ops q1 q2 : -> Agent .
    op r1 : -> Rand .
    – eq c-fkrm2(f,q1,q2,r1) = true .
    eq r1 \in rands(nw(f)) = true .
    –
    eq (rm(p1,p2,p3,enc(k,r,p2)) = rm(enemy,q1,q2,enc(k(enemy),r1,q1))) = false .
    eq f’ = fkrm2(f,q1,q2,r1) .
```

```
    red istep1(p1,p2,p3,k,r) .
close
open ISTEP .
    ops q1 q2 : -> Agent .
    op r1 : -> Rand .
    – eq c-fkrm2(f,q1,q2,r1) = true .
    eq r1 \in rands(nw(f)) = true .
    –
    – eq rm(p1,p2,p3,enc(k,r,p2)) = rm(enemy,q1,q2,enc(k(enemy),r1,q1)) .
    eq p1 = enemy .
    eq p2 = q1 .
    eq p3 = q2 .
    eq k = k(enemy) .
    eq r = r1 .
    –
    eq f' = fkrm2(f,q1,q2,r1) .
    red istep1(p1,p2,p3,k,r) .
close
```

When the validity condition is not met

```
open ISTEP .
    ops q1 q2 : -> Agent .
    op r1 : -> Rand .
    eq c-fkrm2(f,q1,q2,r1) = false .
    eq f' = fkrm2(f,q1,q2,r1) .
    red istep1(p1,p2,p3,k,r) .
close
```

When the validity condition is met, two more cases are divided.

(1) c-fkrm2(f, q1, q2, r1) = true and
(rm(p1, p2, p3, enc(k, r, p2)) = rm(enemy, q1, q2, enc(k(enemy), r1, q1)))) = false

(2) c-fkrm2(f, q1, q2, r1) = true and
Assuming rm(p1, p2, p3, enc(k, r, p2)) = rm(enemy, q1, q2, enc(k(enemy), r1, q1))

If the validity condition is not met, the assumption c-fkrm2(f, q1, q2, r1) = false is used. CafeOBJ returns true for this proof clause. From the above, it was found that ture is returned in all the proof clauses of inv1.

3.7.3 Proof of inv2

Proof clause of induction basis

```
open INV .
```

```
    red inv2(init,k,r) .
close
```

CafeOBJ returns true for this proof clause. As in inv1, this proof clause is divided into five cases, and a proof clause is created for each case where the validity condition is satisfied and when it is not satisfied.

Proof clause about cdcm

When the validity condition is met

```
open ISTEP .
    ops q1 q2 : -> Agent .
    op r1 : -> Rand .
    – eq c-sdcm(f,q1,q2,r1) = true .
    eq r1 \in ur(f) = false .
    –
    eq f' = sdcm(f,q1,q2,r1) .
    red istep2(k,r) .
close
```

When the validity condition is not met

```
open ISTEP .
    ops q1 q2 : -> Agent .
    op r1 : -> Rand .
    eq c-sdcm(f,q1,q2,r1) = false .
    eq f' = sdcm(f,q1,q2,r1) .
    red istep2(k,r) .
close
```

We use the assumption that c-sdcm(f, q1, q2, r1) = true and the assumption that c-sdcm(f, q1, q2, r1) = false, respectively. CafeOBJ returns true for this proof clause.

Proof clause about cdrm

When the validity condition is met

```
open ISTEP .
    op q1 : -> Agent .
    op m1 : -> Msg .
    op nw1 : -> Network .
    – eq c-sdrm(f,q1,m1) = true .
    eq nw(f) = m1 , nw1 .
    eq cm?(m1) = true .
    eq q1 = dst(m1) .
    –
    eq (enc(k,r,enemy) = enc(k(dst(m1)),r(m1),dst(m1))) = false .
    eq f' = sdrm(f,q1,m1) .
    red istep2(k,r) .
```

```
close
open ISTEP .
    op q1 : -> Agent .
    op m1 : -> Msg .
    op nw1 : -> Network .
    – eq c-sdrm(f,q1,m1) = true .
    eq nw(f) = m1 , nw1 .
    eq cm?(m1) = true .
    eq q1 = dst(m1) .
    –
    – eq enc(k,r,enemy) = enc(k(dst(m1)),r(m1),dst(m1)) .
    eq k = k(dst(m1)) .
    eq r = r(m1) .
    eq dst(m1) = enemy .
    –
    eq f' = sdrm(f,q1,m1) .
    red istep2(k,r) .
close
```

When the validity condition is not met

```
open ISTEP .
    op q1 : -> Agent .
    op m1 : -> Msg .
    eq c-sdrm(f,q1,m1) = false .
    eq f' = sdrm(f,q1,m1) .
    red istep2(k,r) .
close
```

When the validity condition is satisfied, two more cases are divided.

(1) c-sdrm(f, q1, m1) = true and
Assuming (enc (k, r, enemy) = enc(k(dst(m1)), r(m1), dst(m1))) = false

(2) c-sdrm(f, q1, m1) = true and
Assuming enc(k, r, enemy) = enc(k(dst(m1)), r(m1), dst(m1))

If the validity condition is not satisfied, the assumption that c-sdrm (f, q1, m1) = false is used. CafeOBJ returns true for this proof clause.

Proof clause about fkcm1

When the validity condition is met

```
open ISTEP .
    ops q1 q2 : -> Agent .
    op r1 : -> Rand .
    – eq c-fkcm1(f,q1,q2,r1) = true .
    eq r1 \in rands(nw(f)) = true .
    –
```

```
    eq f’ = fkcm1(f,q1,q2,r1) .
    red istep2(k,r) .
close
```

When the validity condition is not met

```
open ISTEP .
    ops q1 q2 : -> Agent .
    op r1 : -> Rand .
    eq c-fkcm1(f,q1,q2,r1) = false .
    eq f’ = fkcm1(f,q1,q2,r1) .
    red istep2(k,r) .
close
```

We use the assumption that c-fkcm1(f, q1, q2, r1) = true and the assumption that c-fkcm1(f, q1, q2, r1) = false, respectively. CafeOBJ returns true for this proof clause.

Proof clause about fkrm1

When the validity condition is met

```
open ISTEP .
    ops q1 q2 : -> Agent .
    op c : -> Cipher .
    – eq c-fkrm1(f,q1,q2,c) = true .
    eq c \in ciphers(nw(f)) = true .
    –
    eq (enc(k,r,enemy) = c) = false .
    eq f’ = fkrm1(f,q1,q2,c) .
    red istep2(k,r) .
close
open ISTEP .
    ops q1 q2 : -> Agent .
    op c : -> Cipher .
    – eq c-fkrm1(f,q1,q2,c) = true .
    eq c \in ciphers(nw(f)) = true .
    –
    eq enc(k,r,enemy) = c .
    eq f’ = fkrm1(f,q1,q2,c) .
    red istep2(k,r) .
close
```

When the validity condition is not met

```
open ISTEP .
    ops q1 q2 : -> Agent .
    op c : -> Cipher .
    eq c-fkrm1(f,q1,q2,c) = false .
```

```
    eq f’ = fkrm1(f,q1,q2,c) .
    red istep2(k,r) .
close
```

When the validity condition is satisfied, two more cases are divided.

(1) c-fkrm1(f, q1, q2, c) = true and
    Assuming (enc (k, r, enemy) = c) = false
(2) c-fkrm1(f, q1, q2, c) = true and
    Assuming enc(k, r, enemy) = c

If the validity condition is not met, the assumption c-fkrm1(f, q1, q2, c) = false is used. CafeOBJ returns true for this proof clause.

Proof clause about fkrm2

When the validity condition is met

```
    noindent open ISTEP .
    ops q1 q2 : -> Agent .
    op r1 : -> Rand .
    – eq c-fkrm2(f,q1,q2,r1) = true .
    eq r1 \in rands(nw(f)) = true .
    –
    eq (enc(k,r,enemy) = enc(k(enemy),r1,q1)) = false .
    eq f’ = fkrm2(f,q1,q2,r1) .
    red istep2(k,r) .
close
open ISTEP .
    ops q1 q2 : -> Agent .
    op r1 : -> Rand .
    – eq c-fkrm2(f,q1,q2,r1) = true .
    eq r1 \in rands(nw(f)) = true .
    –
    – eq enc(k,r,enemy) = enc(k(enemy),r1,q1) .
    eq k = k(enemy) .
    eq r = r1 .
    eq q1 = enemy .
    eq f’ = fkrm2(f,q1,q2,r1) .
    red istep2(k,r) .
close
```

When the validity condition is not met

