Master's Thesis

Graphical animations of authentication protocols

Thet Wai Mon

Supervisor: Professor Kazuhiro Ogata

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

August 2021

Abstract

Authentication protocols have become important technical components in this interconnected world. Although these authentication protocols have been carefully designed by security experts, it is not uncommon for security attacks as technology continues to advance. If authentication protocols have some flaws (security holes), confidential information such as monetary transactions may be leaked to malicious third parties resulting in numerous security breaches that have caused extensive financial damage to both individuals and corporations. Ensuring the reliability of authentication protocols is then really important. Therefore, we need to use some technologies for this purpose. Many approaches have been proposed and used to guarantee that authentication protocols are truly reliable. One possible technology is formal verification with theorem proving in which one challenging task is lemma conjecture.

This thesis uses state machine graphical animation tool (SMGA). The main purpose of SMGA is to help human users be able to visually perceive non-trivial characteristics of the protocol by observing its graphical animations. SMGA takes a finite state sequence of a state machine formalizing a protocol and a state picture designed for the protocol and produces its graphical animations by regarding the state sequence as a movie film. Observing such graphical animations allows us to guess the characteristics of the state machine formalized the protocol. We confirm whether the state machine enjoys guessed characteristics because such characteristics may or may not be true properties of the protocol.

One possible way to do so is model checking. This thesis uses Maude as a formal specification language to formally specify the authentication protocols as the state machines and to confirm the guessed characteristics by using Maude model checking facilities. We use Maude search command to confirm the guessed characteristics. However, it does not guarantee that the state machine enjoys the properties because standard model checking techniques cannot handle state machines in which there are an arbitrary number of entities, such as principle. To make sure that the guessed characteristics are invariant properties of the protocol, we conduct the formal verification with CafeInMaude Proof Assistant (CiMPA) and CafeInMaude Proof Generator (CiMPG) provided by CafeInMaude, the world's second implementation of CafeOBJ in Maude, where CafeOBJ is another formal specification language and a sister language of Maude. Thus, the two protocols are also formally

specified in CafeOBJ as observational transition systems (OTSs), a kind of state machines.

The behaviors of two authentication protocols Identify-Friend-or-Foe protocol (IFF) and Needham-Schroeder-Lowe (NSLPK) are graphically visualized using SMGA so that human users can visually perceive non-trivial characteristics of the protocols by observing graphical their animations based on the state picture designs. These characteristics could be used as lemmas to formally verify that the protocol enjoys desired properties. Firstly, state picture designs are carefully made for authentication protocols to produce good graphical animations with SMGA and then find out non-trivial characteristics of the protocol by observing its graphical animations generated from SMGA. We also confirm the correctness of the guessed characteristics using model checking with Maude search command and then the formal verification of the protocols is conducted with CiMPG and CiMPA. We have confirmed that all lemmas used in the formal verification can be discovered by observing graphical animations.

This thesis mainly demonstrates that SMGA has potential to help human users conjecture lemmas needed to conduct formal proofs and how SMGA and CiMPG (and CiMPA) together can be used to conduct the formal verification.

Keywords: graphical animation, SMGA, IFF, NSLPK, state machine, state picture design, CafeInMaude, CiMPA, CiMPG

Acknowledgment

First and foremost I would like to express my special thanks of gratitude to my supervisor Professor Kazuhiro Ogata for giving me the opportunity to do research and providing his invaluable advice, continuous support, and patience during my study. His immense knowledge, sincerity, and motivation have deeply inspired me. Without his support and kindly instruction, it would not have been possible to complete my master's program. It was a great privilege and honor to work and study under his guidance. I am extremely grateful for what he has offered me.

Secondly, I would like to offer my sincere thanks to all members of our lab, especially Mr. Bui Duy Dang and Mr. Tran Dinh Duong for helping me in research. It is their kind help and support that have made my study and daily life in JAIST a wonderful time.

I would also like to extend my special thanks to my Burmese friends at JAIST. My life in Japan was less boring because of them and I have received much help from them since the first day I came to Japan. I owe a debt of gratitude to Ms. Aye Myat Mon and Mr. Ashmaoy Amr for taking care of me when I was not in good health.

Moreover, I am deeply grateful to my friends in Myanmar for keeping in touch with me, giving advice, and their encouragement. Whenever I am depressed, they encourage me to continue trying and they make my stress go away. Thank you all for being with me for many years.

Finally, I would like to express my deep gratitude to my family. No word can be used to express my sincere thanks to them. I am deeply indebted to my parents for their love, caring, sacrifices for educating me and supporting me spiritually throughout my life. Without their tremendous understanding and encouragement, it would be impossible for me to be here.

Thank you so much!

Thet Wai Mon
June 19,2021

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1.1 Motivation

With the widespread use of the Internet and the rapid development of technology, people are using IoT devices that connect to the Internet allowing us to either control or send/receive data from our smartphones or computer and making an online transaction in e-commerce applications. Thus, security protocols that guarantee secure and safe communication over the Internet are becoming more and more important. The authentication process also becomes the most important layer of protection needed for secure communication within computer networks. Authentication is the process of verifying the identity of the things such as an object, a computer, etc. Protocols are a set of rules, data structures that describe how devices to exchange data across networks. They can be considered as languages that two devices must understand seamlessly for information communication, regardless of their infrastructure and design differences.

Authentication protocols are computer communication protocols or cryptography protocols designed for the transfer of authentication data to achieve authentication between two entities. They have become important technical components in this interconnected world. Although these authentication protocols have been carefully designed by security experts, security attacks may happen since technology continues to advance. If authentication protocols have some flaws (security holes), confidential information such as monetary transactions may be leaked to malicious third parties resulting in numerous security breaches that have caused extensive financial damage to both individuals and corporations. Ensuring the reliability of authentication protocols is then really important. Therefore, we need to use some technologies for this purpose. Many technologies have been proposed and used to guarantee that authentication protocols are truly reliable. One of them is formal verification. This approach uses the state machine to formalize the target system as a mathematical model, the requirements that are supposed to be satisfied by the system can be modeled as the properties of its formalized state machine.

System verification then can be conducted as formal verification of state machine properties.

There are two major techniques in formal verification namely model checking and theorem proving. Model checking can be automatically conducted but it cannot guarantee that the state machine enjoys the properties when the reachable state space of the system is infinite due to the state explosion problem. If that is the case, we should use theorem proving to formally verify that the system enjoys the guessed properties. However, human interaction is often required to conjecture some auxiliary lemmas. This task is called lemma conjecture which can be considered as one of the most intellectual challenging tasks in theorem proving.

We aim to come up with a better way to conjecture lemmas in much fewer efforts and less time by observing animations produced by the State Machine Graphical Animation tool (SMGA) [1] to complete formal proof. Several case studies of some protocols have been tackled with SMGA such as some mutual exclusion protocols Qlock [2], MCS [3, 4], a distributed mutual exclusion protocol [5], and a communication protocol ABP (Alternating Bit Protocol) [1]. Authentication protocols have not been yet, and this work is the very first time of tackling authentication protocols with SMGA.

Authentication protocols are often presented as message sequences between two principals called Alice-Bob format. It is possible to grasp the messages exchanged by Alice and Bob and the order of the messages when any intruders, such as Cathy, are never involved. We need to take into account the existence of intruders so as to formally verify that authentication protocols enjoy desired properties. Cathy plays an ordinary principal from the Alice and Bob point of view but does something against authentication protocols, such as faking messages based on the gleaned information. Thus, many messages may be faked by Cathy. If many messages appear, it may not be straightforward to comprehend message sequences. This is why we did not adapt message sequences and came up with a different visualization of authentication protocols.

1.2 The overview of method

State machine graphical animation tool (SMGA) is used in this thesis to graphically visualize the protocol. SMGA has been developed to visualize graphical animations of protocols. The main purpose of SMGA is to help human users be able to visually perceive non-trivial characteristics of the protocol by observing its graphical animations because humans are good at visual perception [6]. An input requires a state picture designed by humans

and a finite state sequence. An output is graphical animations based on series of pictures in which each state is displayed by the state picture design. Observing such graphical animations allows us to guess the characteristics of the state machine of the protocol. These guessed characteristics are required to be confirmed whether they are true properties of the state machine. For this purpose, model checking, the most well-known technique in formal method is used in this thesis. Model checking, however, does not guarantee that the state machine enjoys the properties when the reachable state space of the system is unbounded. If that is the case, some other techniques such as theorem proving can be used to make sure that the system enjoys the guessed properties.

1.3 Thesis organization

The remaining part of this thesis is organized as follows:

- **Chapter 2 - Preliminaries** gives some common notions and background knowledge that related to state machine, SMGA, Maude, CafeInMaude, CiMPA and CiMPG.

- **Chapter 3 - IFF protocol** describes IFF protocol and its formal specification.

- **Chapter 4 - NSLPK protocol** describes NSLPK protocol and its formal specification.

- **Chapter 5 - Graphical animation of IFF protocol** presents the state picture design of IFF in which the idea and the design are conveyed. This chapter also reports on how to find characteristics by observing graphical animations and guessed characteristics are confirmed by model checking.

- **Chapter 6 - Graphical animation of NSLPK protocol** presents the state picture design of NSLPK in which the idea and the design are conveyed. This chapter also reports on how to find characteristics by observing graphical animations and guessed characteristics are confirmed by model checking.

- **Chapter 7 - Formal verification with CiMPA and CiMPG** describes formal verification of two authentication protocols (IFF &

NSLPK) with CiMPA and CiMPG.

- **Chapter 8 - Lessons learned** gives the lessons what we have learned through this thesis.
- **Chapter 9 - Related work** mentions some related work.
- **Chapter 10 - Conclusion** summarizes this thesis with some pieces of future work.

All files such as specifications of the protocols (in Maude/CafeOBJ), state pictures, and input files for SMGA used in this thesis are available at https://github.com/twmon14/ms21.

Chapter 2

Preliminaries

This chapter gives some common notions and background which are requirements for this thesis. We first present the definition of a state machine in Section 2.1. Then we show the basic syntax of Maude through a simple example in Section 2.2. In Section 2.3, we describe SMGA with another example of state picture design. Finally, the description of CafeInMaude, CiMPA, and CiMPG are given in Section 2.4.

2.1 State machine

A state machine is a mathematical model of computation. Based on the current state and given input state machine performs state transitions and produces outputs. A state machine $M \triangleq \langle S, I, T \rangle$ consists of

- a set S of states,
- a set $I \subseteq S$ of initial states, and
- a binary relation $T \subseteq S \times S$ over states.

$(s, s') \in T$ is called a state transition where s' is successor state of s and may be written as $s \rightarrow_M s'$.
The set $R \subseteq S$ of reachable states with respect to (wrt) M is inductively defined as follows:

1. $I \subseteq R$ and
2. if $s \in R$ and $s \rightarrow_M s'$, then $s' \in R$.

A state predicate p is an invariant property wrt M if and only if $(\forall s \in R)$ $p(s)$ that is $p(s)$ holds for all $s \in R$. A state predicate p can be interpreted as a set P of states such that $(\forall s \in P)$ $p(s)$ and $(\forall s \notin P)$ $\neg p(s)$. A finite sequence $s_0, \ldots, s_i, s_{i+1}, \ldots, s_n$ of states is called a finite computation of M if $s_0 \in I$ and $(s_i, s_{i+1}) \in T$ for each $i = 0, \ldots, n-1$. States are expressed as braced soups of observable components. Soups are associative-commutative collections, and observable components are name-value pairs. That is a state of S is expressed as associative-commutative collection of name-value pairs. The juxtaposition operator is used as the constructor of

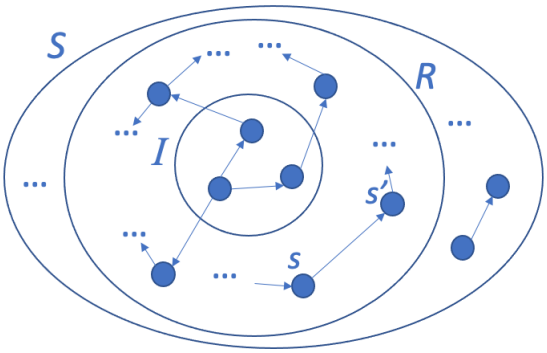


Figure 2.1: State machine & Invariant

soups. Let $oc1, oc2, oc3$ be observable components, and $oc1$ $oc2$ $oc3$ is the soup of observable components. Then a state that consists of these three observable components can be expressed as $\{oc1\ oc2\ oc3\}$, which equals $\{oc3\ oc1\ oc2\}$ and some others due to associativity and commutativity. To specify state transitions we use Maude as a formal specification language.

2.2 Maude

Maude [7] is a rewriting logic-based specification language supporting both equational and rewriting logic. Maude makes it possible to use associative-commutative collections and state transitions are specified as rewrite rules. A rewrite rule starts with the keyword `rl`, followed by a label enclosed with a square bracket and a colon, two patterns (terms that may contain variables) connected with =>, and ends with a full stop. A conditional rule starts with the keyword `crl` and has a condition with the keyword `if` before the full stop. The following are forms of a rewrite rule and conditional rewrite rule:

`rl` $[lb] : l \Rightarrow r$.

where lb is a label and l is replaced with r.

`crl` $[lb] : l \Rightarrow r$ `if` $\ldots$ `/\` c_i `/\` $\ldots$.

where lb is a label and c_i is part of the condition, which may be an equation $lc_i = rc_i$. The negation of $lc_i = rc_i$ can be written as $(lc_i =/= rc_i) =$ `true`, where `= true` can be omitted. If the condition $\ldots$ `/\` c_i `/\` $\ldots$ holds under some substitution σ, $\sigma(l)$ can be replaced with $\sigma(r)$.

Maude is equipped with model checking facilities (a reachability analyzer and an LTL model checker). Maude provides the search command that allows finding a state reachable from s such that the state matches pattern p and

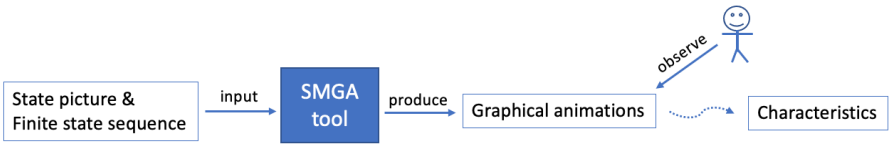
satisfies condition c:

`search [n,m] in MOD :` s `=>*` p `such that` c .

where `MOD` is the name of the Maude module specifying the state machine under model checking, `n` and `m` are optional arguments stating a bound on the number of desired solutions and the maximum depth of the search, respectively. `n` typically is 1 and s is a given state (typically an initial state of the state machine). p is a pattern and c is a condition. The condition part `such that` c can be omitted. The search command searches the reachable states from s for at most n states that can match the pattern p and make the condition c true. In this thesis, Maude search command is used to confirm the characteristics guessed by observing graphical animations of each protocol.