```
open ISTEP .
    ops q1 q2 : -> Agent .
    op r1 : -> Rand .
    eq c-fkrm2(f,q1,q2,r1) = false .
    eq f’ = fkrm2(f,q1,q2,r1) .
```

red istep2(k,r) .
close

When the validity condition is satisfied, two more cases are divided.

(1) eq c-fkrm2(f, q1, q2, r1) = true and
Assuming (enc(k, r, enemy) = enc(k(enemy), r1, q1)) = false

(2) eq c-fkrm2(f, q1, q2, r1) = true and
Assuming enc(k, r, enemy) = enc(k(enemy), r1, q1)

If the validity condition is not met, the assumption c-fkrm2 (f, q1, q2, r1) = false is used. CafeOBJ returns true for this proof clause. From the above, it was found that ture is returned in all the proof clauses of inv2. Therefore, the lemma is proved, and it can be seen that the proof of inv1 using this lemma is also correct.

From these two proofs, it was possible to verify that the IFF authentication protocol has the property of being able to identify an intruder. By using CafeOBJ in this way, it is possible to formal verify that the created protocol satisfies the desired properties.

3.8 Summary of IFF

As a survey of formal verification of authentication protocols using the proof score method, we first performed formal verification of IFF, a simpler authentication protocol than NSLPK, to understand how to model the exchange of two messages, the necessity of assuming the existence of an intruder, and how to create a simple authentication protocol specification and proof score.

Chapter 4

Formal Verification of NSLPK Authentication Protocol with Proof Scores

This Chapter presents the formal verification that NSLPK protocol enjoys some desired properties by writing proof scores.

4.1 NSLPK authentication protocol

The NSLPK authentication protocol is a new authentication protocol proposed by Lowe in 1995 by fixing a bug in the NSPK authentication protocol based on the public key method proposed by Needham and Schroeder in 1986. The NSLPK authentication protocol can be described as follows[1][2][3].

Msg1 p → q : $\epsilon_q(n_p, p)$
Msg2 q → p : $\epsilon_p(n_p, n_q, q)$
Msg3 p → q : $\epsilon_q(n_p)$

It is assumed that each entity is assigned a combination of private and public keys. The private key is known only to the entity to which it is assigned, and the public key is known to all the entities participating in the protocol. $\epsilon_p(m)$ is a message m encrypted with the public key of subject p. This ciphertext can be decrypted only by subject p who possesses the corresponding private key. n_p is the nonce generated by subject p. A nonce is a value that is used at most once. In this protocol, we further assume that the nonce is not analogous. As a nonce, we use a random number.

4.2 Assumption of intruder's presence

As in the IFF, we assume the existence of intruders attacking the protocol. The entities that attack the protocol are collectively modeled as a generic intruder.

4.3 Basic protocol behavior

When a subject p wants to mutually authenticate with another subject q, it generates a nonce n_p, encrypts the pair of n_p and the identifier p with q's public key, and sends $\epsilon_q(n_p, p)$ to q. When q receives a message that seems to be of type Msg1, it first tries to decrypt it. When q receives a message that seems to be of type Msg1, it first tries to decrypt it, and if it is able to retrieve the nonce n_p and the subject's identifier p through decryption, it generates a new nonce n_p and sends a message $\epsilon_p(n_p, n_q, q)$ encrypted with the three sets of n_p, n_q, and the identifier q using p's public key to p. When p receives the message, it sends a message After sending $\epsilon_q(n_p, p)$ to q, when it receives a message that seems to be of the type Msg2, it first tries to decrypt it. If one of the nonces is equal to n_p and the identifier is q, then the communication partner of p can be verified to be q, and p is authenticated. After this, another message $\epsilon_q(n_p)$ is sent to q, which encrypts another nonce n_q with q's public key, and after q sends $\epsilon_p(n_p, n_q, q)$, it receives a message that seems to be of the type Msg3. When we receive a message that seems to be of Msg3 type, we first try to decrypt it. If the decryption yields a nonce, and it is equal to n_q, then q can be sure that the communication partner of q is p, and q is mutually authenticated by p. In this case, p and q believe that the two nonces n_p and n_q are secret information shared only by p and q.

4.3.1 Confidentiality

Confidentiality is one of the properties that NSLPK must satisfy. This is the property that "p and q believe that this is shared only by p and q and that the two nonces cannot be leaked to third parties other than p and q.

4.4 Creating a model of NSLPK

Since NSLPK assumes the existence of intruders, we also assume the existence of intruders in creating the model of NSLPK, the observation transition system $\mathcal{S}_{NSLPK}$.

The observation transition system $\mathcal{S}_{NSLPK}$ consists of an observation function $\mathcal{O}_{NSLPK}$, a set of initial states $\mathcal{L}_{NSLPK}$, and a set of transition functions $\mathcal{T}_{NSLPK}$. $\mathcal{O}_{NSLPK}$, $\mathcal{L}_{NSLPK}$, and $\mathcal{T}_{NSLPK}$ are $\mathcal{O}_{NSLPK}$, $\mathcal{L}_{NSLPK}$, and $\mathcal{T}_{NSLPK}$ are as follows.

$\mathcal{O}_{NSLPK}\{nw : Field \rightarrow Network,$
$ur : Field \rightarrow URands\}$
$\mathrm{L}_{NSLPK}\{init \mid nw(init) = void \wedge$
$ur(init) = empty\}$
$\mathrm{T}_{NSLPK}\{sdm1 : System\ Principal\ Principal\ Random \rightarrow System,$
$sdm2 : System\ Principal\ Random\ Message \rightarrow System,$
$sdm3 : System\ Principal\ Random\ Message\ Message \rightarrow System,$
$fkm11 : System\ Principal\ Principal\ Principal\ Cipher1 \rightarrow System,$
$fkm12 : System\ Principal\ Principal\ Principal\ Nonce \rightarrow System,$
$fkm21 : System\ Principal\ Principal\ Principal\ Cipher2 \rightarrow System,$
$fkm22 : System\ Principal\ Principal\ Principal\ Nonce\ Nonce \rightarrow$
$System,$
$fkm31 : System\ Principal\ Principal\ Principal\ Cipher3 \rightarrow System,$
$fkm32 : System\ Principal\ Principal\ Principal\ Nonce \rightarrow System\}$

where Field, Network, URands, Principal, Random, Message, Nonce, and Cipher1, 2, 3 are the state, network, multiset of random numbers, subject, random number, message, nonce, and ciphertext types of $\mathcal{S}_{NSLPK}$, respectively. Nons and ciphertext types.

The network is also modeled as a multiset of messages. Furthermore, due to the presence of intruders, we assume that once a message is sent, it stays in the network. This is because once a message is sent, it may be sent many times by intruders. Also, void represents an empty multiset for the network, and empty represents an empty multiset for random numbers.

4.5 Observation function and transition function

Given a state F, the observation function nw,ur returns the random numbers available in that state (ur(F)) and the multiset of messages sent up to that state (nw(F)). The transition functions sdm1, sdm2, and sdm3 correspond to the subject sending Msg1, Msg2, and Msg3, respectively, according to the protocol. In contrast, the transition functions fkm11, fkm12, fkm21, fkm22, fkm31, and fkm32 correspond to forging Msg1, Msg2, and Msg3 using random numbers and ciphertext collected by the intruder, respectively.

4.6 Create CafeOBJ specification for NSLPK

Create a CafeOBJ specification for $\mathcal{S}_{NSLPK}$. Since messages in NSLPK are ciphertext, the network is a multiset of ciphertext as in IFF. Since messages in NSLPK are ciphertext, the network is a multiset of ciphertext, as in IFF.

4.6.1 Creating a CafeOBJ specification for a data type

Specifications of principal : PRINCIPAL

```
mod* PRINCIPAL principal-sort Principal {
    [Principal]
    op intruder : -> Principal
    var P : Principal
}
```

The constant intruder represents a generic intruder.

Specifications of random numbers : RANDOM

```
mod* RANDOM principal-sort Random {
    [Random]
    var R : Random
}
```

Specifications of nonce : NONCE

```
mod! NONCE principal-sort Nonce {
    pr(PRINCIPAL + RANDOM)
    [Nonce]
    op n : Principal Principal Random -> Nonce
    op creator : Nonce -> Principal
    op forwhom : Nonce -> Principal
    op random : Nonce -> Random
    vars N1 N2 : Nonce
    var C : Principal
    var W : Principal
    var R : Random
    eq creator(n(C,W,R)) = C .
    eq forwhom(n(C,W,R)) = W .
    eq random(n(C,W,R)) = R .
    eq (N1 = N2) = (creator(N1) = creator(N2) and forwhom(N1) = for-
whom(N2)
       and random(N1) = random(N2)) .
}
```

The operation n is a construct of a nonce. Given two subjects C (the creator of the nonce) and W (the recipient of the nonce), and a random number R, the term n(C, W, R) represents the nonce created by subject C to authenticate subject W. The uniqueness of that nonce depends on the random number R. Given a term representing a nonce, the operations creator, forwhom, and random return the first, second, and third arguments of that term, respectively.

Specifications of cipher1 : CIPHER1

```
! CIPHER1 principal-sort Cipher1 {
   pr(PRINCIPAL + NONCE)
   [Cipher1]
   op enc1 : Principal Nonce Principal -> Cipher1
   op key : Cipher1 -> Principal
   op nonce : Cipher1 -> Nonce
   op principal : Cipher1 -> Principal
   vars E11 E12 : Cipher1
   var K : Principal
   var N : Nonce
   var P : Principal
   eq key(enc1(K,N,P)) = K .
   eq nonce(enc1(K,N,P)) = N .
   eq principal(enc1(K,N,P)) = P .
   eq (E11 = E12) = (key(E11) = key(E12) and nonce(E11) = nonce(E12)
and
     principal(E11) = principal(E12)) .
}
```

Specifications of cipher2 :CIPHER2

```
mod! CIPHER2 principal-sort Cipher2 {
   pr(PRINCIPAL + NONCE)
   [Cipher2]
   op enc2 : Principal Nonce Nonce Principal -> Cipher2
   op key : Cipher2 -> Principal
   op nonce1 : Cipher2 -> Nonce
   op nonce2 : Cipher2 -> Nonce
   op principal : Cipher2 -> Principal
   vars E21 E22 : Cipher2
   var K : Principal
   var N1 : Nonce
   var N2 : Nonce
   var P : Principal
   eq key(enc2(K,N1,N2,P)) = K .
   eq nonce1(enc2(K,N1,N2,P)) = N1 .
   eq nonce2(enc2(K,N1,N2,P)) = N2 .
```

```
    eq principal(enc2(K,N1,N2,P)) = P .
    eq (E21 = E22) = (key(E21) = key(E22) and nonce1(E21) = nonce1(E22)
and
      nonce2(E21) = nonce2(E22) and principal(E21) = principal(E22)) .
}
```

Specifications of cipher3 : CIPHER3

```
mod! CIPHER3 principal-sort Cipher3 {
    pr(PRINCIPAL + NONCE)
    [Cipher3]
    op enc3 : Principal Nonce -> Cipher3
    op key : Cipher3 -> Principal
    op nonce : Cipher3 -> Nonce
    vars E31 E32 : Cipher3
    var K : Principal
    var N : Nonce
    eq key(enc3(K,N)) = K .
    eq nonce(enc3(K,N)) = N .
    eq (E31 = E32) = (key(E31) = key(E32) and nonce(E31) = nonce(E32))
.
}
```

The operations enc1, enc2, and enc3 are the ciphertext constructors of Msg1, Msg2, and Msg3, respectively. Given the subjects K and P and the nonces N, N1, and N2, the terms enc1(K,N,P), enc2(K,N1,N2,P), and enc3(K,N) are, respectively, the ciphertext $\epsilon_q(n_p, p)$, $\epsilon_p(n_p, n_q, q)$ and $\epsilon_q(n_p)$, respectively. Given a term representing a ciphertext, the operations Key and nonce return the first and second arguments of the term, respectively.

Specifications of message : MESSAGE

```
mod! MESSAGE principal-sort Message {
    pr(PRINCIPAL + CIPHER1 + CIPHER2 + CIPHER3)
    [Message]
    op m1 : Principal Principal Principal Cipher1 -¿ Message
    op m2 : Principal Principal Principal Cipher2 -¿ Message
    op m3 : Principal Principal Principal Cipher3 -¿ Message
    op m1? : Message -> Bool
    op m2? : Message -> Bool
    op m3? : Message -> Bool
    op creator : Message -> Principal
    op sender : Message -> Principal
    op receiver : Message -> Principal
    op cipher1 : Message -> Cipher1
    op cipher2 : Message -> Cipher2
```

```
op cipher3 : Message -> Cipher3
vars M M1 M2 : Message
vars C S R : Principal
var E1 : Cipher1
var E2 : Cipher2
var E3 : Cipher3
eq m1?(m1(C,S,R,E1)) = true .
eq m1?(m2(C,S,R,E2)) = false .
eq m1?(m3(C,S,R,E3)) = false .
eq m2?(m1(C,S,R,E1)) = false .
eq m2?(m2(C,S,R,E2)) = true .
eq m2?(m3(C,S,R,E3)) = false .
eq m3?(m1(C,S,R,E1)) = false .
eq m3?(m2(C,S,R,E2)) = false .
eq m3?(m3(C,S,R,E3)) = true .
eq creator(m1(C,S,R,E1)) = C .
eq creator(m2(C,S,R,E2)) = C .
eq creator(m3(C,S,R,E3)) = C .
eq sender(m1(C,S,R,E1)) = S .
eq sender(m2(C,S,R,E2)) = S .
eq sender(m3(C,S,R,E3)) = S .
eq receiver(m1(C,S,R,E1)) = R .
eq receiver(m2(C,S,R,E2)) = R .
eq receiver(m3(C,S,R,E3)) = R .
eq cipher1(m1(C,S,R,E1)) = E1 .
eq cipher2(m2(C,S,R,E2)) = E2 .
eq cipher3(m3(C,S,R,E3)) = E3 .
ceq (M1 = M2) = (m1?(M2) and creator(M1) = creator(M2) and sender(M1) = sender(M2) and receiver(M1) = receiver(M2) and cipher1(M1) = cipher1(M2)) if m1?(M1) .
ceq (M1 = M2) = (m2?(M2) and creator(M1) = creator(M2) and sender(M1) = sender(M2) and receiver(M1) = receiver(M2) and cipher2(M1) = cipher2(M2)) if m2?(M1) .
ceq (M1 = M2) = (m3?(M2) and creator(M1) = creator(M2) and sender(M1) = sender(M2) and receiver(M1) = receiver(M2) and cipher3(M1) = cipher3(M2)) if m3?(M1) .
}
```

The operations m1, m2, and m3 take three subjects and ciphertext 1, 2, and 3 as arguments and return a message. the three subjects represent the true author, sender, and receiver of the message, respectively, and the operations correspond to Msg1, Msg2, and Msg3, respectively.

The operations m1?, m2?, and m3? are predicates that determine whether a given message is Msg1, Msg2, or Msg3, respectively, and the operations creator, sender, and receiver are predicates that determine the true creator, sender, and receiver, respectively, from a given message.

The operations cipher1, cipher2, and cipher3 return ciphertext1, ciphertext2, and ciphertext3 from the given message, respectively.

Specification of a generic multiset for messages : BAG

```
mod! BAG (D :: EQTRIV) principal-sort Bag {
    [Elt.D < Bag]
    op void : -> Bag
    op _,_ : Bag Bag -> Bag {assoc comm id: void}
    op _\in_ : Elt.D Bag -> Bool
    var B : Bag
    vars E1 E2 : Elt.D
    eq E1 \in void = false .
    ceq E1 \in (E2,B) = true if E1 = E2 .
    ceq E1 \in (E2,B) = E1 \in B if not(E1 = E2) .
}
```

Specification of a generic multiset for random numbers : SET