2.3 State Machine Graphical Animation

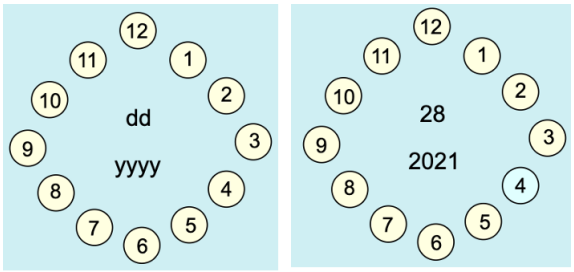
State machine graphical animation tool (SMGA) is developed by Nguyen and Ogata [1]. The main purpose of SMGA is to help human users be able to recognize state patterns and perceive non-trivial characteristics of the protocol by observing its graphical animations. SMGA cannot automatically produce visual state picture design however it allows us to design a good state picture. Designing a state picture affects how well human users can recognize non-trivial characteristics of protocols [4]. Thus we need to carefully design state pictures. As an input, SMGA takes a state picture designed by humans and a finite state sequence generated by Maude. An output is graphical animations of series of pictures in which each state is displayed by the state picture design.



There are two ways to visualize the observable components at each state. They are (1) text display and (2) visual display. For example, one state that simulates calendar is specified as follows:

```
(dd: 28) (mm: 4) (yy: 2021)
```

where `dd`, `mm`, `yy` are observable components, `28`, `4`, `2021` are their values, respectively. The following figures display state picture design (on the left-hand side), and a state picture (on the right-hand side) of the example in which `mm` is displayed as (2), `dd` and `yy` are displayed as (1).



2.4 CafeInMaude, CiMPA and CiMPG

Writing proof scores in theorem proving to verify that systems enjoy some desired properties have been intensively used. Observational transition systems (OTSs) [8] are used to formalized systems as state machines in this formal verification to proof score approach. OTSs are specified in CafeOBJ [9] , a formal specification language in which formal verification is conducted by writing what is called "proof scores" [8] and executing them. CafeOBJ and Maude are sibling languages of OBJ family. Taking into account the similarities between them, CafeOBJ interpreter in Maude (CafeInMaude) [10] is implemented by combining CafeOBJ specification and theorem proving capabilities with efficient and extensible Maude commands and tools. CafeInMaude introduces CafeOBJ specifications into Maude system and hence, proof scores written in CafeOBJ can be loaded and executed in CafeInMaude.

CafeInMaude comes with two extension tools, CiMPA (CafeInMaude Proof Assistant) and CiMPG (CafeInMaude Proof Generator) [11]. The former is a proof assistant that allows users to write the proof scripts manually to prove invariant properties based on their CafeOBJ specifications. The manually written proof scripts are executed with CiMPA, if CiMPA can discharge all goals successfully, then the proofs are correct and the properties are proved; otherwise, we need to revise the proof scripts and make the verification attempt again until it is successful. However, writing proof scripts is not easy for non-expert users whereas working with proof scores is easier and flexible. The disadvantage of working with proof scores is it is subject to human errors because users may overlook some cases and it may lead to incorrect proofs. CiMPA can address this issue but the weakness is the lack of flexibility.

To overcome this weakness, CiMPG can be used. CiMPG allows users to combine the flexibility of the proof score approach with the reliability of CiMPA. In this thesis, we use two authentication protocols to confirm the effectiveness of CiMPG. Once we have a complete proof score, it is preferable

to use CiMPG to automatically generate proof scripts for CiMPA, however, we need to annotate the proof score. If CiMPA cannot discharge all goals successfully, the proof scores are something wrong and we need to revise the proof scores, making verification attempts again. The process of formal verification with CiMPA and CiMPG is summarized in the following figure.

Chapter 3

IFF protocol

This chapter describes the IFF protocol in Section 3.1. The formal specification of IFF written in Maude is presented in Section 3.2.

3.1 Identity-Friend-or-Foe protocol

Identity-Friend-or-Foe (IFF) [12] protocol checks if a principal is a member of a group. The IFF protocol can be described as the following two message exchanges:

Check $p \to q : r$

Reply $q \to p : \varepsilon_k(r, q)$

Each principal (or agent) such as p and q belongs to only one group. A symmetric key is given to each group whose members share the symmetric key. Symmetric keys are different from group to group. If a principal p wants to check if a principal q is a member of the p's group, p generates a fresh random number r and sends it to q as a Check message. On receipt of the message, q sends back to p a Reply message that consists of r and ID q encrypted by the symmetric key k of the q's group. When p receives the Reply message, p tries to decrypt the ciphertext of the message with the symmetric key of the p's group. If the decryption is successful and the plaintext consists of r and q, p then concludes that q is a member of the p's group. The protocol is supposed to have the property that if p receives a valid Reply message from q, q is always a member of the p's group. The property is called the Identifiable property.

We suppose that the cryptosystem used is perfect, there is only one legitimate group in which all members are trustable, and there are also untrustable principals who are not members of the group. Trustable principals exactly follow the protocol, but untrustable ones may do something against the protocol as well. The combination and cooperation of untrustable principals are modeled as the most general enemy or intruder (which is treated as the Dolev-Yao most generic intruder). The enemy gleans as much information as possible from messages flowing in the network and fakes messages based

on gleaned information, provided that the enemy cannot break the perfect cryptosystem.

3.2 Formal specification of IFF in Maude

The ciphertext for Check message is defined by the following operator in IFF protocol.

```
op enc : Prin Rand Prin -> Cipher .
```

where `Prin` is the sort denoting principals; first parameter represents the symmetric key of the principal's group; `Rand` is the sort denoting the random number; given principal q and a random number r term $\texttt{enc}(q, r, q)$ denotes the ciphertext obtained by encrypting r and q with the symmetric key of the q's group.

Two kinds of messages used in the IFF protocol are specified as follows:

```
op cm : Prin Prin Prin Rand -> Msg .
op rm : Prin Prin Prin Cipher -> Msg .
```

where `Msg` is the sort denoting messages. The first, second, and third arguments of each of `cm`, `rm` operators are the actual creator, the seeming sender, and the receiver of the corresponding message. The first argument is meta-information that is only available to the outside observer and the agent that has sent the corresponding message, and that cannot be forged by the intruder; while the remaining arguments may be forged by the intruder. The network is modeled as a multiset of messages, which the intruder can use as his/her storage. Any message that has been sent into the network is supposed to be never deleted from the network because the intruder can replay the message repeatedly, although the intruder cannot forge the first argument.

A state of the protocol is expressed as a soup of observable components. Let ms, rs, and ps be the collections of messages, random numbers, and principals, respectively. ps may contain an intruder. Let *cipher* be the collection of `Cipher`. We use the following observable components to formalize the IFF protocol as a state machine M_{IFF}:

- (`nw` : ms) - It says that the network is constructed by ms.
- (`cipher` : cs) - It says that the collection of `Cipher` ciphertexts gleaned by the intruder is cs.
- (`urand` : urs) - It says that the collection of random numbers gleaned by the intruder is urs.

- (`prins` : ps) - It says that the principals participating in the protocol are ps.
- (`rand` : rs) - It says that the available random numbers are rs. Whenever a principal wants to send a Check message, it needs to generate a random and globally unique number. To formalize that behavior, we provide a fixed random number from the beginning, and every time a process needs to generate a random number, an element is extracted and used.

Each state in S_{IFF} is expressed as $\{obs\}$, where obs is a soup of observable components. We suppose that two principals `p` & `q` together with an intruder participate in the IFF protocol, one initial state of I_{IFF} namely `init` is defined as follows: where `intr` is a constant of sort `Prin` denoting the intruder, and `emp` denotes an empty collection.

```
{(nw: emp) (rand: r) (urand: emp)
(cipher: emp) (prins: (p q intr))} .
```

Transition functions are defined as rewrite rules in Maude. The first two transitions formalize sending messages exactly following the protocol. These two rewrite rules `sdcm` and `sdrm` are defined respectively. Let `OCs` be a Maude variable of observable component soups; `P` & `Q` be Maude variables of principals; `Ps` be a Maude variable of collections of principals; `NW`, `R` be Maude variables denoting a network and a random number respectively; `Rs`, `CE` be Maude variables denoting a collection of random numbers and `Cipher` respectively. The first rewrite rule `sdcm` is defined as follows:

```
rl [sdcm] : {(nw: NW) (prins: (P Q Ps))
(rand: R) (cipher: CE) (urand: Rs) OCs }
=> {(nw: (cm: cm(P,P,Q,R) NW)) (cipher: CE)
(urand: (R Rs)) (rand: emp) (prins: (P Q Ps)) OCs} .
```

The rewrite rule says that when `R` is in `rand`, a new Check message is put into the network, intruder gleans information such as random number and ciphertext used in that message and `R` is removed from `rand`.

The second rewrite rule `sdrm` is defined as follows:

```
rl [sdrm] : {(nw: (cm(P',P,Q,R) NW))
(prins: (P Q Ps)) (cipher: CE) (urand: Rs) OCs}
=> {(nw: (rm(Q,Q,P,enc(Q,R,Q)) cm(P',P,Q,R) NW))
(cipher: (enc(Q,R,Q) CE)) (urand: (R Rs)) (prins: (P Q Ps)) OCs} .
```

where `P'` is a Maude variable denoting a principal of `Prin` sort. The rewrite rule says that when a new Reply message is put into the network on receipt

of the first message, intruder gleans the ciphertext used in that message and puts it into the collection of `cipher`.

In addition to two rewrite rules that formalize sending messages exactly following the protocol mentioned above, we also introduce three more rewrite rules to formalize the intruder's faking messages based on the gleaned information:

- `fakecm1` & `fakerm1`: a random number `R` is available to the intruder, the intruder fakes and sends an Check or an Reply message using `R`, respectively,
- `fakerm2`: a ciphertext `C` is available to the intruder, the intruder fakes and sends an Reply message using `C`.

The rewrite rule `fakecm1` is defined as follows:

```
rl [fakecm1] : {(nw: NW) (urand: (R Rs))
(prins: (P Q Ps)) (cipher: CE) OCs}
=> {(nw: (cm(intr,P,Q,R) NW)) (urand: (R Rs))
(prins: (P Q Ps)) (cipher: CE) OCs}} .
```

The rewrite rule says that when `R` is in `urand`, a new intruder's faking Check message is put into the network. The rewrite rule `fakerm1` is defined as follows:

```
rl [fakerm1] : {(nw: NW) (urand: (R Rs))
(prins: (P Q Ps)) (cipher: CE) OCs}
=> {(nw: (rm(intr,Q,P,enc(intr,R,Q)) NW)) (urand: (R Rs))
(prins: (P Q Ps)) (cipher: CE) OCs}} .
```

The rewrite rule says that when `R` is in `urand`, intruder fakes Reply message and puts it into the network. The last rewrite rule `fakerm2` is defined as follows:

```
rl [fakerm2] : {(nw: NW) (prins: (P Q Ps)) (cipher: (C CE)) OCs}
=> {(nw: (rm(intr,Q,P,C)) NW)) (prins: (P Q Ps))
(cipher: (C CE)) OCs} .
```

The rewrite rule says that when `C` is in `cipher`, intruder fakes Reply message and puts it into the network using gleaned ciphertext.

Chapter 4

NSLPK protocol

This chapter describes the NSLPK protocol in Section 4.1 and its formal specification written in Maude in Section 4.2.

4.1 Needham-Schroeder-Lowe Public Key protocol

Needham-Schroeder-Public-Key (NSPK) [13] authentication protocol has been often used as a benchmark in the formal analysis of security protocols. Gavin Lowe found a non-trivial attack on NSPK in which some secret information is leaked to a third party and proposed a revised version. A revised version of NSPK by Lowe is called NSLPK [14]. The messages exchanged in NSLPK are as follows:

Init $p \rightarrow q : \varepsilon_q(n_p, p)$
Resp $q \rightarrow p : \varepsilon_p(n_p, n_q, q)$
Ack $p \rightarrow q : \varepsilon_q(n_q)$

NSLPK uses public-key cryptography. Each principal such as p and q has a private/public key pair. The public key is shared with all principals but the private one is only available to its owner. $\varepsilon_p(m)$ denotes the ciphertext obtained by encrypting the message m with the principal p's public key. n_p is a nonce (a random number) generated by principal p. A nonce is a unique and non-guessable number that is used only one time in a protocol run.

If a principal p wants to authenticate a principal q, p generates a nonce n_p and sends to q an Init message that consists of n_p and ID p encrypted by the public key of q. When q receives the Init message, q tries to decrypt the ciphertext received by its private key. q then generates a nonce n_q and sends back to p a Resp message that consists of n_p, n_q, and ID q encrypted by the public key of p. On receipt of the Resp message, p tries to decrypt the ciphertext received by its private key. If the decryption is successful, p obtains two nonces and a principal ID and checks if the principal ID equals the sender of the message and one of the nonces is the exact one that p has

sent to the sender in this session. p then sends back to q an Ack message that contains n_q encrypted by the public key of q. On receipt of the message, q decrypts it, obtains a nonce, and checks if the nonce is the one that q has sent to the sender in this session.

We suppose that the cryptosystem used is perfect, there is only one legitimate group, all members of the group are trustable, and there are also untrustable principals who are not members. Trustable principals exactly follow the protocol, but untrustable ones may do something against the protocol as well. The combination and cooperation of untrustable principals are modeled as Dolev-Yao, the most general intruder. The intruder can glean as much information as possible from messages flowing in the network and create fake messages based on the gleaned information, provided that the intruder cannot break the perfect cryptosystem.

4.2 Formal specification of NSLPK in Maude

The following three operators represent three kinds of ciphertexts for three messages used in NSLPK protocol.

```
op enc1 : Prin Nonce Prin -> Cipher1 .
op enc2 : Prin Nonce Nonce Prin -> Cipher2 .
op enc3 : Prin Nonce -> Cipher3 .
```

where `Prin` is the sort denoting principals; `Nonce` is the sort denoting the nonce that consists of random numbers; `Cipher1`, `Cipher2`, and `Cipher3` are the sorts denoting three kinds of ciphertexts contained in Init, Resp, and Ack messages, respectively. Given principals p, q and a nonce n_p term $\texttt{enc1}(q, n_p, p)$ denote the ciphertext $\varepsilon_q(n_p, p)$ obtained by encrypting nonce n_p and ID p with the principal q's public key. Given principals p, q and nonces n_p, n_q term $\texttt{enc2}(p, n_p, n_q, q)$ denote the ciphertext $\varepsilon_p(n_p, n_q, q)$ obtained by encrypting nonce n_p, n_q and ID q with the principal p's public key. Given principals p, q and a nonce n_q term $\texttt{enc3}(q, n_q)$ denote the ciphertext $\varepsilon_q(n_q)$ obtained by encrypting nonce n_q with the principal q's public key. Hereinafter, let us use `Cipher1` (or `Cipher2`, or `Cipher3`) ciphertexts to refer to the ciphertexts sent in Init (or Resp, or Ack) messages.

A `Nonce` is defined by the following operator:

```
op n : Prin Prin Rand -> Nonce .
```

where the first and second arguments belong to sort `Prin` and the third argument `Rand` is the sort denoting random numbers that makes the nonce globally unique and unguessable. Given principals p, q and random value r,

a nonce is formalized as the term $\mathtt{n}(p, q, r)$ denoting a nonce generated by principal p to be sent to principal q for authenticating where r is a fresh random number.