```
    noindent mod! SET (D :: EQTRIV) principal-sort Set {
    [Elt.D < Set]
    op empty : -> Set
    op __ : Set Set -> Set assoc comm idem id: empty
    op _\in_ : Elt.D Set -> Bool
    var S : Set
    vars E1 E2 : Elt.D
    eq E1 \in empty = false .
    ceq E1 \in (E2 S) = true if E1 = E2 .
    ceq E1 \in (E2 S) = E1 \in S if not(E1 = E2) .
}
```

Specify the dummy argument D

```
mod* COLLECTION(D :: TRIV) principal-sort Collection {
    [Elt.D < Collection]
    op _\in_ : Elt.D Collection -> Bool
}
```

D is declared as a sub-sort of sort Bag and sort Col. D is declared as a sub-sort of sort Bag and sort Col. This means that each element of a multiset can be regarded as a multiset with only that element. The constants void and empty represent the empty multiset, while the operations _,_ and __ are the constructors of the non-empty multiset. Also, the operations _,_ and __ are declared to satisfy the exchange law (comm) and the join law (assoc).

Specification of a multiset of ciphertext : NETWORK

```
mod! NETWORK {
pr(PRINCIPAL + NONCE)
pr(CIPHER1 + CIPHER2 + CIPHER3)
pr(BAG(MESSAGE)*sort Bag -¿ Network)
pr(COLLECTION(NONCE)*sort Collection -> ColNonce)
pr(COLLECTION(CIPHER1)*sort Collection -> ColCipher1)
pr(COLLECTION(CIPHER2)*sort Collection -> ColCipher2)
pr(COLLECTION(CIPHER3)*sort Collection -> ColCipher3)
op cnonce : Network -> ColNonce
op cenc1 : Network -> ColCipher1
op cenc2 : Network -> ColCipher2
op cenc3 : Network -> ColCipher3
var NW : Network
var M : Message
var N : Nonce
var E1 : Cipher1
var E2 : Cipher2
var E3 : Cipher3
eq N \in cnonce(void) = (creator(N) = intruder) .
ceq N \in cnonce(M,NW) = true if m1?(M) and key(cipher1(M)) =
   intruder and nonce(cipher1(M)) = N .
ceq N \in cnonce(M,NW) = true if m2?(M) and key(cipher2(M)) =
   intruder and nonce1(cipher2(M)) = N .
ceq N \in cnonce(M,NW) = true if m2?(M) and key(cipher2(M)) =
   intruder and nonce2(cipher2(M)) = N .
ceq N \in cnonce(M,NW) = true if m3?(M) and key(cipher3(M)) =
   intruder and nonce(cipher3(M)) = N .
ceq N \in cnonce(M,NW) = N \in cnonce(NW) if not(m1?(M) and key(cip
    her1(M)) = intruder and nonce(cipher1(M)) = N) and not(m2?(M)
    and key(cipher2(M)) = intruder and nonce1(cipher2(M)) = N) and
    not(m2?(M) and key(cipher2(M)) = intruder and nonce2(cipher2(M))
    = N) and not(m3?(M) and key(cipher3(M))
    = intruder and nonce(cipher3(M)) = N) .
eq E1 \in cenc1(void) = false .
ceq E1 \in cenc1(M,NW) = true if m1?(M) and not(key(cipher1(M))
   = intruder) and E1 = cipher1(M) .
ceq E1 \in cenc1(M,NW) = E1 \in cenc1(NW) if not(m1?(M) and not(key
    (cipher1(M)) = intruder) and E1 = cipher1(M)) .
eq E2 \in cenc2(void) = false .
```

```
  ceq E2 \in cenc2(M,NW) = true if m2?(M) and not(key(cipher2(M))
      = intruder) and E2 = cipher2(M) .
  ceq E2 \in cenc2(M,NW) = E2 \in cenc2(NW) if not(m2?(M) and not(key
      (cipher2(M)) = intruder) and E2 = cipher2(M)) .
  eq E3 \in cenc3(void) = false .
  ceq E3 \in cenc3(M,NW) = true if m3?(M) and not(key(cipher3(M)) =
      intruder) and E3 = cipher3(M) .
  ceq E3 \in cenc3(M,NW) = E3 \in cenc3(NW) if not(m3?(M) and not(key
      (cipher3(M)) = intruder) and E3 = cipher3(M)) .
  }
```

It embodies a multiset of ciphertext from a generic multiset. Sort names are renamed from BAG to NETWORK, and from Col to ColNonce and ColCiphers1, ColCiphers2, and ColCiphers3.

The operations cnonce and key return the nonce and ciphertext in the given multiset, respectively.

4.6.2 Creation of CafeOBJ specifications for observation transition system

Having created the CafeOBJ specification for the data type used in the observation transition system $\mathcal{S}_{NSLPK}$, the next step is to create the CafeOBJ specification for $\mathcal{S}_{NSLPK}$.

Specification of the observation transition system : NSLPK

```
mod* NSLPK {
  pr(NETWORK)
  pr(SET(RANDOM)*sort Set -> URand)
  [System]
  op init : -> System
  op ur : System -> URand
  op nw : System -> Network
  op sdm1 : System Principal Principal Random -> System constr
  op sdm2 : System Principal Random Message -> System constr
  op sdm3 : System Principal Random Message Message -> System constr
  op fkm11 : System Principal Principal Cipher1 -> System constr
  op fkm12 : System Principal Principal Nonce -> System constr
  op fkm21 : System Principal Principal Cipher2 -> System constr
  op fkm22 : System Principal Principal Nonce Nonce -> System constr
  op fkm31 : System Principal Principal Cipher3 -> System constr
  op fkm32 : System Principal Principal Nonce -> System constr
  var S : System
```

```
    vars M M1 M2 : Message
    vars P Q : Principal
    var R : Random
    vars N N1 N2 : Nonce
    var E1 : Cipher1
    var E2 : Cipher2
    var E3 : Cipher3
    ⋯
}
```

The constant init represents an arbitrary initial state of $\mathcal{S}_{NSLPK}$. The operations nw and ur correspond to the observation functions of $\mathcal{S}_{NSLPK}$, and the remaining operations correspond to the transition functions. The equations defining the initial state and behavior of $\mathcal{S}_{NSLPK}$ are declared at the $\cdots$. They are described in the following sections.

Defining the initial state :

```
    eq nw(init) = void .
    eq ur(init) = empty .
```

These equations correspond to $\mathcal{L}_{NSLPK}$.

Definition of the transition function : sdm1

```
    op c-sdm1 : System Principal Principal Random -> Bool
    eq c-sdm1(S,P,Q,R) = not(R \in ur(S)) .
    ceq ur(sdm1(S,P,Q,R)) = R ur(S) if c-sdm1(S,P,Q,R) .
    ceq nw(sdm1(S,P,Q,R)) = m1(P,P,Q,enc1(Q,n(P,Q,R),P)) , nw(S) if c-sdm1(S,P,Q,R) .
    ceq sdm1(S,P,Q,R) = S if not c-sdm1(S,P,Q,R) .
```

Each transition function has its own validity condition, and the validity condition of this transition function is that R is not included in the multiset of random numbers. Given a message m1(P,P,Q,enc1(Q,n(P,Q,R),P)) generated by this transition function, add the message to the multiset nw(S) of the network and the random numbers to the multiset ur(S) of random numbers.

Definition of the transition function : sdm2

```
    op c-sdm2 : System Principal Random Message -> Bool
    eq c-sdm2(S,Q,R,M)    = (M \in nw(S) and m1?(M) and receiver(M) =
        Q and key(cipher1(M)) = Q and principal(cipher1(M)) = sender(M)
        and not(R \in ur(S))) .
    ceq ur(sdm2(S,Q,R,M)) = R ur(S) if c-sdm2(S,Q,R,M) .
    ceq nw(sdm2(S,Q,R,M)) = m2(Q,Q,sender(M),enc2(sender(M),nonce(cipher1(M)),n(Q,sender(M),R),Q)),nw(S) if c-sdm2(S,Q,R,M) .
    ceq sdm2(S,Q,R,M) = S if not c-sdm2(S,Q,R,M) .
```

The validity condition of this transition function is that there is a message 1 that appears to have been sent, represented by sender(M), and that the random number R is truly new. Given sdm2(S,Q,R,M) generated by this transition function, we add the message to the network multiset nw(S) and the random number to the multiset of random numbers ur(S).

Definition of the transition function : sdm3

```
op c-sdm3 : System Principal Random Message Message -> Bool
eq c-sdm3(S,P,R,M1,M2) = (M1 \in nw(S) and M2 \in nw(S)
   and m1?(M1) and m2?(M2) and creator(M1) = P and sender(M1) =
   P and receiver(M1) = sender(M2) and key(cipher1(M1)) = sender(M2)
   and nonce(cipher1(M1)) = n(P,sender(M2),R) and principal(cipher1(M
  1)) = P and receiver(M2) = P and key(cipher2(M2)) = P and nonce1(ciph
  er2(M2)) = n(P,sender(M2),R) and principal(cipher2(M2)) = sender(M2)
  ) .
eq ur(sdm3(S,P,R,M1,M2)) = ur(S) .
ceq nw(sdm3(S,P,R,M1,M2)) = m3(P,P,sender(M2),enc3(sender(M2),no-
nce2(cipher2(M2)))) , nw(S) if c-sdm3(S,P,R,M1,M2) .
ceq sdm3(S,P,R,M1,M2) = S if not c-sdm3(S,P,R,M1,M2) .
```

The validity condition of this transition function is that the subject P sends message 1, represented by the receiver (M1), to Q, and there is a message 2 that appears to have been sent from Q to P in response to M1. Given sdm3(S,P,R,M1,M2) generated by this transition function, we add the message to the network's multiset nw(S) and the random number to the multiset of random numbers ur(S).

Definition of the transition function : fkm11

```
op c-fkm11 : System Principal Principal Cipher1 -> Bool
eq c-fkm11(S,P,Q,E1) = E1 \in cenc1(nw(S)) .
eq ur(fkm11(S,P,Q,E1)) = ur(S) .
ceq nw(fkm11(S,P,Q,E1)) = m1(intruder,P,Q,E1) , nw(S) if c-fkm11(S,P,Q
   ,E1)
ceq fkm11(S,P,Q,E1) = S if not c-fkm11(S,P,Q,E1) .
```

The validity condition of this transition function is that E1 is included in the multiset of ciphertext 1. Given fkm11(S,P,Q,E1) generated by this transition function, add the message to the multiset of networks nw(S) and the random number to the multiset of random numbers ur(S).

Definition of the transition function : fkm12

```
op c-fkm12 : System Principal Principal Nonce -> Bool
eq c-fkm12(S,P,Q,N) = N \in cnonce(nw(S)) .
eq ur(fkm12(S,P,Q,N)) = ur(S) .
ceq nw(fkm12(S,P,Q,N)) = m1(intruder,P,Q,enc1(Q,N,P)) , nw(S) if c-
   fkm12(S,P,Q,N) .
ceq fkm12(S,P,Q,N) = S if not c-fkm12(S,P,Q,N) .
```

The validity condition of this transition function is that N is contained in a multiset of nonces. Given fkm12(S,P,Q,N) generated by this transition function, add the message to the multiset of networks nw(S) and the random number to the multiset of random numbers ur(S).

Definition of the transition function : fkm21

```
op c-fkm21 : System Principal Principal Cipher2 -> Bool
eq c-fkm21(S,P,Q,E2) = E2 \in cenc2(nw(S)) .
eq ur(fkm21(S,P,Q,E2)) = ur(S) .
ceq nw(fkm21(S,P,Q,E2)) = m2(intruder,P,Q,E2) , nw(S) if c-fkm21(S,P,Q,E2)
ceq fkm21(S,P,Q,E2) = S if not c-fkm21(S,P,Q,E2) .
```

The validity condition of this transition function is that E2 is included in the multiset of ciphertext 2. Given fkm21(S,P,Q,E2) generated by this transition function, add the message to the multiset of networks nw(S) and the random number to the multiset of random numbers ur(S).

Definition of the transition function : fkm22

```
op c-fkm22 : System Principal Principal Nonce Nonce -> Bool
eq c-fkm22(S,P,Q,N1,N2) = N1 \in cnonce(nw(S)) and N2 ∈ cnonce(nw(S)).
eq ur(fkm22(S,P,Q,N1,N2)) = ur(S) .
ceq nw(fkm22(S,P,Q,N1,N2)) = m2(intruder,P,Q,enc2(Q,N1,N2,P)) , nw(S) if c-fkm22(S,P,Q,N1,N2) .
ceq fkm22(S,P,Q,N1,N2) = S if not c-fkm22(S,P,Q,N1,N2) .
```

The validity condition of this transition function is that N1 and N2 are contained in the nonce multiset. Given fkm22(S,P,Q,N1,N2) generated by this transition function, add the message to the multiset of networks nw(S) and the random number to the multiset of random numbers ur(S).

Definition of the transition function : fkm31

```
op c-fkm31 : System Principal Principal Cipher3 -> Bool
eq c-fkm31(S,P,Q,E3) = E3 \in cenc3(nw(S)) .
eq ur(fkm31(S,P,Q,E3)) = ur(S) .
ceq nw(fkm31(S,P,Q,E3)) = m3(intruder,P,Q,E3) , nw(S) if c-fkm31(S,P,Q,E3) .
ceq fkm31(S,P,Q,E3) = S if not c-fkm31(S,P,Q,E3)
```

The validity condition of this transition function is that E3 is included in the multiset of ciphertext 3. Given fkm31(S,P,Q,E3) generated by this transition function, add the message to the multiset of networks nw(S) and the random number to the multiset of random numbers ur(S).

Definition of the transition function : fkm32

```
op c-fkm32 : System Principal Principal Nonce -> Bool
eq c-fkm32(S,P,Q,N) = N \in cnonce(nw(S)) .
eq ur(fkm32(S,P,Q,N)) = ur(S) .
```

```
ceq nw(fkm32(S,P,Q,N)) = m3(intruder,P,Q,enc3(Q,N)) , nw(S) if c-fkm32
    (S,P,Q,N) .
ceq fkm32(S,P,Q,N) = S if not c-fkm32(S,P,Q,N)
```

The validity condition of this transition function is that N is contained in a multiset of nonces. Given fkm32(S,P,Q,N) generated by this transition function, add the message to the multiset of networks nw(S) and the random number to the multiset of random numbers ur(S).

4.7 Verification of NSLPK

NSLPK assumes the existence of apparently indistinguishable intruders. We verify that the NSLPK authentication protocol, modeled as described above, can identify such an intruder. In the following, we provide proof using CafeOBJ.

4.7.1 Verification of intruder identification

```
mod INV {
 pr(NSLPK)
 op e1 : -> Cipher1
 op e2 : -> Cipher2
 op e3 : -> Cipher3
 op r : -> Random
 ops n n1 n2 : -> Nonce
 ops p q p1 q1 : -> Principal
 op inv100 : System Cipher1 -> Bool
 op inv110 : System Cipher2 -> Bool
 op inv120 : System Cipher3 -> Bool
 op inv130 : System Nonce -> Bool
 op inv140 : System Cipher1 -> Bool
 op inv150 : System Cipher2 -> Bool
 op inv160 : System Nonce -> Bool
 op inv170 : System Principal Principal Principal Random Nonce -> Bool
 op inv180 : System Principal Principal Principal Random Nonce -> Bool
 op inv190 : System Principal Principal Random Nonce -> Bool
 op inv200 : System Principal Principal Random -> Bool
 op inv210 : System Principal Principal Random -> Bool
 op inv220 : System Principal Principal Random Nonce -> Bool
 op inv230 : System Principal Principal Random -> Bool
 op inv240 : System Principal Principal Random -> Bool
```