Three kinds of messages used in the NSLPK protocol are specified as follows:

```
op m1 : Prin Prin Prin Cipher1 -> Msg
op m2 : Prin Prin Prin Cipher2 -> Msg
op m3 : Prin Prin Prin Cipher3 -> Msg
```

where `Msg` is the sort denoting messages. The first, second, and third arguments of each of `m1`, `m2` and `m3` operators are the actual creator, the seeming sender, and the receiver of the corresponding message. The first argument is meta-information that is only available to the outside observer and the agent that has sent the corresponding message, and that cannot be forged by the enemy; while the remaining arguments may be forged by the enemy. The network is modeled as a multiset of messages, which the intruder can use as his/her storage. Any message that has been sent into the network is supposed to be never deleted from the network because the intruder can replay the message repeatedly, although the intruder cannot forge the first argument.

A state of the protocol is expressed as a soup of observable components. Let ms, rs, ns, and ps be the collections of messages, random numbers, nonces, and principals, respectively. ps may contain an intruder. Let $c1s$, $c2s$, and $c3s$ be the collections of `Cipher1`, `Cipher2`, and `Cipher3` ciphertexts, respectively. We use the following observable components to formalize the NSLPK protocol as a state machine M_{NSLPK}, :

- (`nw` : ms) - It says that the network is constructed by ms.
- (`cenc1` : $c1s$) - It says that the collection of `Cipher1` ciphertexts gleaned by the intruder is $c1s$.
- (`cenc2` : $c2s$) - It says that the collection of `Cipher2` ciphertexts gleaned by the intruder is $c2s$.
- (`cenc3` : $c3s$) - It says that the collection of `Cipher3` ciphertexts gleaned by the intruder is $c3s$.
- (`nonces` : ns) - It says that the collection of *nonces* gleaned by the intruder is ns.
- (`prins` : ps) - It says that the principals participating in the protocol are ps.
- (`rand` : rs) - It says that the available random numbers are rs. Every time a principal wants to send an Init or a Resp message, it needs to generate a random and globally unique number. To formalize that behavior, we provide a fixed collection of random numbers from

the beginning, and every time a process needs to generate a random number, an element is extracted and used.

Each state in S_{NSLPK} is expressed as $\{obs\}$, where obs is a soup of observable components. We suppose that two principals `p` & `q` together with an intruder participate in the NSLPK protocol, one initial state of I_{NSLPK} namely `init` is defined as follows:

```
{(nw: emp) (rand: (r1 r2)) (nonces: emp)
(cenc1: emp) (cenc2: emp) (cenc3: emp)
(prins: (p q intr))} .
```

where `intr` is a constant of sort `Prin` denoting the intruder, and `emp` denotes an empty collection.

Three rewrite rules `Challenge`, `Response`, and `Confirmation` that formalize three actions of message exchanges are defined respectively. Let `OCs` be a Maude variable of observable component soups; `P` & `Q` be Maude variables of principals; `Ps` be a Maude variable of collections of principals; `NW`, `R`, and `N` be Maude variables denoting a network, a random number, and a nonce, respectively; `Rs`, `CE1`, and `Ns` be Maude variables denoting a collection of random numbers, `Cipher1`, and nonces, respectively. The first rewrite rule `Challenge` is defined as follows:

```
rl [Challenge] : {(nw: NW) (prins: (P Q Ps))
(rand: (R Rs)) (cenc1: CE1) (nonces: Ns) OCs }
=> {(nw: (m1(P,P,Q,enc1(Q,n(P,Q,R),P)) NW))
(cenc1: (if Q == intr then CE1 else
(enc1(Q,n(P,Q,R),P) CE1) fi)) (nonces:
(if Q == intr then (n(P,Q,R) Ns) else Ns fi))
(rand: Rs) (prins: (P Q Ps)) OCs} .
```

The rewrite rule says that when `R` is in `rand`, a new Init message is put into the network, intruder gleans information such as nonce and ciphertext used in that message if that message sends to the intruder, and `R` is removed from `rand`.

The second rewrite rule `Response` is defined as follows:

```
rl [Response] : {(nw: (m1(P,P,Q,enc1(Q,N,P)) NW))
(prins: (P Q Ps)) (rand: (R Rs))
(cenc2: CE2) (nonces: Ns) OCs }
=> {(nw: (m2(Q,Q,P,enc2(P,N,n(Q,P,R),Q)) m1(P,P,Q,enc1(Q,N,P)) NW))
(cenc2: (if P == intr then CE2 else
(enc2(Q,N,n(Q,P,R),Q) CE2) fi)) (nonces:
(if P == intr then (N n(Q,P,R) Ns) else Ns fi))
(rand: Rs) (prins: (P Q Ps)) OCs} .
```

where N and CE2 are Maude variables denoting a nonce and a collection of Cipher2. The rewrite rule says that when R is in rand, a new Resp message is put into the network, intruder gleans the nonce and ciphertext used in that message if that message sends to the intruder, and R is removed from rand. The third rewrite rule Confirmation is defined as follows:

```
rl [Confirmation] : {(nw: (m2(Q,Q,P,enc2(P,N,N',Q))
m1(P,P,Q,enc1(Q,N,P)) NW))
(prins: (P Q Ps)) (cenc3: CE3) (nonces: Ns) OCs }
=> {(nw: (m3(P,P,Q,enc3(Q,N')) m2(Q,Q,P,enc2(P,N,N',Q))
m1(P,P,Q,enc1(Q,N,P)) NW))
(cenc3: (if Q == intr then CE3 else
(enc3(Q,N') CE3) fi))
(nonces: (if Q == intr then (N' Ns) else Ns fi))
(prins: (P Q Ps)) OCs} .
```

where N' and CE3 are Maude variables denoting a second nonce used in second message and a collection of Cipher3. The rewrite rule says that when a new Ack message is put into the network, intruder gleans the nonce and ciphertext used in that message if that message sends to the intruder.

In addition to the three rewrite rules that formalize sending messages exactly following the protocol mentioned above, we also introduce six more rewrite rules to formalize the intruder's faking messages based on the gleaned information:

- fake12, fake22, and fake32: a ciphertext C is available to the intruder, the intruder fakes and sends an Init, or a Resp, or an Ack message using C, respectively.
- fake11 and fake31: a nonce N is available to the intruder, the intruder fakes and sends an Init or an Ack message using N, respectively,
- fake21: two nonces N1 and N2 are available to the intruder, the intruder fakes and sends a Resp message using N1 and N2.

The rewrite rule fake11 is defined as follows:

```
rl [fake11] : {(nw: NW) (nonces: (N Ns))
(prins: (P Q Ps)) (cenc1: CE1) OCs} =>
{(nw: (m1(intr,P,Q,enc1(Q,N,P)) NW)) (cenc1:
(if Q == intr then CE1 else (enc1(Q,N,P) CE1)
fi)) (nonces: (N Ns)) (prins: (P Q Ps)) OCs} .
```

The rewrite rule says that when N is in nonces, a new intruder's faking Init message is put into the network and the intruder gleans the ciphertext sent in that message. The remaining five rewrite rules are defined likewise.

Chapter 5

Graphical animation of IFF protocol

This chapter includes the state picture design of IFF protocol and how to design state picture in Section 5.1 and Section 5.2. The useful tips on how to guess characteristics of IFF by observing graphical animations and confirming the guessed characteristics by using Maude model checker are described in the last Section 5.3.

5.1 Simple state picture design of IFF

IFF protocol is simple to design state picture as it has only two exchange messages. However, we need to design carefully so that non-trivial characteristics of the protocol can be guessed based on state picture design. Firstly we start thinking of how to visualize messages and the flow of messages in the network. Initially, we try to make a design for the network as shown in Fig. 5.1. The design is simple, however, is not easy to observe and/or analyze the messages in the network because there are some contents (parameters) inside each message.

As shown in Fig. 5.1, there are two rectangles in which the first rectangle represents a network that contains Check messages, the second one displays Reply messages that have been put into the network respectively. "..." is displayed whenever the content of the network is overflowed. To make a better state picture design, by observing that the number of messages increases by input sequence of states, we come up with an idea that displays the contents of the most recent message that has been put into the network (hereinafter, let us call such a message as the latest message in this thesis).

Although there is only one ciphertext in Reply message, we use one consistent form (i.e., `enc`), in the state picture, to visualize the ciphertext of messages. The form is as follows: enc(Key, Rand, Identifier) where Key is a symmetric key that represents a principal (possibly `intr`) meaning that a symmetric key of a principal belongs to only one group. A symmetric key

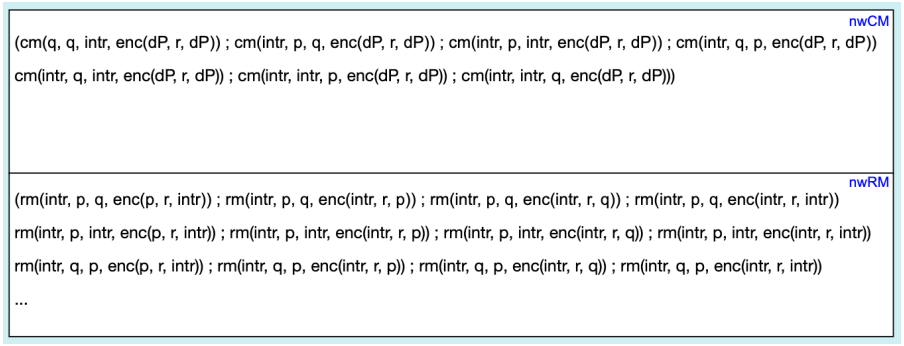


Figure 5.1: A simple state picture for IFF protocol

is given to each group, whose members share the key. When the ciphertext is in the form of `enc(dP,Rand,dP)` in Check message, Identifier receives a dummy principal `dP` as its value and Key receives a dummy principal `dP` as its value respectively.

One possible way to observe&analyze the network is by observing each message in the network. It is also equivalent to observe the latest message. Explicitly displaying the detailed contents of the latest message helps us recognize some characteristics of the protocol. We divide the network into two sub-networks for each message in Fig. 5.1. Putting all messages in one network is another possible way to make the state picture design. Each way of design has pros and cons. Designing two sub-networks makes us able to immediately recognize the specific message type in each specific network and more transparent in our visual perception when we observe each specific message or the order/relation between messages. Putting all messages in one place is simple but it may be hard to distinguish each message.

5.2 State picture design for IFF

Some observable components are introduced to visualize the state picture design. These observable components used in IFF state picture are displayed in Table 5.1:

The state picture design for IFF is depicted in Fig. 5.2. The idea of design comes from state picture design tips of the work [4]. Fig. 5.2 displays a state picture for IFF protocol in which the left side of the picture can be understood as the header and body of the message. Each type of network and collection of ciphertexts gleaned by the intruder is displayed on the right side. We visualize the header of the message as three roles that are

Observable components	Description
`newmsg`	The latest message (cm,rm)
`cm`	Latest message `cm`
`rm`	Latest message `rm`
`nwCM`	Network contains `cm` messages
`nwRM`	Network contains `rm` messages
`urand`	Used random numbers
`cipher`	Ciphertexts gleaned by intruder

Table 5.1: Observable components used in IFF state picture

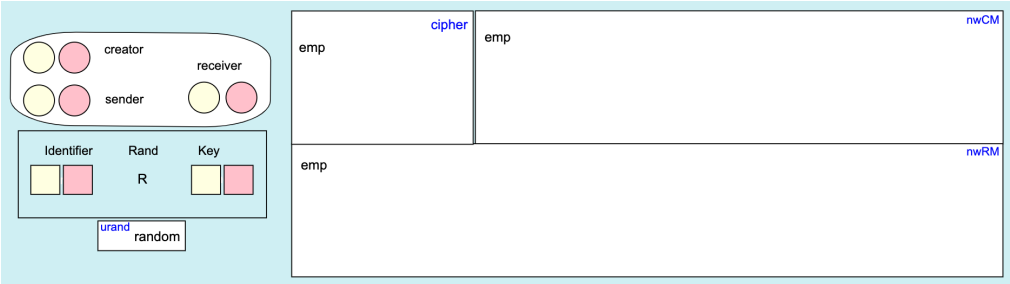
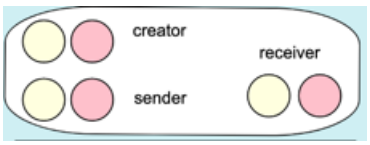


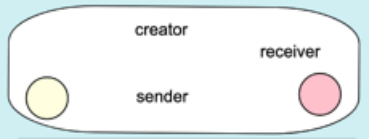
Figure 5.2: A state picture for IFF protocol

creator, sender, and receiver of message respectively. Then, the values of observable components are displayed with different colors and shapes to the corresponding place in which their roles belong to. The shapes do not matter. Pink and light yellow colors represent two different principals p and q, blank represents `intr`.

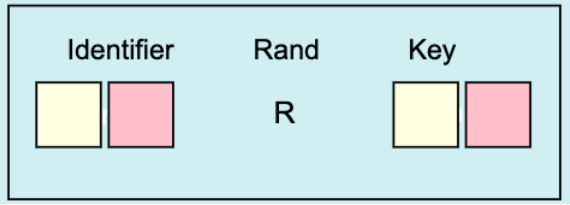
The state picture design of IFF is described in detail. The representations of the creator, sender, and receiver of the message designed in Fig. 5.2 are as follows:



The creator of the message appears at the top-left place, light yellow and pink circles represent two different principals `p` and `q`. If the value is `intr`, it is displayed as blank. For example, when creator is `intr`, sender is `p`, receiver is `q`, it is displayed as follows:

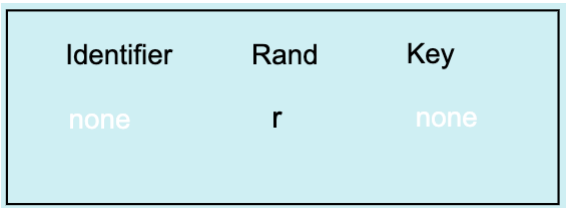


The contents of the ciphertext of message shown in Fig. 5.2 are as follows:

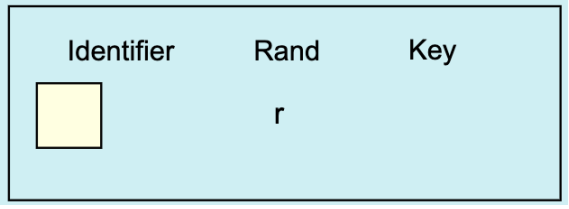


The Identifier of the ciphertext appears at the top-left place of the rectangle, pink and light yellow squares represent two principals `q` and `p`, respectively. If the value is `intr`, nothing is displayed. For the case in which the message is a `Check` message, the text "none" is displayed. The Key of the ciphertext appears at the top-right place of the rectangle. If the value is `intr`, nothing is displayed. For the case in which the message is a `Check` message, the text "none" is displayed. The Rand representation appears at the middle place in which the random number value used in the ciphertext is displayed.