```
op inv250 : System Principal Principal Principal Random Nonce -> Bool
op inv260 : System Principal Principal Nonce Nonce -> Bool
var S : System
var E1 : Cipher1
var E2 : Cipher2
var E3 : Cipher3
var R : Random
vars N N1 N2 : Nonce
vars P Q P1 Q1 : Principal
eq inv100(S,E1) = (E1 in cenc1(nw(S)) implies not(key(E1) = intruder))
eq inv110(S,E2) = (E2 in cenc2(nw(S)) implies not(key(E2) = intruder))
eq inv120(S,E3) = (E3 in cenc3(nw(S)) implies not(key(E3) = intruder))
eq inv130(S,N) = (N in cnonce(nw(S)) implies (creator(N) = intruder or
   forwhom(N) = intruder)) .
eq inv140(S,E1) = (E1 in cenc1(nw(S)) and principal(E1) = intruder
    implies nonce(E1) in cnonce(nw(S))) .
eq inv150(S,E2) = (E2 in cenc2(nw(S)) and principal(E2) = intruder
    implies nonce2(E2) ∈ cnonce(nw(S))).
eq inv160(S,N) = (creator(N) = intruder implies N in cnonce(nw(S))) .
eq inv170(S,P,Q,Q1,R,N) = (not(P = intruder) and m1(P,P,Q,enc1(Q,n(P
   ,Q,R),P)) in nw(S) and m2(Q1,Q,P,enc2(P,n(P,Q,R),N,Q))
   in nw(S) implies m2(Q,Q,P,enc2(P,n(P,Q,R),N,Q)) in nw(S)) .
eq inv180(S,P,Q,P1,R,N) = (not(Q = intruder) and m2(Q,Q,P,enc2(P,N,n
   (Q,P,R),Q)) in nw(S) and m3(P1,P,Q,enc3(Q,n(Q,P,R))) in nw(S) im
    plies m3(P,P,Q,enc3(Q,n(Q,P,R))) in nw(S)) .
eq inv190(S,P,Q,R,N) = (not(P = intruder) and enc2(P,n(P,Q,R),N,Q)
    in cenc2(nw(S)) implies R in ur(S)) .
eq inv200(S,P,Q,R) = (not(P = intruder) and not(Q = intruder) and
    enc1(Q,n(P,Q,R),P) in cenc1(nw(S)) implies R in ur(S)) .
eq inv210(S,P,Q,R) = (not(P = intruder) and n(P,Q,R) in cnonce(nw(S))
    implies R in ur(S)) .
eq inv220(S,P,Q,R,N) = (not(P = intruder) and m1(P,P,Q,enc1(Q,n(P,Q
   ,R),P)) in nw(S) and enc2(P,n(P,Q,R),N,Q) in cenc2(nw(S)) implies
   m2(Q,Q,P,enc2 (P,n(P,Q,R),N,Q)) in nw(S)) .
eq inv230(S,P,Q,R) = (not(Q = intruder) and not(P = intruder) and
    enc3(Q,n(Q,P,R)) in cenc3(nw(S)) implies m3(P,P,Q,enc3(Q,n(Q,P,R)))
   in nw(S)) .
eq inv240(S,P,Q,R) = (not(Q = intruder) and enc3(Q,n(Q,P,R)) in cenc3(n
   w(S)) implies R in ur(S)) .
eq inv250(S,P1,P,Q,R,N) = (not(Q = intruder) and not(P1 = intruder)
    and enc2(P1,N,n(Q,P,R),Q) in cenc2(nw(S)) implies R in ur(S)) .
```

```
  eq inv260(S,P,Q,N1,N2) = (not(P = intruder) and m2(P,P,Q,enc2(Q,N1,N
    2,P)) in nw(S) implies forwhom(N2) = Q) .
}
```

The operations inv100, inv170, and inv180 are properties of NSLPK that we want to prove in the main. The others are supplementary problems. The operations inv100 160 are properties related to confidentiality, and the operations inv170 260 are properties related to mutual authentication.

We declare the module that describes the logical formula to be proved at each induction step as follows.

```
mod ISTEP {
  pr(INV)
  ops s s' : -> System
  op istep100 : Cipher1 -> Bool
  op istep110 : Cipher2 -> Bool
  op istep120 : Cipher3 -> Bool
  op istep130 : Nonce -> Bool
  op istep140 : Cipher1 -> Bool
  op istep150 : Cipher2 -> Bool
  op istep160 : Nonce -> Bool
  op istep170 : Principal Principal Principal Random Nonce -> Bool
  op istep180 : Principal Principal Principal Random Nonce -> Bool
  op istep190 : Principal Principal Random Nonce -> Bool
  op istep200 : Principal Principal Random -> Bool
  op istep210 : Principal Principal Random -> Bool
  op istep220 : Principal Principal Random Nonce -> Bool
  op istep230 : Principal Principal Random -> Bool
  op istep240 : Principal Principal Random -> Bool
  op istep250 : Principal Principal Principal Random Nonce -> Bool
  op istep260 : Principal Principal Nonce Nonce -> Bool
  var E1 : Cipher1
  var E2 : Cipher2
  var E3 : Cipher3
  var R : Random
  vars N N1 N2 : Nonce
  vars P Q P1 Q1 : Principal
  eq istep100(E1) = inv100(s,E1) implies inv100(s',E1) .
  eq istep110(E2) = inv110(s,E2) implies inv110(s',E2) .
  eq istep120(E3) = inv120(s,E3) implies inv120(s',E3) .
  eq istep130(N) = inv130(s,N) implies inv130(s',N) .
  eq istep140(E1) = inv140(s,E1) implies inv140(s',E1) .
  eq istep150(E2) = inv150(s,E2) implies inv150(s',E2) .
```

```
  eq istep160(N) = inv160(s,N) implies inv160(s',N) .
  eq istep170(P,Q,Q1,R,N) = inv170(s,P,Q,Q1,R,N) implies inv170(s',P,Q,
    Q1,R,N).
  eq istep180(P,Q,P1,R,N) = inv180(s,P,Q,P1,R,N) implies inv180(s',P,Q,P
    1,R,N).
  eq istep190(P,Q,R,N) = inv190(s,P,Q,R,N) implies inv190(s',P,Q,R,N) .
  eq istep200(P,Q,R) = inv200(s,P,Q,R) implies inv200(s',P,Q,R) .
  eq istep210(P,Q,R) = inv210(s,P,Q,R) implies inv210(s',P,Q,R) .
  eq istep220(P,Q,R,N) = inv220(s,P,Q,R,N) implies inv220(s',P,Q,R,N) .
  eq istep230(P,Q,R) = inv230(s,P,Q,R) implies inv230(s',P,Q,R) .
  eq istep240(P,Q,R) = inv240(s,P,Q,R) implies inv240(s',P,Q,R) .
  eq istep250(P1,P,Q,R,N) = inv250(s,P1,P,Q,R,N) implies inv250(s',P1,P,Q
    ,R,N).
  eq istep260(P,Q,N1,N2) = inv260(s,P,Q,N1,N2) implies inv260(s',P,Q,N1,
    N2).
}
```

The constant f represents an arbitrary state, and the constant f' represents the posterior state of state f.

4.7.2 Proof scores of inv100 through inv260

Like iff authentication protocol, NSLPK authentication protocol's proofs are divided into as many cases as there are transition functions. The proofs are divided into cases where the validity condition holds and where it does not hold for each case. If a supplementary title is needed, it is used. When the NSLPK authentication protocol created above is executed, the program returns true in all cases, and we can formally verify that the created protocol satisfies the desired properties.

4.8 Summary of NSLPK

As a survey of formal verification of authentication protocols using the proof score method, we conducted a formal verification of NSLPK, which is more complex than IFF because it assumes the exchange of three messages. We were able to understand that the modeling, specification, and proof scoring are more complex than IFF because it assumes three messages are exchanged.

Chapter 5

Formal Verification of IFF Authentication Protocol with CiMPA and CiMPG

This Chapter gives two more ways of the formal verification with IFF protocol.

5.1 Rewriting the specification of the IFF authentication protocol to use CiMPG and CiMPA

In order to formally verify the IFF authentication protocol with CiMPG/CiMPA, we will first rewrite the specification. First, we will put the INV module created in 3.7.1 into the IFF module as follows.