Let us consider the following example. Identifier is `none`, Rand is `r` and Key is `none`. The values in example are displayed as follows:



When the latest message is `Check` message, only random value is shown and the other values are displayed as `none`. If it is not `Check` message, then it must be `Reply` message. An example of displaying the ciphertext (`intr`,`r`,`p`) of `Reply` message is as follows:



The values in the example are Identifier is `p`, Rand number is `r` and symmetric Key of the `intr`'s group. The used random numbers are displayed in urand and the ciphertexts gleaned by the intruder are displayed in cipher in Fig.5.2. The used random number urand is displayed in the rectangle at left-bottom and the values of cipher are displayed in the square at the top-left place of the right side of the state picture.

Two types of network representations are designed on the right side with two rectangles in Fig. 5.2. "..." is displayed whenever the messages are overflowed in the network. The detailed representations of the contents of a message have been described in this section. In the next section 5.3, we guess the characteristics of IFF by observing graphical animations based on our state picture design.

5.3 Guessing and confirmation of some IFF characteristics

This section presents guessing the characteristics of IFF protocol by observing graphical animations using SMGA. The guessed characteristics need to be checked that they are the actual characteristics of the protocol. Therefore, we confirm the characteristics guessed by model checking using Maude search command. Observing graphical animations of a state machine helps us to recognize some relations between observable components.

Before we observe the graphical animations, there are some tips on how to conjecture the characteristics of the protocol. The useful tips to observe the authentication protocols are as follows:

1. By knowing the relations between observable components, we may find how the values of these observable components are changing at the same time,
2. By concentrating on only one observable component, we may find that when the value of that observable component has a specific value, then any other observable components may have some specific values,
3. By concentrating on only two different observable components, we may find a relation between them,
4. By observing the order of the message in the network, we may find a relation between them,

   from which we may conjecture some characteristics.

Fig. 5.3 shows two pictures of states for M_{IFF}. Taking a look at the first picture (of State 2) makes us recogize that when the latest message is `rm(q,q,p,enc(q,r,q))` and `Key` is not `intr` (first parameter of the

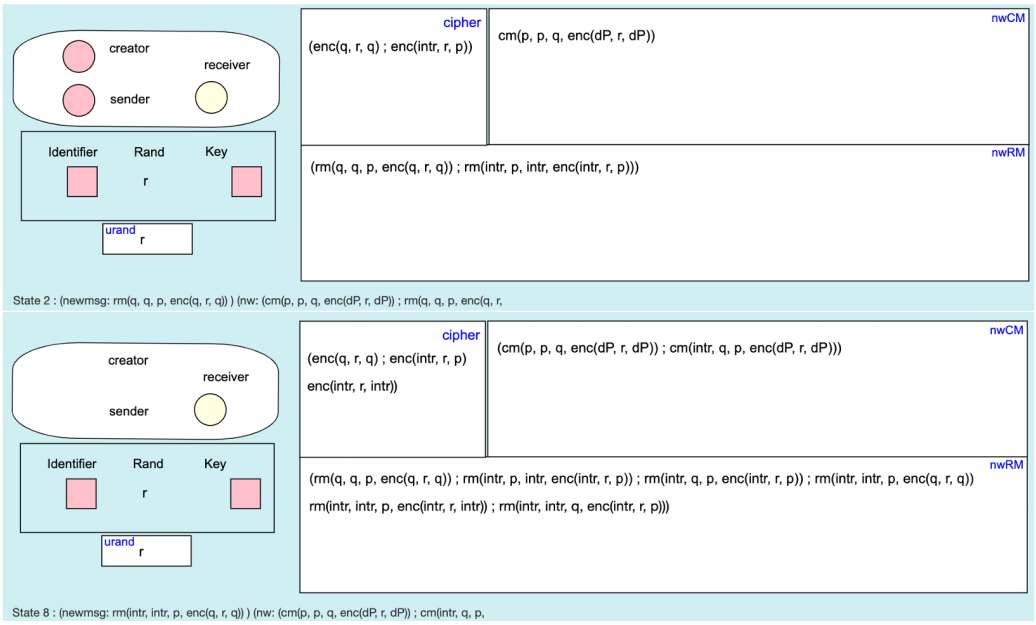


Figure 5.3: Two state pictures for IFF protocol (1)

ciphertext), then `sender` and `Identifier` of the message are not `intr`, namely `q`. Maude search command is used to confirm this characteristic as follows:

```
search [1,5] in IFF : init =>* {(nw: (rm(P1,P2,P3,enc(K,R,P2)) NW))
OCs} such that not(K =/= intr implies P2 =/= intr) .
```

where P1, P2, and P3 are creator, sender, and receiver of the message denoting `Prin` sort. The search tries to find a state such that there is a `rm(P1,P2,P3,enc(K,R,P2))` message and it is not the case that if `K` is not `intr`, `P2` is not `intr`. No counterexample is found and then IFF enjoys this characteristic at depth 5 of the state space. This characteristic is the Identifiable property of IFF protocol.

Carefully taking a look at the first picture (of State 2) and the second picture (of State 8) Fig. 5.3 helps us perceive that when the latest message is `rm` and `Key` is not `intr`, then `Identifier` of the message is not `intr` which is confirmed by using search command as follows:

```
search [1,5] in IFF : init =>* {(nw: (rm(P1,P2,P3,C) NW)) OCs}
such that not(k(C) =/= intr implies p(C) =/= intr) .
```

The search command tries to find a state such that there is a `rm` message and it is not the case that if `Key` is not `intr`, `Identifier` is not `intr`. No

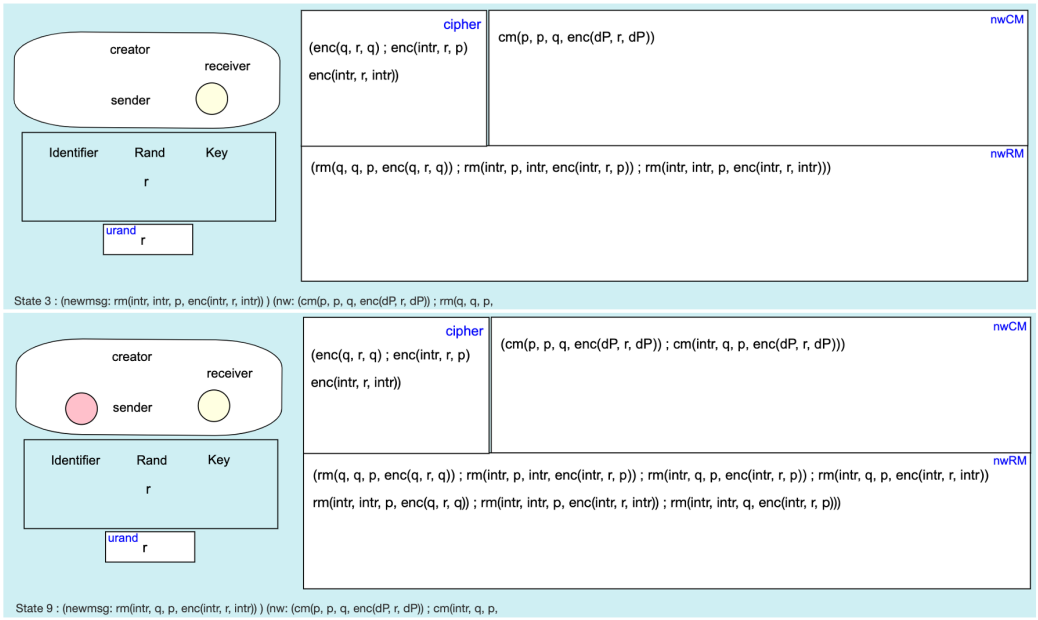


Figure 5.4: Two state pictures for IFF protocol (2)

counterexample is found and then IFF enjoys this characteristic at depth 5 of the state space.

Taking a look at the first picture (of State 3) and the second picture (of State 9) in Fig. 5.4 allows us to guess another characteristic such that whenever `Identifier` of the latest message is `intr`, then the ciphertext `enc(intr,r,intr)` of the message is in the collection of `cipher` which is confirmed by the following search command:

```
search [1,5] in IFF : init =>* {(cipher: (enc(K,R,intr) CE)) OCs}
such that not(K == intr) .
```

where K is the Maude variable representing the symmetric key of the principal `K's` group. The search tries to find a state such that there is a ciphertext `enc(K,R,intr)` whose `Identifier` is `intr` in the collection of `cipher`, and it is not the case that the `K` is `intr`. No counterexample is found at depth 5 of the state space and then IFF seems to enjoy this characteristic. Note that Maude search command does not guarantee that IFF surely enjoys these characteristics in all possible situations. To make sure that IFF enjoys the guessed properties, interactive theorem proving is one possible technique to do so. In this Section, we have confirmed that all lemmas (including one more characteristic that is not used in the formal verification) used in the formal verification of IFF protocol with CiMPA and CiMPG described in Chapter

7 can be discovered by observing graphical animations, which demonstrates the usefulness of SMGA to help human users conjecture lemmas needed to conduct formal proofs.

Chapter 6

Graphical animation of NSLPK protocol

This chapter describes the state picture design of NSLPK, how to design state picture in Section 6.1 and Section 6.2. Finally, the detailed explanation on how to conjecture characteristics of NSLPK by observing graphical animations and the confirming guessed characteristics using Maude model checker is presented in the last Section 6.3.

6.1 Simple state picture design of NSLPK

As we know state picture designs affect how well human users can detect non-trivial characteristics of protocols. Hence, we design carefully the state picture. When we start thinking of the state picture design, the network component becomes the important part that we should focus on because authentication protocols exchange messages, and those messages are in the network. Initially, we try to make a design for the network as shown in Fig. 6.1. The design is simple, however, is hard to observe and/or analyze the messages in the network because there are many contents(parameters) inside each message. As shown in Fig. 6.1, there are three rectangles in which the first rectangle represents a network that contains all messages, the second one displays the most recent message that has been put into the network, and the collection of *nonces* gleaned by the intruder is displayed in the last rectangle. "..." is displayed whenever the content of the network is overflowed. During making a better state picture design, by observing that the number of messages increases by one after each state, we come up with an idea that displays the contents of the most recent message that has been put into the network (hereinafter, let us call such a message as the latest message in this thesis).

Although there are three kinds of ciphertexts (i.e., `enc1`, `enc2`, and `enc3`), in the state picture design, we use only one form to visualize ciphertexts. The form is as follows: enc$i$(public-key, nonce1, nonce2, cipher-creator), where

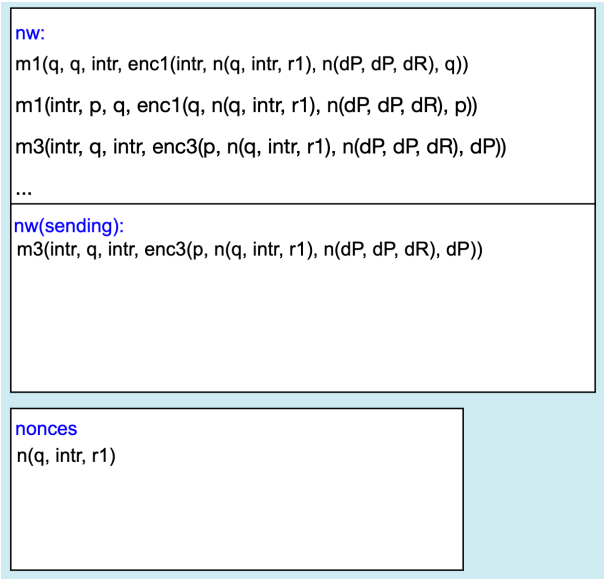


Figure 6.1: A simple state picture for NSLPK protocol

public-key is a principal (possibly `intr`), nonce1 for m1, m2, and m3 is in the following form: nonce1(generator, random, forwhom); nonce2 is in the following form: nonce2(generator, random, forwhom). When the ciphertext is in the form of `enc3`, cipher-creator receives a dummy principal `dP` as its value. Similarly, when the ciphertext is in the form of `enc1` or `enc3`, nonce2 receives a dummy value denoted by nonce2(`dP`,`dR`,`dP`), where `dR` denotes a dummy random number.

One possible way to observe&analyze the network is to observe&analyze each message in the network. Observing each message in the network is also equivalent to observe the latest message that is put into the network. Explicitly displaying the detailed contents of the latest message allows us to guess some non-trivial characteristics, which is discussed in Section 6.3. Furthermore, we design three sub-networks for three types of messages instead of one network that contains all messages. One network that contains all messages is another possible way to make the state picture design. Each way of design has some advantages as well as disadvantages.

As shown in Fig. 6.1 putting all messages in one place is simple but it is hard to distinguish each message. For IFF protocol, it seems to be not hard to distinguish each message because IFF has only two messages. However, it is not easy for NSLPK protocol. As there are more exchange messages in protocol, it is harder to distinguish one message from another.

Observable components	Description
`newmsg`	The latest message (m1,m2,m3)
`m1`	Latest message `m1`
`m2`	Latest message `m2`
`m3`	Latest message `m3`
`nwM1`	Network contains messages `m1`
`nwM2`	Network contains messages `m2`
`nwM3`	Network contains messages `m3`
`urand`	Used random numbers
`nonces`	Nonces gleaned by intruder

Table 6.1: Observable components used in NSLPK state picture

Designing three sub-networks for three types of messages helps us be able to immediately recognize the specific message type in each specific network. It makes us more transparent in our visual perception when we observe each specific message or the order/relation between messages.

6.2 State picture design for NSLPK

In addition to the observable components presented in Sect. 4.2, some more observable components are introduced to visualize the state picture design. These observable components are shown in Table 6.1.

Our state picture design for NSLPK is depicted in Fig. 6.2. Fig. 6.3 displays a state picture for NSLPK showing one arbitrary state. We first divide two roles that are creator and sender of the message into two separate places. Then, observable components are put to the corresponding place in which their roles belong to. For example, public-key should be put to the receiver's side because the sender uses the public-key of the receiver for encryption. The values of observable components are displayed with different colors and shapes. The different shapes do not matter. For example, pink and light yellow colors represent two different principals p and q, blank represents `intr`, triangles represent the contents of the nonce.

The state picture design of NSLPK is described in detail. The representation of the three types of messages designed in Fig.6.2 is as follows:

The type of the lastest message is represented by a small light gray square.

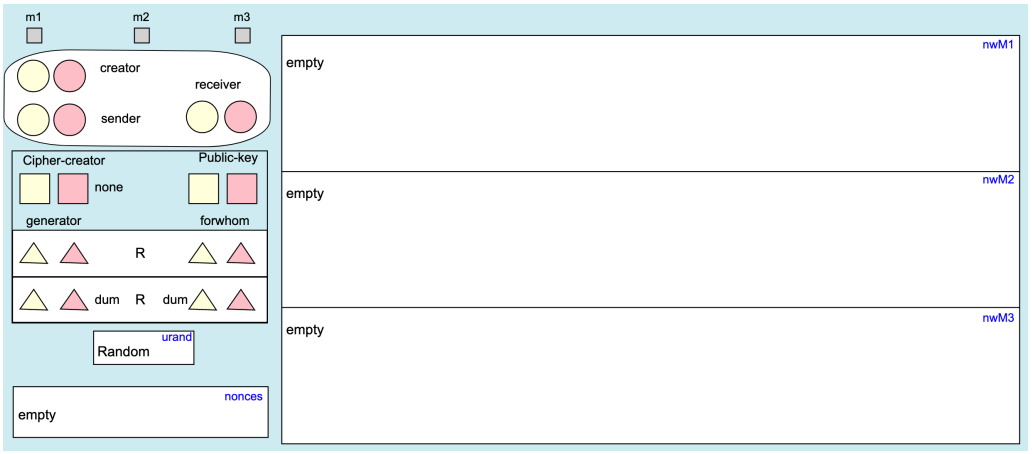
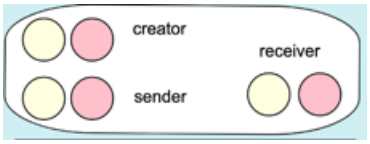


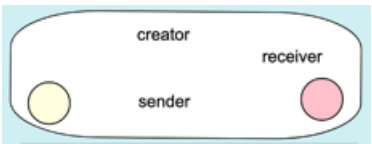
Figure 6.2: A state picture design for NSLPK protocol

At one glance this helps us recognize which type of message is showing. For example, when the lastest message is a message `m2`, there is only one light gray square displayed under `m2` as shown in the following picture:

The representations of the creator, sender, and receiver of the message designed in Fig. 6.2 are as follows:



The creator of the message appears at the top-left place, pink and light yellow circles represent two different principals `q` and `p`. If the value is `intr`, it is displayed as blank. The sender and receiver of the message appear at the bottom-left and bottom-right places, respectively. For example, when creator is `intr`, sender is `p`, receiver is `q`, it is displayed as follows:



The contents of the ciphertext of message shown in Fig. 6.2 are as follows:

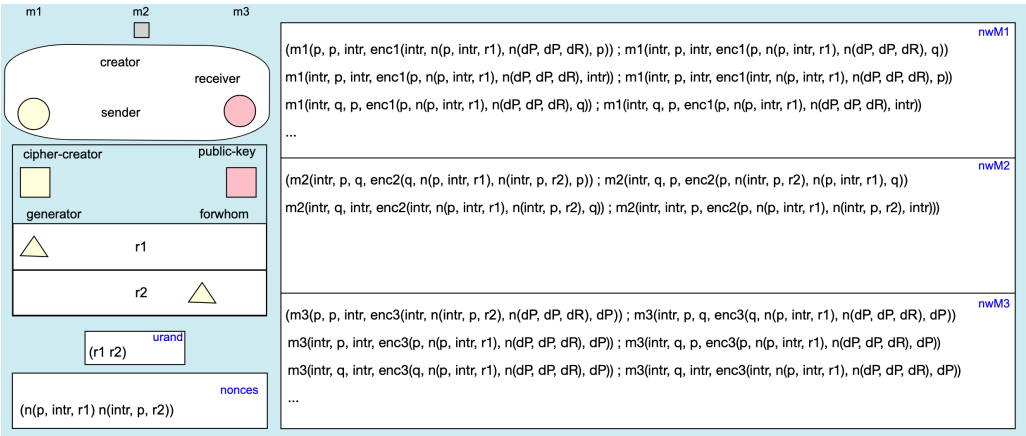
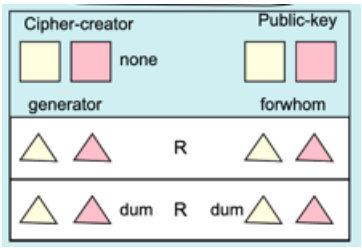


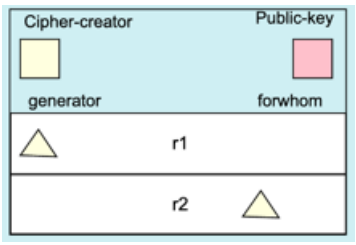
Figure 6.3: A state picture for NSLPK protocol



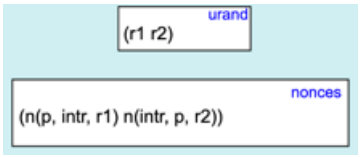
The cipher-creator of the ciphertext appears at the top-left place of the rectangle, pink and light yellow squares represent two principals `q` and `p`, respectively. If the value is `intr`, nothing is displayed. For the case the message is a message `m3`, the text "none" is displayed. The public-key of the ciphertext appears at the top-right place which is the same side of receiver because the ciphertext is encrypted by receiver's public key. If the value is `intr`, nothing is displayed.

The two nonces of the ciphertext are shown with two rectangles inside the primary rectangle, where the upper rectangle visualizes the first nonce and the lower rectangle visualizes the second nonce. Each nonce has three parameters. In the first nonce, the generator and forwhom representations appear at the left-hand side and right-hand side, respectively; pink and light yellow triangles are the principals `q` and `p`, respectively. If the value is `intr`, nothing is displayed. The random representation appears at the middle place in which the random number value used in the ciphertext is displayed. The second nonce is represented likewise. If the message is `m3` message, the text `dum` is displayed for the values of generator and forwhom in the second nonce, where `dum` denotes the dummy value `dP`.

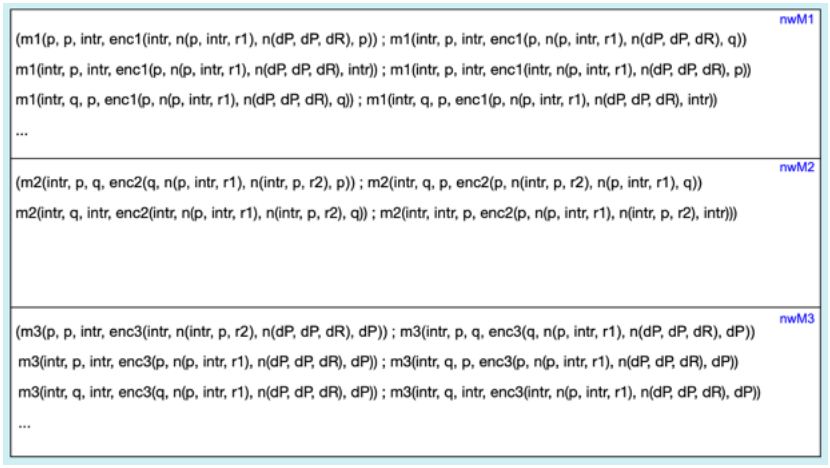
Considering the following example. cipher-creator is `p` and public-key is `q`. In the first nonce generator is `p`, random is `r1`, and forwhom is `intr`. In the second nonce, generator is `intr`, random is `r2`, and forwhom is `p`. The values in example are displayed as follows:



The used random numbers are displayed in urand and the nonces gleaned by the intruder are displayed in nonces. In Fig. 6.2, the representations of urand and nonces are designed at the left-bottom corner. The values of both urand and nonces are displayed in two rectangles as follows:



Three types of network representations are designed on the right side with three rectangles in Fig. 6.2. "..." is displayed whenever the messages are overflowed. This can be seen in the figure below:



We have described the detailed representations of the contents of a message. In the next section 6.3, we guess the characteristics of NSLPK by observing graphical animations based on our design, and then guessed characteristics are confirmed by Maude search command.

6.3 Guessing and confirmation of some NSLPK characteristics

This section presents how to guess the characteristics of the NSLPK protocol by observing graphical animations using SMGA and how to confirm the characteristics guessed by model checking. Observing graphical animations of a state machine allows users to recognize some relations between observable components (OCs). However, observing all OCs at the same time is less likely to recognize the characteristics since there are many OCs in the design picture.

There are some useful tips on how to conjecture characteristics of the NSLPK protocol by observing graphical animations with SMGA as follows:

1. By concentrating on one observable component, we may find that if the value of that observable component is `intr`, any other observable components may have some specific values, from which we may conjecture some characteristics.
2. By concentrating on two different observable components, we may find a relation between them, from which we may conjecture some characteristics.
3. By observing the order of the message in the network, we may find a relation between them, from which we may conjecture some characteristics.
4. By carefully investigating the conditions of some characteristics that have been already conjectured, from which we may conjecture some other characteristics.

Hence, we sometimes need to concentrate on some specific OCs when we observe the graphical animations. Some characteristics of the NSLPK protocol are straightforward to guess by observing graphical animations. However, some are not, precisely the characteristics that include two messages. We need to carefully observe such kind of characteristics.

Maude is used to generate a finite input sequence of states based on the Maude specification of the protocol, then feed it to SMGA which produces graphical animation of the input sequence of states. Observing such graphical animations allow us to guess the characteristics. However, the characteristics guessed are required to be checked whether or not the protocol really enjoys those characteristics. In this section, the guessed characteristics of NSLPK protocol are confirmed at a specific depth (depth 5) of the state space because the reachable state space of NSLPK is huge.

Fig.6.4 shows four states of pictures for M_{NSLPK}. Taking a look at the

first picture (of State 0) and the second picture (of State 3) helps us recognize that there is `n(intr,q,r1)` in `nonces` when `generator` is `intr`. Taking a look at the third picture (of State 30) and the fourth picture (of State 46) helps us recognize that there is `n(p,intr,r2)` in `nonces` when `forwhom` is `intr`. Any nonce gleaned by the intruder is stored in the collection of `nonces`. Hence, observing the graphical animation of these four pictures helps us guess the characteristics such that any nonce gleaned by the intruder has been generated by the intruder or a non-intruder principal that wanted to authenticate the intruder. Then Maude search command is used to confirm this characteristic as follows:

```
search [1,5] in NSLPK : init =>* {(nonces: (n(P,Q,R) Ns)) OCs}
such that not(P == intr or Q == intr) .
```

The search command tries to find a state such that there is a nonce in the collection of `nonces` whose `generator` is P, `forwhom` is Q and `random number` is R, respectively, and it is not the case that `generator` or `forwhom` is `intr`, namely both P and Q are not `intr`. No counterexample is found at depth 5 of the state space and then NSLPK seems to enjoy this characteristic. This characteristic is one of the properties of NSLPK protocol namely Nonce Secrecy Property. Note that the model checking result does not guarantee that NSLPK surely enjoys the guessed characteristics because of the state explosion problem in model checking.

Taking a look at the second picture (of State 3) and the third picture (of State 30) in Fig.6.4 allows us to guess another characteristic such that whenever `receiver` is `intr` (that displays blank in the state pictures) in the latest message, then the nonce of that message (`m1` or `m2` or `m3`) is in `nonces` which is confirmed by the following three search commands for three kinds of messages, respectively:

```
search [1,5] in NSLPK : init =>* {(nw: (m1(P,P,Q,C1) NW))
(nonces: Ns) OCs} such that not(Q == intr implies n1(C1) \in Ns) .
search [1,5] in NSLPK : init =>*
{(nw: (m2(P,P,Q,enc2(Q,N,N',P)) NW)) (nonces: Ns) OCs}
such that not(Q == intr implies (N' N \in Ns)) .
search [1,5] in NSLPK : init =>* {(nw: (m3(P,P,Q,C3) NW))
(nonces: Ns) OCs} such that not(Q == intr implies n1(C3) \in Ns) .
```

where C1, C2 and C3 are Maude variables of three kinds of ciphers respectively. Each search command tries to find a state such that there is a message (`m1` or `m2` or `m3`) in the network whose `receiver` is Q, and it is not the case that if the `receiver` of the corresponding message is `intr`, then the nonce or nonces in that message is in the collection of `nonces`. No counterexample

is found at depth 5 of the state space and then NSLPK seems to enjoy this characteristic.

Carefully observing graphical animations helps us perceive one more characteristic. Taking a look at the four pictures of Fig. 6.4, we recognize one more characteristic such that when a nonce is in `nonces`, the random number used in the nonce is stored in the collection of used random numbers `urand`. This characteristic is confirmed by using search command as follows:

```
search [1,5] in NSLPK : init =>* {(nonces: (n(P,Q,R) Ns))
(urand: Rs) OCs} such that not(R \in Rs) .
```

The search tries to find a state such that there is a nonce in the collection of `nonces` whose `random number` is R, and it is not the case that R is in the collection of used random numbers `urand`. No counterexample is found at depth 5 and then NSLPK seems to enjoy this characteristic.

The input file for SMGA is generated by using Maude specification of NSLPK protocol. Maude randomly generates the finite state sequences of input. Using one input file is not enough to observe the characteristics. The more input files we use, the more characteristics we may find out. Thus, we prepare another input file that consists of a finite sequence of states so that we can guess more characteristics by observing the behavior of the protocol. To guess some non-trivial characteristics, we observe the graphical animations in which the order of the messages is mainly focused on.

Carefully observing graphical animations of the order of messages in the network and expecting that `m2` message should follow `m1` message, from which we guess a characteristic that includes two messages as shown in Fig. 6.5. Taking a look at the first picture (of State 0), there exists a message `m1(p,p,q,enc1(q,n(p,q,r1),n(dP,dP,dR),p))` in `nwM1`. After some `m1` messages are faked by the intruder based on the gleaned information, there exists a message `m2(q,q,p,enc2(p,n(p,q,r1), n(q,p,r2),q))` in `nwM2` at the second picture (of State 5). Taking a look at the third picture (of State 19), we observe that the intruder creates many faked `m2` messages including `m2(intr,q,p,enc2(p,n(p,q,r1),n(q,p,r2), q))`. Observing the order of messages in the network allows us to conjecture the following characteristic:

- If there exists a message `m1` created by a non-intruder principal and sent to another non-intruder principal, and
- there exists a message `m2` (either created by the intruder or a non-intruder principal) that is sent to the sender of `m1`, then
- the message `m2` originates from a non-intruder principal who is the receiver of the `m1`.

This characteristic can be formally specified in Maude as follows.

```
(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))
```

Let us repeat again that P, Q, and $Q1$ are Maude variables of `Prin` (possibly intruder); R is a Maude variable of `Rand`, and N is a Maude variable of `Nonce`. where $Q1$ may or may not be equal to Q which means that $Q1$ may be an intruder. If that is the case, that `m2` message is faked by intruder. This characteristic is confirmed by the following Maude search command:

```
search [1,5] in NSLPK : init =>* {(nw: (m1(P,P,Q,enc1(Q,N,NON,P))
m2(Q1,Q,P,enc2(P,N,N',Q)) NW)) OCs} such that not(P =/= intr implies
m2(Q,Q,P,enc2(P,N,N',Q)) \in (m2(Q1,Q,P,enc2(P,N,N',Q)) NW)) .
```

The search command tries to find a state such that there are two messages `m1(P,P,Q,enc1(Q,N,NON,P))` and `m2(Q1,Q,P,enc2(P,N,N',Q))` in `NW` and it is not the case that if P is not `intr`, then `m2(Q,Q,P,enc2(P,N,N',Q))` in `NW`, namely that if there exists a message `m1` created by a non-intruder principal P and sent to another non-intruder principal Q, and there exists a message `m2` created by $Q1$ who may be an intruder that is sent to the sender of `m1`, then the message `m2` originates from a non-intruder principal Q and it is in the network. No counterexample is found at depth 5 and then NSLPK seems to enjoy this characteristic.