```
op inv1 : Field Agent Agent Agent Key Rand -> Bool
op inv2 : Field Key Rand -> Bool
var F : Field
vars P1 P2 P3 : Agent
var K : Key
var R : Rand
eq inv1(F,P1,P2,P3,K,R) = ((not(K = k(enemy)) and rm(P1,P2,P3,enc(K,R,
   P2)) in nw(F)) implies not(P2 = enemy)) .
eq inv2(F,K,R) = (enc(K,R,enemy) in ciphers(nw(F)) implies (K = k(enem
   y))) .
```

The reason for rewriting the proof script in this way is that when the proof script is created using CiMPG, the properties and supplementary issues that we want to prove the need to be included in the module in which the protocol specification is written. Then, we rewrite the proof score as follows.

```
open IFF .
:id(iff)
op a1 : -> Agent .
op a2 : -> Agent .
op a3 : -> Agent .
op f : -> Field .
op k : -> Key .
op r : -> Rand .
red inv1(init,a1,a2,a3,k,r) .
close
open IFF .
:id(iff)
op a1 : -> Agent .
op a2 : -> Agent .
op a3 : -> Agent .
op f : -> Field .
op k : -> Key .
op r : -> Rand .
op r1 : -> Agent .
op r2 : -> Agent .
op r3 : -> Rand .
eq (r3 in rands(nw(f))) = true .
red inv1(f,a1,a2,a3,k,r) implies inv1(fkcm1(f,r1,r2,r3),a1,a2,a3,k,r) .
close
open IFF .
:id(iff)
op a1 : -> Agent .
op a2 : -> Agent .
op a3 : -> Agent .
op f : -> Field .
op k : -> Key .
op r : -> Rand .
op r1 : -> Agent .
op r2 : -> Agent .
op r3 : -> Rand .
eq (r3 in rands(nw(f))) = false .
red inv1(f,a1,a2,a3,k,r) implies inv1(fkcm1(f,r1,r2,r3),a1,a2,a3,k,r) .
close
open IFF .
```

```
:id(iff)
op a1 : -> Agent .
op a2 : -> Agent .
op a3 : -> Agent .
op f : -> Field .
op k : -> Key .
op r : -> Rand .
op r1 : -> Agent .
op r2 : -> Agent .
op r3 : -> Cipher .
eq r3 in ciphers(nw(f)) = true .
eq a1 = enemy .
eq r1 = a2 .
eq r2 = a3 .
eq k(r3) = k .
eq r(r3) = r .
eq p(r3) = a2 .
eq k = k(enemy) .
red inv1(f,a1,a2,a3,k,r) implies inv1(fkrm1(f,r1,r2,r3),a1,a2,a3,k,r) .
close
open IFF .
:id(iff)
op a1 : -> Agent .
op a2 : -> Agent .
op a3 : -> Agent .
op f : -> Field .
op k : -> Key .
op r : -> Rand .
op r1 : -> Agent .
op r2 : -> Agent .
op r3 : -> Cipher .
eq r3 in ciphers(nw(f)) = true .
eq a1 = enemy .
eq r1 = a2 .
eq r2 = a3 .
eq k(r3) = k .
eq r(r3) = r .
eq p(r3) = a2 .
eq (k = k(enemy)) = false .
eq a2 = enemy .
eq enc(k,r,enemy) in ciphers(nw(f)) = true .
```

```
red inv2(f,k,r) implies inv1(f,a1,a2,a3,k,r) implies inv1(fkrm1(f,r1,r2,r3),a1
    ,a2,a3,k,r) .
close
···
open IFF .
:proof(iff)
close
```

Since we rewrote the INV module inside the IFF module, it is the IFF module that is opened each time. Also, since we cannot use the SUCCESSOR state as in the original proof score, we have eliminated f'.

It is also necessary to rewrite the case equation. For example, in the IFF authentication protocol, the following equation is used in the proof score

```
eq r1 in rands(nw(f)) = true .
eq c-fkcm1(f,q1,q2,r1) = false .
```

This is because CafeOBJ has a high degree of freedom. However, CiMPG and CiMPA will cause an error if this is not done. Therefore, we need to rewrite it as follows.

```
eq (r3 in rands(nw(f))) = true .
eq (r3 in rands(nw(f))) = false .
```

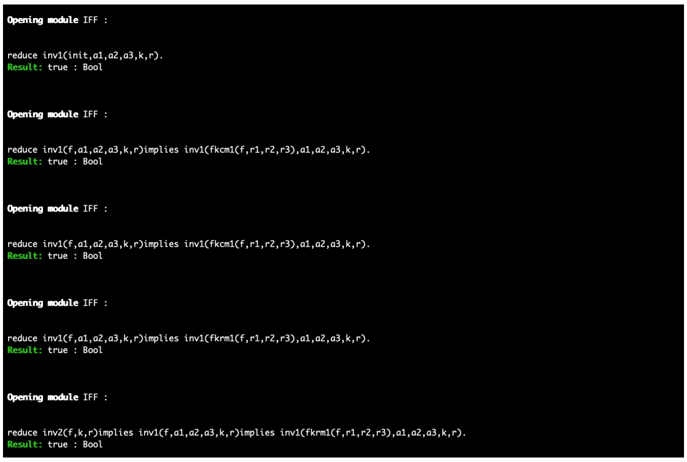
Thus, the left expression of the equation used for case separation must be the same. Then, every time we open a module, we execute the :id(iff) command, and finally the :proof(iff) command. These two commands are used to generate the proof script. After rewriting the proof as described above and reading it into CiMPG, the proof is correct and the proof script is returned.

5.2 Execution results of the IFF authentication protocol using CiMPG and CiMPA

Rewriting the above and reading it into CiMPG returns the result that the proof is correct, as shown below.

In addition, part of the proof script generated by CiMPG is as follows.

```
open IFF .
:goal{
        eq [iff :nonexec] :
        inv1(F:Field,A:Agent,A1:Agent,A0:Agent,K:Key,R:Rand) = true .
        }
:ind on (F:Field)
:apply(si)
```



```
Opening module IFF :


reduce inv1(init,a1,a2,a3,k,r).
Result: true : Bool



Opening module IFF :


reduce inv1(f,a1,a2,a3,k,r)implies inv1(fkcm1(f,r1,r2,r3),a1,a2,a3,k,r).
Result: true : Bool



Opening module IFF :


reduce inv1(f,a1,a2,a3,k,r)implies inv1(fkcm1(f,r1,r2,r3),a1,a2,a3,k,r).
Result: true : Bool



Opening module IFF :


reduce inv1(f,a1,a2,a3,k,r)implies inv1(fkrm1(f,r1,r2,r3),a1,a2,a3,k,r).
Result: true : Bool



Opening module IFF :


reduce inv2(f,k,r)implies inv1(f,a1,a2,a3,k,r)implies inv1(fkrm1(f,r1,r2,r3),a1,a2,a3,k,r).
Result: true : Bool
```

Figure 5.1: Some of the execution results

```
:apply(tc)
:def csb1 = :ctf [RRand in rands(nw(FField)) .]
:apply(csb1)
:imp [iff] by A0:Agent <- A0@Agent ; A1:Agent <- A1@Agent ; A:Agent
        <- A@Agent ; K:Key <- K@Key ; R:Rand <- R@Rand ;
:apply (rd)
:imp [iff] by A0:Agent <- A0@Agent ; A1:Agent <- A1@Agent ; A:Agent
        <- A@Agent ; K:Key <- K@Key ; R:Rand <- R@Rand ;
:apply (rd)
:apply(tc)
:def csb2 = :ctf [CCipher in ciphers(nw(FField)) .]
:apply(csb2)
:def csb3 = :ctf eq A@Agent = enemy .
:apply(csb3)
:def csb4 = :ctf eq AAgent = A1@Agent .
:apply(csb4)
:def csb5 = :ctf eq A0Agent = A0@Agent .
:apply(csb5)
:def csb6 = :ctf eq k(CCipher) = K@Key .
:apply(csb6)
:def csb7 = :ctf eq r(CCipher) = R@Rand .
:apply(csb7)
:def csb8 = :ctf eq p(CCipher) = A1@Agent .
:apply(csb8)
:def csb9 = :ctf eq K@Key = k(enemy).
:apply(csb9)
```