Similarly, we expect that a message `m3` should follow a message `m2`. Taking a look at the second picture (of State 5) in Fig. 6.5, there exists a message `m2(q,q,p,enc2(p,n(p,q,r1),n(q,p,r2),q))` in `nwM2` . Taking a look at the first picture (of State 40) in Fig. 6.6, there exists a message `m3(p,p,q,enc3(q,n(q,p,r2),n(dP,dP,dR), dP))` in `nwM3`. Taking a look at the second picture (of State 43), there exists a message `m3(intr,p,q, enc3(q,n(q,p,r2),n(dP,dP,dR),dP))` in `nwM3` which is created by `intr`. Carefully observing the order of the messages in the network, we also guess the following characteristic:

- If there exists a message m2 created by a non-intruder principal and sent to another non-intruder principal, and
- there exists a message m3 (either created by the intruder or a non-intruder principal) that is sent to the sender of the message m2, then
- the message m3 originates from the non-intruder principal who is the receiver of the message m2.

The formal description of this characteristic is specified as follows.

```
(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)
implies m3(P,P,Q,enc3(Q,n(Q,P,R))) \in nw(S))
```

where $P1$ may or may not be equal to P which means that $P1$ may be an intruder. If that is the case, that $m3$ message is faked by the intruder. This characteristic is confirmed by Maude search command as follows:

```
search [1,5] in NSLPK : init =>* {(nw: (m2(Q,Q,P,enc2(P,N,N',Q))
m3(P1,P,Q,enc3(Q,N',NON,P)) NW)) OCs}
such that not(Q =/= intr implies
m3(P,P,Q,enc3(Q,N',NON,P)) \in (m3(P1,P,Q,enc3(Q,N',NON,P)) NW)) .
```

The search command tries to find a state such that there are two messages `m2(Q,Q,P,enc2(P,N,N',Q))` and `m3(P1,P,Q,enc3(Q,N',NON,P))` in `NW` and it is not the case that if Q is not `intr`, then `m3(P,P,Q,enc3(Q,N',NON,P))` in `NW`, namely that if there exists a message `m2` created by a non-intruder principal Q and sent to another non-intruder principal P, and there exists a message `m3` created by $P1$ who may be an intruder that is sent to the sender of `m2`, then the message `m3` originates from a non-intruder principal P and it is in the network. No counterexample is found at depth 5 and then NSLPK seems to enjoy this characteristic. The above two characteristics that describe the message order in the network are called One-to-many Correspondence Property of NSLPK protocol.

Let us list the above characteristics that we observe with their informal descriptions as follows:

1. If `receiver` of the latest message is `intr`, then a nonce/nonces in that message is/are in `nonces`.
2. If a nonce is in the collection of `nonces`, then either generator or forwhom of the nonce is `intr`.
   (a) If generator of a nonce is `intr`, then the nonce is in `nonces`.
   (b) If generator and forwhom of a nonce are not `intr`, then the nonce is not in `nonces`.
3. If a nonce is in `nonces`, then random number of the nonce is in `urand`.
4. If message `m1(P,P,Q,enc1(Q,n(P,Q,R),P))` is in `nwM1` and message `m2(Q1,Q,P, enc2(P,n(P,Q,R),N,Q))` is in `nwM2` and `P` is not `intr` then `m2(Q,Q,P,enc2(P,n(P,Q,R), N,Q))` is in the network and originates from `Q`.
5. If message `m2(Q,Q,P,enc2(P,N,n(Q,P,R),Q))` is in `nwM2` and message `m3(P1,P,Q, enc3(Q, n(Q,P,R))` is in `nwM3`, and `Q` is not `intr` then `m3(P,P,Q,enc3(Q,n(Q,P,R))` is in the network and originates from `P`.

Observing graphical animations of the NSLPK produced by SMGA could help us visually perceive more characteristics. The informal descriptions of some guessed characteristics are as follows:

1. If the latest message is `m1` and `cipher-creator` of `m1` is `intr`, then the nonce of `m1` is in `nonces` (i.e., the nonce is gleaned by the intruder).
2. If the latest message is `m1` that forms as `m1(P,P,Q,C1)` and the receiver `Q` is `intr`, then the generator of the nonce is not `intr`.
3. If the latest message is `m1` that forms as `(m1(P,P,Q,enc1(Q,N,P))` and `P` is `intr`, then generator or forwhom of the nonce `N` is `intr`.
4. If the latest message is `m2` that forms as `m2(P,P,Q,enc2(Q,N1,N2,P))` and `P` is not `intr`, then the forwhom of `N2` is `Q`.
5. If the latest message is `m3` that forms as `m3(P,P,Q,enc3(Q,N))`, and `P` and `Q` are not `intr`, then the generator of `N` is `Q`.
6. If `public-key` of the latest message `m1` is `intr`, then a nonce/nonces in that message is/are in `nonces`.
7. If `public-key` of the latest message `m2` is `intr`, then a nonce/nonces in that message is/are in `nonces`.
8. If `public-key` of the latest message `m3` is `intr`, then a nonce/nonces in that message is/are in `nonces`.
9. If a nonce in the latest message forms as `n(P,Q,R)`, and `P` is not `intr`, then `R` is in `urand`.

The guessed characteristics are confirmed by using Maude search commands as follows:

```
Characteristic 1:
search [1,5] in NSLPK : init =>* {(nw: (m1(P,P,Q,C1) NW))
(nonces: Ns) OCs}
such that not(p(C1) == intr implies n1(C1) \in Ns) .

Characteristic 2:
search [1,5] in NSLPK : init =>* {(nw: (m1(P,P,Q,C1) NW)) OCs}
such that not(Q == intr implies gen(n1(C1)) =/= intr) .

Characteristic 3:
search [1,5] in NSLPK : init =>*
{(nw: (m1(P,P,Q,enc1(Q,N,NON,P)) NW)) OCs} such that
not(P == intr implies (gen(N) == intr or forwhom(N) == intr)) .

Characteristic 4:
search [1,5] in NSLPK : init =>*
{(nw: (m2(P,P,Q,enc2(Q,N,N',P)) NW)) OCs}
such that not(P =/= intr implies forwhom(N') == Q) .

Characteristic 5:
search [1,5] in NSLPK : init =>*
```

```
{(nw: (m3(P,P,Q,enc3(Q,N)) NW)) OCs}
such that not(P =/= intr and Q =/= intr implies gen(N) == Q) .

Characteristic 6:
search [1,5] in NSLPK : init =>* {(nw: (m1(P,P,Q,C1) NW))
(nonces: Ns) OCs}
such that not(k(C1) == intr implies n1(C1) \in Ns) .

Characteristic 7:
search [1,5] in NSLPK : init =>* {(nw: (m2(P,P,Q,C2) NW))
(nonces: Ns) OCs}
such that not(k(C2) == intr implies (n1(C2) n2(C2) \in Ns)) .

Characteristic 8:
search [1,5] in NSLPK : init =>* {(nw: (m3(P,P,Q,C3) NW))
(nonces: Ns) OCs}
such that not(k(C3) == intr implies n1(C3) \in Ns) .

Characteristic 9:
search [1,5] in NSLPK : init =>* {(nonces: (n(P,Q,R) Ns))
(urand: Rs) OCs} such that not(P =/= intr implies R \in Rs) .
```

Each search command tries to find the reachable states that match the pattern and make the conditions (`such that`) true. Each search does not find any counterexamples. Therefore, NSLPK enjoys these characteristics at depth 5. Although model checking is useful and convenient to confirm guessed characteristics, it does not guarantee that the protocol surely enjoys guessed characteristics when the reachable state space of the state machine that formalized the protocol is huge. To make sure that, we conduct formal verification of the protocol with CiMPG in the next Chapter 7.

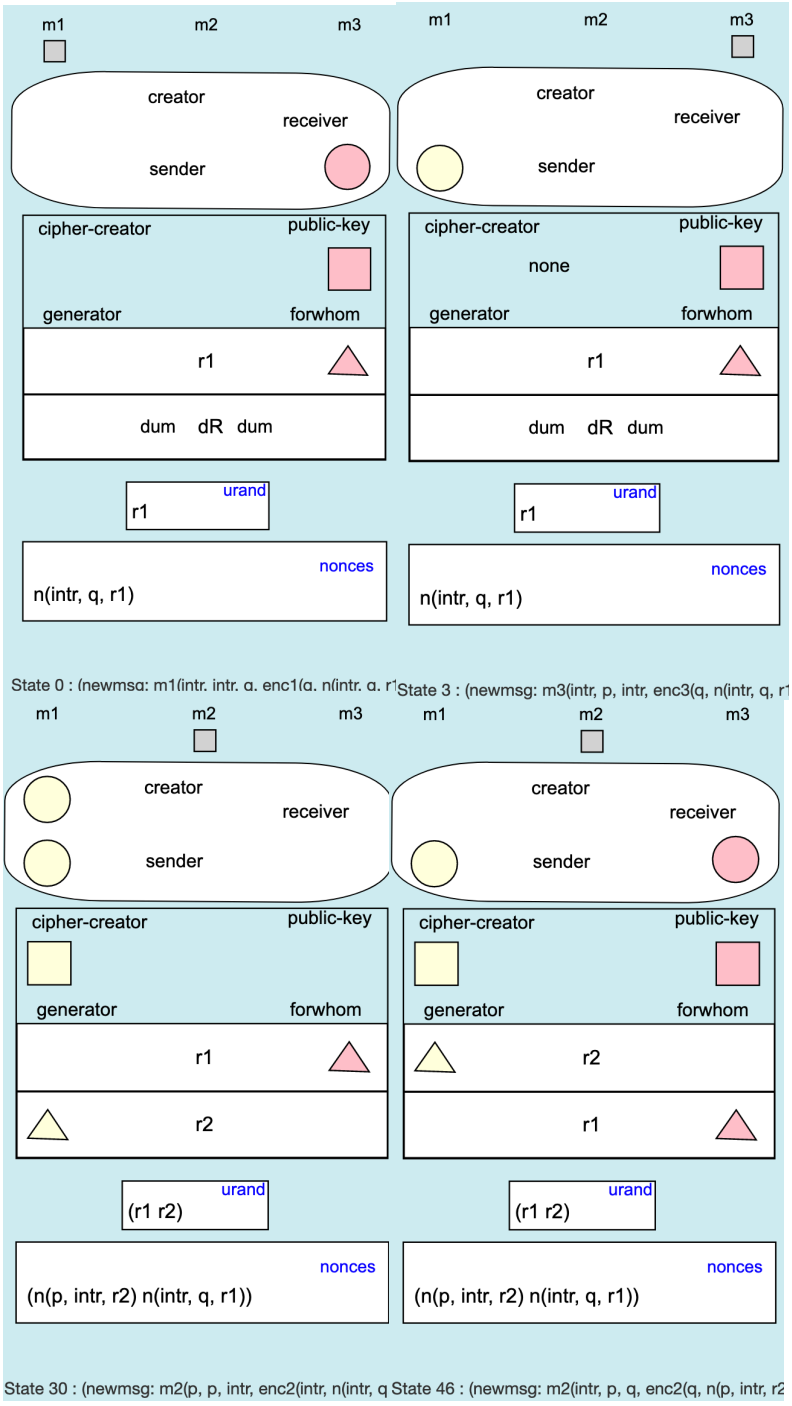
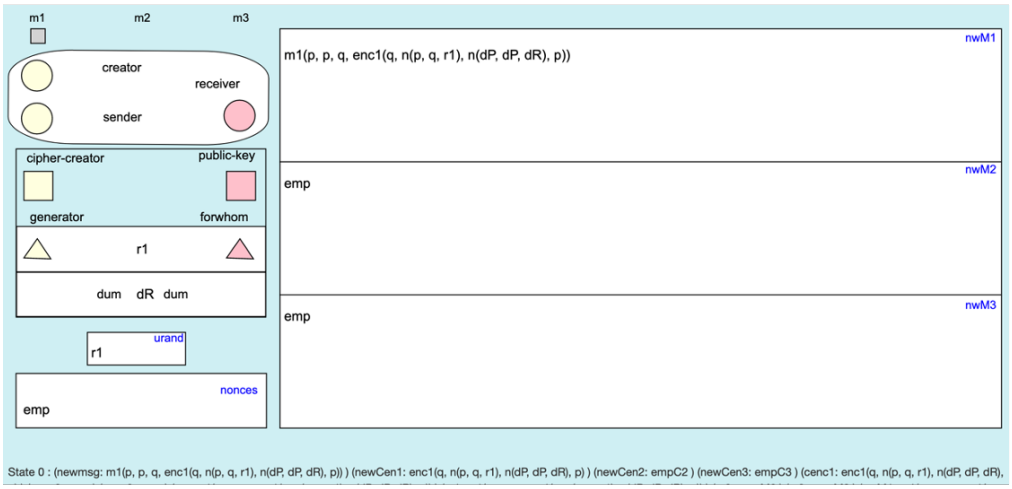
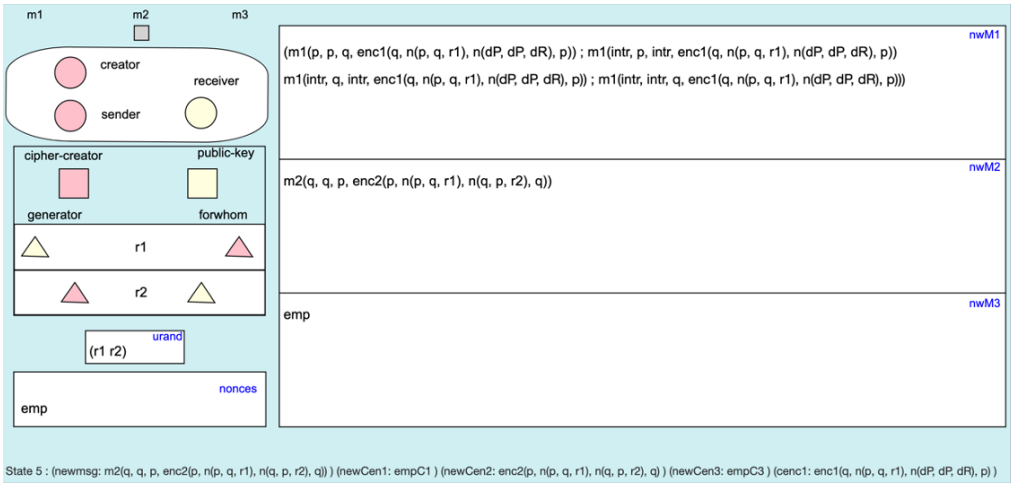


Figure 6.4: Some state pictures for NSLPK protocol (1)





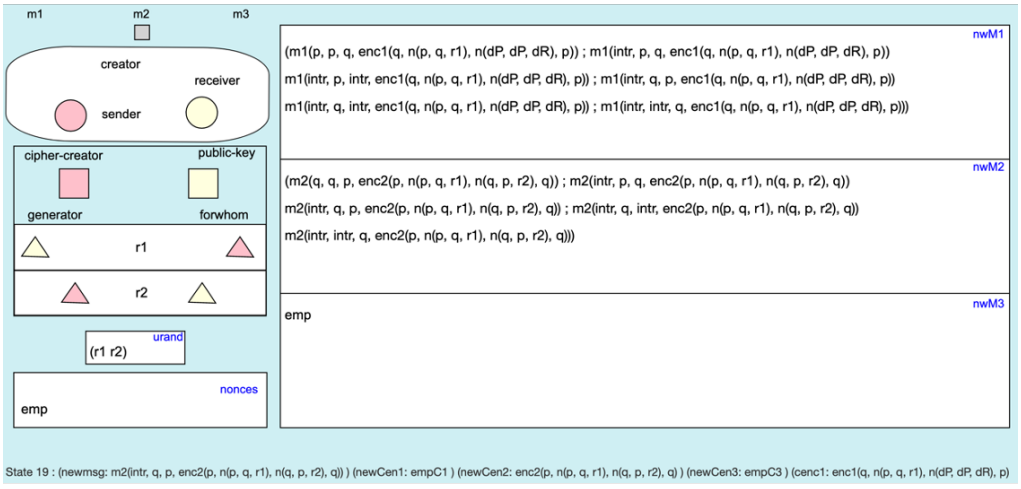


Figure 6.5: Some state pictures for NSLPK protocol (2)

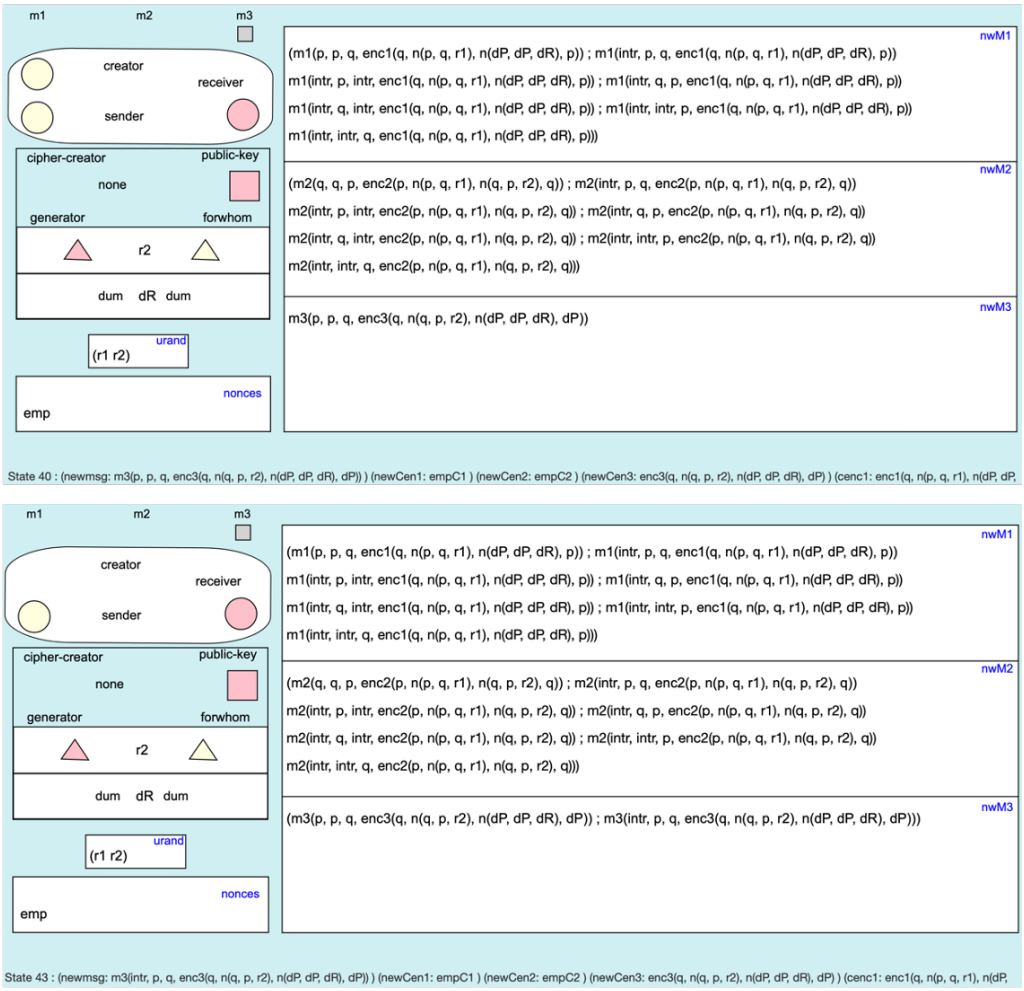


Figure 6.6: Some state pictures for NSLPK protocol (3)

Chapter 7

Formal verification of IFF & NSLPK with CiMPA and CiMPG

This chapter describes the formal verification of two authentication protocols with CiMPA and CiMPG. In Section 7.1, the formal specification of IFF in CafeOBJ is explained. In Section 7.2, formal verification of IFF with CiMPA is described in sub-section 7.2.1 and formal verification of IFF with CiMPG is described in sub-section 7.2.2. The formal specification of NSLPK in CafeOBJ is presented in Section 7.3 and formal verification of NSLPK with CiMPG is described in the last Section 7.4.

7.1 Formal specification of IFF in CafeOBJ

We first declare the following operators to specify the ciphertext used in the protocol as follows:

```
op enc : Key Rand Prin -> Cipher .
op k : Cipher -> Key .
op r : Cipher -> Rand .
op p : Cipher -> Prin .
```

where `Key` is the sort (or type) representing symmetric keys, `Rand` is the sort denoting random numbers, `Prin` is the sort representing principals, and `Cipher` is the sort denoting ciphertexts. Given a key k, a random number r and a principal p, the operator `enc`(k, r, p) denotes the ciphertext obtained by encrypting r and p with k. Operators `k`, `r` and `p` return the first, second and third arguments of `enc`(k, r, p), respectively.

We specify two messages Check and Reply by two operators `cm` and `rm` as follows:

```
op cm : Prin Prin Prin Rand   -> Msg
op rm : Prin Prin Prin Cipher -> Msg
```

where `Msg` is the sort denoting messages. The first, second, and third arguments of each of `cm` and `rm` are the actual creator, the seeming sender,

and the receiver of the corresponding message. The first argument of each message is meta-information that is only available to the outside observer and the principal that has sent the corresponding message, and that cannot be forged by the enemy; while the remaining arguments may be forged by the enemy.

The network is modeled as a multiset of messages, which the enemy can use as his/her storage. Any message that has been sent or put once into the network is supposed to be never deleted from the network because the enemy can replay the message repeatedly, although the enemy cannot forge the first argument. Consequently, the empty network (i.e., the empty multiset) means that no messages have been sent.

The enemy tries to glean two kinds of values from the network. They are random numbers and ciphertexts. The collections of these values gleaned by the enemy are denoted by operators `rands` and `ciphers`, which are declared as follows:

```
op rands   : Network -> ColRands
op ciphers : Network -> ColCiphers
```

where `Network` is the sort denoting networks, `ColRands` is the sort denoting collections of random numbers, and `ColCiphers` is the sort denoting collections of ciphertexts. `ciphers` is defined by the following equations:

```
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 \in ciphers(NW)
if not(rm?(M) and C = c(M)) .
```

where `void` denotes the empty multiset (or empty network), operator `rm?` checks if a given message is a Reply message, operator `c` takes a Reply message as a parameter and returns its ciphertext (i.e., the fourth argument of `rm` operator), `\in` is an infix operator checking the existence of an element in a collection, and operator `,` in `M , NW` denotes the data constructor of nonempty multisets. The equations say that a ciphertext `C` is available to the enemy if and only if there exists a Reply message whose content is `C`. `rands` can be defined likewise.

Each state of IFF protocol is characterized by two observational functions `nw` and `ur` to observe the network and the set of used random numbers, respectively as follows:

```
op nw : Sys -> Network .
op ur : Sys -> URands .
```

where `Sys` is the sort denoting the state space of IFF, `URands` is the sort denoting the set of used random numbers.

A constructor is used to represent an arbitrary initial state as follows:

```
op init  : -> Sys {constr}
```

`init` is defined in terms of equation, specifying the values observed by the two observer functions in an arbitrary initial state as follows:

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

We use five transitions that are also constructors as follows:

```
op sdcm  : Sys Prin Prin Rand -> Sys {constr}
op sdrm  : Sys Prin Msg -> Sys {constr}
op fkcm1 : Sys Prin Prin Rand -> Sys {constr}
op fkrm1 : Sys Prin Prin Cipher -> Sys {constr}
op fkrm2 : Sys Prin Prin Rand -> Sys {constr}
```

where `sdcm` and `sdrm` transitions formalize sending Check and Reply messages exactly following the protocol. The remaining actions `fkcm1`, `fkrm1`, and `fkrm2` are the enemy's faking messages, which can be understood as follows:

- `fkcm1`: If a random number `R` is available to the enemy, the enemy fakes and sends a Check message using `R`.
- `fkrm1`: If a ciphertext `C` is available to the enemy, the enemy fakes and sends a Reply message using `C`.
- `fkrm2`: If a random number `R` is available to the enemy, the enemy fakes and sends a Reply message using `R`.

Each of the five transition functions is defined in terms of equations specifying the values observed by the two observer functions. Each transition function has an effective condition. Let S be a CafeOBJ variable of Sys, $P1$, $P2$ be CafeOBJ variables of $Prin$ and R be a CafeOBJ variable of $Rand$.

The transition `sdcm` is defined as follows:

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

The equations say that if the effective condition `c-sdcm(S,P1,P2,R)` is true (i.e., `R` is fresh random number and has not been used), then the Check message `cm(P1,P1,P2,R)` is put into the network `nw(S)`, `R` is put into `ur(S)` in the state denoted by `sdcm(S,P1,P2,R)`; if `c-sdcm(S,P1,P2,R)` is false, nothing changes.

The transition `sdrm` is defined as follows:

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

The equations say that if the effective condition `c-sdrm(S,P1,M1)` is true (i.e., `M1` is *Check* message and it is in the network, and $P1$ is the destination (receiver) of `M1`), then the Reply message `rm(P1,P1,src(M1),enc(k(P1),r(M1),P1))` is put into the network `nw(S)` in the state denoted by `sdrm(S,P1,M1)`; if the effective condition is false, nothing changes. The other three transitions are defined and can be understood likewise.

7.2 Formal verification of IFF with CiMPA and CiMPG

The formal verification of IFF with CiMPA and CiMPG has been conducted by Fujii [15].

7.2.1 Formal verification of IFF with CiMPA

One property of IFF we would like to confirm is whenever p receives a valid Reply message from q, q is always a member of the p's group. Such property is called Identifiable property. The property is specified as follows:

```
op inv1 : Sys Prin Prin Prin Key Rand -> Bool .
eq inv1(S,P1,P2,P3,K,R) = ((not(K = k(enemy))
and rm(P1,P2,P3,enc(K,R,P2)) \in nw(S)) implies not(P2 = enemy)) .
```

We describe how to prove that IFF satisfies the property by writing proof scripts and running with CiMPA. In the proof of `inv1`, we need to use another lemma `inv2` which is defined as follows:

```
op inv2 : Sys Key Rand -> Bool .
eq inv2(S,K,R) = (enc(K,R,enemy) \in ciphers(nw(S))
implies (K = k(enemy))) .
```

The proof starts with the goals we need to prove:

```
open IFF .
:goal{
eq [iff1 :nonexec] : inv1(S:Sys,P:Prin,
P1:Prin,P0:Prin,K:Key,R:Rand) = true .
eq [iff :nonexec] : inv2(S:Sys,K:Key,R:Rand) = true . }
```

where `IFF` is the module in which the specification of IFF together with `inv1` and `inv2` are available. `:nonexec` instructs CafeInMaude not to use the equations as rewrite rules.

Then, we select the variable `S` with the command `:ind on` as the variable on which we start proving the goals by simultaneous induction:

```
:ind on (S:Sys)
:apply(si)
```

The command `:apply(si)` starts the proof by simultaneous induction on the chosen variable `S`, generating six sub-goals for six transitions `fkcm1`, `fkrm1`, `fkrm2`, `init`, `sdcm`, and `sdrm`, where `si` stands for simultaneous induction. Each sub-goal consists of two equations to prove, corresponding to `inv1` and `inv2`. With the first sub-goal for `fkcm1`, we first apply theorem of constants by using the command: `:apply(tc)`. The command generates two sub-goals as follows:

```
1-1.> TC eq [iff1 :nonexec]: inv1(fkcm1(S#Sys,P#Prin,P0#Prin,
R#Rand),P@Prin,P1@Prin,P0@Prin,K@Key,R@Rand) = true .
1-2. TC eq [iff :nonexec]: inv2(fkcm1(S#Sys,P#Prin,P0#Prin,R#Rand),
K@Key,R@Rand) = true .
```

The command `:apply(tc)` replaces CafeOBJ variables with fresh constants in goals. `S#Sys`, `P#Prin`, `P0#Prin`, and `R#Rand` are fresh constants introduced by the command `:apply(si)`, while `P@Prin`, `P1@Prin`, `P0@Prin`, `K@Key`, and `R@Rand` are fresh constants introduced by `:apply(tc)`.

To discharge goal 1-1, the following commands are first introduced:

```
:def c1 = :ctf [R#Rand \in rands(nw(S#Sys)) .]
:apply(c1)
```

Goal 1-1 is split into two sub-goals 1-1-1 and 1-1-2 corresponding to when `R#Rand \in rands(nw(S#Sys))` holds and does not hold, respectively. Then, two sub-goals are discharged by the following commands:

```
:imp [iff1] by {K:Key <- K@Key ; P0:Prin <- P0@Prin ;
P1:Prin <- P1@Prin ; P:Prin <- P@Prin ; R:Rand <- R@Rand ;}
:apply (rd)
```

```
:imp [iff1] by {K:Key <- K@Key ; P0:Prin <- P0@Prin ;
P1:Prin <- P1@Prin ; P:Prin <- P@Prin ; R:Rand <- R@Rand ;}
:apply (rd)
```

The induction hypothesis is instantiated by replacing the variables with the fresh constants and the instance is used as the premise of the implication. For example, `P1:Prin` is replaced with `P1@Prin`. Then, `:apply(rd)` is used to check if the current goal can be discharged. Two goals 1-1-1 and 1-1-2 are discharged in this case. The current goal moves to 1-2.