```
:imp [iff] by A0:Agent <- A0@Agent ; A1:Agent <- A1@Agent ; A:Agent
      <- A@Agent ; K:Key <- K@Key ; R:Rand <- R@Rand ;
:apply (rd)
:apply(tc)
···
close
```

The same result can be obtained for the complement used to prove the properties of the IFF authentication protocol by rewriting it in the same way.

5.3 Summary of IFF authentication protocol using CiMPG and CiMPA

In Chapter 3, we proved the IFF authentication protocol using the proof score method, and in Chapter 5, we proved it using the method of generating proof scripts using CiMPG and CiMPA. In the proof score method, there is a possibility of making mistakes in the number of cases and methods, and using CiMPG and CiMPA can reduce this possibility. However, to use CiMPG and CiMPA, it is necessary to rewrite the specification and proof score, which is difficult if you are not familiar with it. Each of these has its advantages and disadvantages, as we found out in this study.

Chapter 6

Lessons Learned

This Chapter gives describes what we learned through the research project.

6.1 Security

Through this research project, We were able to learn about security, which We had never thought about in-depth before. We were able to learn what authentication is for communication on a network where security is important, and what threats exist in communication. By learning these things, We were able to deepen my understanding of what secure communication is all about. By using CafeOBJ, a programming language that We had never touched before, We were able to deepen my understanding of the advantages and disadvantages of CafeOBJ, how to use it, how to write specifications, and how to write proof scores. We were able to deepen my understanding of the advantages and disadvantages of CafeOBJ, how to use it, how to write specifications, and how to write proof scores.

6.2 Proof Scores and Proof Scripts

CiMPG converts proof scores into proof scripts for CiMPA, but in this investigation, we found out that not any proof score can be converted. Even if a proof score is created and the result is correct, the proof script will not be generated correctly, using the following separation of cases.
sub case1 nw(f) = m1, nw
sub case2 m1 in nw(f) = false

In order to have CiMPG generate the proof script correctly, the following case study must be done.
sub case1 nw(f) = m1, nw

sub case2 (nw(f) = m1, nw) = false

In addition, there is a subtle difference between the results of formal verification using the same proof score with CafeOBJ and with CafeInMaude. CafeInMaude may return false even if CafeOBJ returns true. This is due to the fact that CafeOBJ uses the equality rewrite rule (equals are considered as arrows, and left and right are clearly distinguished). Based on these facts, we believe that rewriting to CafeInMaude can be done smoothly by paying attention to the case separation method and rewriting rules, not using the SUCCESSOR state, and creating a proof score. Thus, we understand that there are advantages and disadvantages to CafeOBJ and CafeInMaude CiMPG and CiMPA.

Chapter 7

Conclusion

This Chapter gives summarizes the report and gives some pieces of our future work.

7.1 Summary of the report

With the rapid spread and development of the Internet, security protocols that guarantee safe and secure communication on the Internet are becoming more and more important. However, it is not uncommon for serious attacks to exist even in security protocols that have been carefully designed by security experts, due to misunderstandings in the operating environment or the objectives they are trying to achieve. Furthermore, such flaws are difficult to detect in normal operations or in traditional software testing. For this reason, techniques for formally verifying the correctness of security protocols have been studied and many methods have been proposed.

Against this backdrop, we undertook this research project with the aim of acquiring techniques to reduce the number of authentication protocol failures, which will enable us to contribute to safer and more secure shopping on e-commerce sites and safer and more secure communications on the Internet.

In Chapter 2, we deepened our understanding of authentication, authentication protocols and CafeOBJ using simple examples such as QLOCK.

In Chapter 3, we used the IFF authentication protocol to deepen our understanding of authentication protocols and CafeOBJ.

In Chapter 4, the NSLPK authentication protocol, which is more complex than the IFF authentication protocol, is used to further deepen the understanding of the authentication protocol and CafeOBJ.

In Chapter 5, the IFF authentication protocol is verified with CiMPG and CiMPA as a new case study.

In Chapter 6, we summarize what we learned in this research project.

7.2 Future prospects

One of the future prospects is the formal verification of the NSLPK authentication protocol using CiMPG and CiMPA.Formal verification of the NSLPK authentication protocol using CiMPG and CiMPA has not yet been done, and we believe that once completed, we will be able to make more contributions to the technology of security protocols.

Another suggestion is an easier way to rewrite CafeOBJ to CafeInMaude. If the rewriting from CafeOBJ to CafeInMaude can be made easier, more protocols can be formally verified using CiMPG and CiMPA, and more contributions to the technology of security protocols can be made. If the rewriting from CafeOBJ to CafeInMaude can be made easier, more protocols can be formally verified using CiMPG and CiMPA, and more contributions to security protocol technology will be possible.

Bibliography

[1] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, Vol. 21, No. 12, pp. 993–999, 1978.

[2] Gavin Lowe. An attack on the needham-schroeder public-key authentication protocol. *Inf. Process. Lett.*, Vol. 56, No. 3, pp. 131–133, 1995.

[3] Gavin Lowe. Breaking and fixing the needham-schroeder public-key protocol using FDR. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop, TACAS '96, Passau, Germany, March 27-29, 1996, Proceedings*, Vol. 1055 of *Lecture Notes in Computer Science*, pp. 147–166. Springer, 1996.

[4] Razvan Diaconescu and Kokichi Futatsugi. *Cafeobj Report - The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, Vol. 6 of *AMAST Series in Computing.* World Scientific, 1998.

[5] Kazuhiro Ogata and Kokichi Futatsugi. Proof scores in the ots/cafeobj method. In Elie Najm, Uwe Nestmann, and Perdita Stevens, editors, *Formal Methods for Open Object-Based Distributed Systems, 6th IFIP WG 6.1 International Conference, FMOODS 2003, Paris, France, November 19.21, 2003, Proceedings*, Vol. 2884 of *Lecture Notes in Computer Science*, pp. 170–184. Springer, 2003.

[6] Kazuhiro Ogata and Kokichi Futatsugi. Some tips on writing proof scores in the ots/cafeobj method. In Kokichi Futatsugi, Jean-Pierre Jouannaud, and José Meseguer, editors, *Algebra, Meaning, and Computation, Essays Dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*, Vol. 4060 of *Lecture Notes in Computer Science*, pp. 596–615. Springer, 2006.

[7] Kazuhiro Ogata and Kokichi Futatsugi. Proof score approach to analysis of electronic commerce protocols. *Int. J. Softw. Eng. Knowl. Eng.*, Vol. 20, No. 2, pp. 253–287, 2010.

[8] Kazuhiro Ogata and Kokichi Futatsugi. Equational approach to formal analysis of TLS. In *25th International Conference on Distributed Computing Systems (ICDCS 2005), 6-10 June 2005, Columbus, OH, USA*, pp. 795–804. IEEE Computer Society, 2005.

[9] Kazuhiro Ogata and Kokichi Futatsugi. Rewriting-based verification of authentication protocols. *Electron. Notes Theor. Comput. Sci.*, Vol. 71, pp. 208–222, 2002.

[10] Adrián Riesco, Kazuhiro Ogata, and Kokichi Futatsugi. Cafeinmaude: A cafeobj interpreter in maude. In Perdita Stevens and Andrzej Wasowski, editors, *Fundamental Approaches to Software Engineering - 19th International Conference, FASE 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, Vol. 9633 of *Lecture Notes in Computer Science*, pp. 377–380. Springer, 2016.

[11] Adrián Riesco, Kazuhiro Ogata, and Kokichi Futatsugi. A maude environment for cafeobj. *Formal Aspects Comput.*, Vol. 29, No. 2, pp. 309–334, 2017.

[12] Adrián Riesco and Kazuhiro Ogata. A formal proof generator from semi-formal proof documents. In Dang Van Hung and Deepak Kapur, editors, *Theoretical Aspects of Computing - ICTAC 2017 - 14th International Colloquium, Hanoi, Vietnam, October 23-27, 2017, Proceedings*, Vol. 10580 of *Lecture Notes in Computer Science*, pp. 3–12. Springer, 2017.

[13] Adrián Riesco and Kazuhiro Ogata. Prove it! inferring formal proof scripts from cafeobj proof scores. *ACM Trans. Softw. Eng. Methodol.*, Vol. 27, No. 2, pp. 6:1–6:32, 2018.

[14] Ross J. Anderson. *Security engineering - a guide to building dependable distributed systems.* Wiley, 2001.