Goal 1-2 is split into two sub-goals by case splitting the following equation `c2` and they are discharged by the following commands:

```
:def c2 = :ctf [R#Rand \in rands(nw(S#Sys)) .]
:apply(c2)
:imp [iff] by {K:Key <- K@Key ; R:Rand <- R@Rand ;}
:apply (rd)
:imp [iff] by {K:Key <- K@Key ; R:Rand <- R@Rand ;}
:apply (rd)
```

We have all done with goal 1, the current goal moves to 2. With goal 2, we first introduce the following commands to conduct case splitting.

```
:def c3 = :ctf [C#Cipher \in ciphers(nw(S#Sys)) .]
:apply(c3)
:def c4 = :ctf {eq P@Prin  = enemy .}
:apply(c4)
:def c5 = :ctf {eq P#Prin  = P1@Prin .}
:apply(c5)
:def c6 = :ctf {eq P0#Prin  = P0@Prin .}
:apply(c6)
:def c7 = :ctf {eq k(C#Cipher) = K@Key .}
:apply(c7)
:def c8 = :ctf {eq r(C#Cipher) = R@Rand .}
:apply(c8)
:def c9 = :ctf {eq p(C#Cipher) = P1@Prin .}
:apply(c9)
:def c10 = :ctf {eq K@Key  = k(enemy) .}
:apply(c10)
```

Case splittings are carried out based on one Boolean term and seven equations. The first sub-goal in which the Boolean term is true and seven equations hold can be discharged:

```
:imp [iff1] by {K:Key <- K@Key ; P0:Prin <- P0@Prin ;
P1:Prin <- P1@Prin ; P:Prin <- P@Prin ; R:Rand <- R@Rand ;}
:apply (rd)
```

However, with the sub-goals in which the Boolean term is true, first six equations hold and the last equation does not hold, we need to conduct case splitting more as well as use `inv2` as a lemma:

```
:def c11 = :ctf {eq P1@Prin  = enemy .}
:apply(c11)
:def c12 = :ctf [enc(K@Key,R@Rand,enemy)
  \in ciphers(nw(S#Sys)) .]
:apply(c12)
:imp [iff] by {K:Key <- K@Key ; R:Rand <- R@Rand ;}
:imp [iff1] by {K:Key <- K@Key ; P0:Prin <- P0@Prin ;
P1:Prin <- P1@Prin ; P:Prin <- P@Prin ; R:Rand <- R@Rand ;}
:apply (rd)
```

The lemma `inv2` is instantiated by replacing the variables `K:Key` and `R:Rand` with the fresh constants `K@Key` and `R@Rand`. The induction hypothesis is instantiated by replacing the variables with the fresh constants, and the instance is used as the premise of the implication. Then, `:apply(rd)` is used to discharge the current goal. The remaining sub-goals of 2 can be discharged directly without using any lemma. The remaining goals from 3 to 6 can be discharged likewise.

7.2.2 Formal verification of IFF with CiMPG

The following is the proof score for the case corresponding to goal 1-1-1 in the previous section:

```
open IFF .
op s : -> Sys .
ops a b c d e : -> Prin .
op k : -> Key .
ops r1 r2 : -> Rand .
eq (r2 \in rands(nw(s))) = true .
red inv1(s,a,b,c,k,r1) implies inv1(fkcm1(s,d,e,r2),a,b,c,k,r1) .
close
```

where `open` makes the module `IFF` available, `close` stops the use of the module and `red` reduces (computes) the given term. `s` and `k` correspond to `S#Sys` and `K@Key` in the last section, respectively. `a`, `b`, `c`, `d`, and `e` correspond to `P@Prin`, `P1@Prin`, `P0@Prin`, `P#Prin`, and `P0#Prin`, respectively. `r1` and `r2` correspond to `R@Rand` and `R#Rand`, respectively. The details of the proof score approach as well as how to write proof scores to conduct formal verification can be found in papers [8]. In comparison with proof scripts, proof scores are easier to understand for human users, and writing proof scores are also more

flexible than writing proof scripts. That is the reason why when conducting formal verification, we prefer to write proof scores rather than proof scripts. However, because of the flexibility, proof scores are subject to human errors. For example, during the verification users may overlook some cases, leading to the flaw verification.

After writing proof scores that IFF protocol enjoys the property, we can confirm that the proof scores are correct by doing the formal verification with CiMPA as described in the previous section. Although we can conduct the formal verification with CiMPA once we have completed formal verification by writing proof scores in CafeOBJ, it would be preferable to automatically confirm the correctness of proof scores. CiMPG makes it possible to automatically confirm the correctness of proof scores by generating proof scripts for CiMPA from the proof scores.

To use CiMPG, we need to add one more open-close fragment to the proof scores, which is as follows:

```
open IFF .
  :proof(iff)
close
```

where `iff` is just an identifier, can be replaced by another more preferable one.

Moreover, we need to write `:id(iff)` in each open-close fragment. For example, the above open-close fragment becomes as follows:

```
open IFF .
:proof(iff)
op s : -> Sys .
ops a b c d e : -> Prin .
op k : -> Key .
ops r1 r2 : -> Rand .
eq (r2 \in rands(nw(s))) = true .
red inv1(s,a,b,c,k,r1) implies inv1(fkcm1(s,d,e,r2),a,b,c,k,r1) .
close
```

Feeding the annotated proof scores into CiMPG, CiMPG generates the proof script for CiMPA. The generated proof script is quite similar to the one written manually. Feeding the generated proof script into CiMPA, CiMPA discharges all goals, confirming that the proof scores are correct. Formal verification of IFF protocol is conducted on a computer that carries 1.6 GHz processor and 16 GB memory. In this IFF case study, CiMPG took 1m15s to generate proof script from correct proof scores and, CiMPA took 8s to discharge all goals.

7.3 Formal specification of NSLPK in CafeOBJ

The following three operators represent three kinds of ciphertexts used in the protocol.

```
op enc1 : Prin Nonce Prin -> Cipher1 .
op enc2 : Prin Nonce Nonce Prin -> Cipher2 .
op enc3 : Prin Nonce -> Cipher3 .
```

where `Prin` is the sort denoting principals; `Nonce` is the sort denoting the nonce numbers; `Cipher1`, `Cipher2`, and `Cipher3` are the sorts denoting three kinds of ciphertexts contained in Init, Resp, and Ack messages, respectively. Given principals p, q and a nonce n_p term `enc1`(q, n_p, p) denote the ciphertext $\varepsilon_q(n_p, p)$ obtained by encrypting nonce n_p and ID p with the principal q's public key. Given principals p, q and nonces n_p, n_q term `enc2`(p, n_p, n_q, q) denote the ciphertext $\varepsilon_p(n_p, n_q, q)$ obtained by encrypting nonce n_p, n_q and ID q with the principal p's public key. Given principals p, q and a nonce n_q term `enc3`(q, n_q) denote the ciphertext $\varepsilon_q(n_q)$ obtained by encrypting nonce n_q with the principal q's public key. Hereinafter, let us use `Cipher1` (or `Cipher2`, or `Cipher3`) ciphertexts to refer to the ciphertexts sent in Init (or Resp, or Ack) messages.

A `Nonce` is defined by the following operator:

```
op n : Prin Prin Rand -> Nonce .
```

where the first and second arguments are `Prin` and the third argument `Rand` is the sort denoting random numbers that makes the nonce globally unique and unguessable. Given principals p, q and random value r, a nonce is formalized as the term `n`(p, q, r) denoting a nonce generated by principal p to be sent to principal q for authenticating where r is a fresh random number.

Three kinds of messages used in NSLPK protocol are specified as follows:

```
op m1 : Prin Prin Prin Cipher1 -> Msg
op m2 : Prin Prin Prin Cipher2 -> Msg
op m3 : Prin Prin Prin Cipher3 -> Msg
```

where `Msg` is the sort denoting messages. The first, second, and third arguments of each of `m1`, `m2` and `m3` operators are the actual creator, the seeming sender, and the receiver of the corresponding message. The first argument is meta-information that is only available to the outside observer and the agent that has sent the corresponding message, and that cannot be forged by the enemy; while the remaining arguments may be forged by the enemy. The network is modeled as a multiset of messages, which the intruder

can use as his/her storage. Any message that has been sent into the network is supposed to be never deleted from the network because the intruder can replay the message repeatedly, although the intruder cannot forge the first argument.

The intruder tries to glean four kinds of values from the network, which are nonces and three kinds of ciphertexts from three kinds of messages. Then, we use following four operators to denote those values:

```
op cnonce : Network -> ColNonce
op cenc1 : Network -> ColCipher1
op cenc2 : Network -> ColCipher2
op cenc3 : Network -> ColCipher3
```

where `Network` is the sort denoting networks (i.e., multisets of messages) and `ColNonce` is a sort denoting a collection of nonces. `ColCipher1`, `ColCipher2` and `ColCipher3` denotes the collections of ciphers corresponding to three kinds of ciphertexts.

The equations defining `cenc1` are as follows:

```
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)) .
```

where `E1` is a CafeOBJ variable of `Cipher1` and `void` denotes the empty multiset (or empty network). `\in` is an infix operator checking the existence of an element in a collection. Operator `,` in `M , NW` denotes the data constructor of nonempty multisets. `m1?` checks if a given message is an Init message. Operator `cipher1` takes an Init message as an argument and returns its ciphertext (i.e., the fourth argument of `m1` operator). Operator `key` takes a ciphertext as an argument and returns the principal in which the ciphertext is encrypted with its public key. The equations say that a ciphertext `E1` is available to the intruder iff there exists an Init message whose content is `E1` and `E1` is not encrypted by the intruder's public key. Let us note that, if `E1` is encrypted by the intruder's public key, `E1` can be rebuilt by the intruder. `cenc2`, and `cenc3` can be defined likewise.

`cnonce` is formally specified by the following equations.

```
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(cipher1(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) .
```

where `N` and `M` are CafeOBJ variables of sorts `Nonce` and `Msg` respectively. `cnonce` is a CafeOBJ variable of sort `ColNonce` denoting the collection of nonces gleaned by the intruder. The equations say that if a nonce is created by the intruder, the nonce is available to the intruder. If there exists a message in the network and the cipher in the message is encrypted with the intruder's public key, the nonces in the cipher are available to the intruder. Those are the only nonces available to the intruder.

We use two observational functions `nw` and `ur` to observe the network and the set of used random numbers, respectively as follows:

```
op nw : Sys -> Network .
op ur : Sys -> URand .
```

where `Sys` is the sort denoting the state space of NSLPK, `URand` is the sort denoting a set of used random numbers. We define nine transitions, together with one constant that represents an arbitrary initial state to specify NSLPK as follows:

```
op init : -> Sys {constr}
op sdm1 : Sys Prin Prin Rand -> Sys {constr}
op sdm2 : Sys Prin Rand Msg -> Sys {constr}
op sdm3 : Sys Prin Rand Msg Msg  -> Sys {constr}
op fkm11 : Sys Prin Prin Cipher1 -> Sys {constr}
op fkm12 : Sys Prin Prin Nonce -> Sys {constr}
op fkm21 : Sys Prin Prin Cipher2 -> Sys {constr}
op fkm22 : Sys Prin Prin Nonce Nonce -> Sys {constr}
op fkm31 : Sys Prin Prin Cipher3 -> Sys {constr}
op fkm32 : Sys Prin Prin Nonce -> Sys {constr}
```

where `URand` is the sort denoting sets of random numbers. `ur`, `nw`, and `init` can be understood as in the last section. The first three transitions formalize sending messages exactly following the protocol, while the remaining formalize the intruder's faking messages, which can be understood as follows:

- `fkm11`, `fkm21`, and `fkm31`: If a ciphertext `C` is available to the intruder, the intruder fakes and sends a/an Init, or Resp, or Ack message using `C`, respectively.
- `fkm12` and `fkm32`: If a nonce `N` is available to the intruder, the intruder fakes and sends an Init or Ack message using `N`, respectively,
- `fkm22`: If two nonces `N1` and `N2` are available to the intruder, the intruder fakes and sends a Resp message using `N1` and `N2`.

Let `S` be a CafeOBJ variable of `Sys`, and `P` & `Q` are CafeOBJ variables of `Prin`. `fkm11` is defined as follows:

```
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) .
```

where `c-fkm11(S,P,Q,E1)` is `E1 \in cenc1(nw(S))`, `intruder` is a constant of `Prin` denoting the intruder. The equations say that if `c-fkm11(S,P,Q,E1)` is true, then the Init message `m1(intruder,P,Q,E1)` is put into the network `nw(S)`, `ur(S)` does not change in the state denoted by `fkm11(S,P,Q,E1)`; if `c-fkm11(S,P,Q,E1)` is false, nothing changes. The remaining transitions can be defined likewise.

7.4 Formal verification of NSLPK with CiMPG

There are two properties of NSLPK that we would like to verify namely nonce secrecy property and one-to-many correspondence property. The former says that all nonces available to the intruder are those created by the intruder or those created for the intruder. Let `N` be a CafeOBJ variable of `Nonce`, respectively, we specify the nonce secrecy property as follows:

```
eq inv130(S,N) = (N \in cnonce(nw(S))
implies (creator(N) = intruder or forwhom(N) = intruder)) .
```

The one-to-many correspondence property is specified by the following two equations:

```
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)
implies m3(P,P,Q,enc3(Q,n(Q,P,R))) \in nw(S)) .
```

where `P1` & `Q1` are CafeOBJ variables of `Prin`, `R` is a CafeOBJ variable of `Rand`. `inv170` says that whenever `P` successfully sent an Init message to `Q`, and received a corresponding Resp seemingly from `Q`, the principal that `P` is communicating with is really `Q` even though there are malicious principals (e.g., `Q1`). `inv180` can be understood likewise.

To verify the nonce secrecy property, we prove that `inv130` is an invariant of the OTS formalizing NSLPK. The formal verification is also conducted in two ways: by writing proof scripts with CiMPA and by using CiMPG to generate proof scripts from proof scores. Both of them require the use of the following lemmas:

```
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 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) \in cnonce(nw(S))) .

eq inv160(S,N) = (creator(N) = intruder
implies N \in cnonce(nw(S))) .
```

where `E2` and `E3` are CafeOBJ variables of `Cipher2` and `Cipher3`, respectively.

In each way of verification, what we need to do is quite similar to what we have described in the last section with formal verification of IFF. However, with CiMPG, we also need to make some modifications to the existing proof scores. Let us consider an example in which we want to split the current case into two sub-cases: (1) message `m` is in `nw(s)`, which is the network of the current state, and (2) `m` is not in `nw(s)`. CafeOBJ allows us to

write proof scores to conduct case splitting by introducing two equations: (i) `nw(s) = (m , nw')` to characterize (1) and (ii) `m \in nw(s) = false` to characterize (2), where `nw'` is a constant denoting an arbitrary network (or list of messages).

With CiMPA, if we declare equation (i) and apply for case splitting, then it will automatically split the current goal into two sub-goals in which (i) holds in the first sub-goal, while it does not hold in the second one. Thus, the second sub-goal is characterized by the equation `(nw(s) = (m , nw')) = false`. In this sub-goal, it does not guarantee that `m` is not in `nw(s)` since `m` can be in `nw'`. CiMPG cannot be used for this case. In the existing proof scores of formal verification of NSLPK, there are many times in which case splitting is "flexibly" applied in the same way as based on two equations (i) and (ii) mentioned above. This flexible case splitting is an advantage of the CafeOBJ/proof score method but also is a disadvantage because we need to ensure that the equations used for case splitting cover every case and do not overlap each other. However, to make it possible for CiMPG to generate the proof scripts, the existing proof score needs to be modified. With the example mentioned above, two equations used for case splitting should be `m \in nw(s) = true` and `m \in nw(s) = false`.

For NSLPK case study, CiMPG took about 12 hours to generate the complete proof scripts and CiMPA took about 50 minutes to discharge all goals.

Chapter 8

Lessons learned

This chapter describes what we have learned throughout the research thesis. Specifically, we have learned a particular class of authentication protocols in Section 8.1. Lessons learned from graphical animations of authentication protocols are presented in Section 8.2. What we have learned about proof scores, proof scripts, proof assistant CiMPA, and proof generator CiMPG when we conduct the formal verification, are described in the last Section 8.3.

8.1 Authentication protocols

We were able to learn about the authentication protocols that we had never learned in-depth before. We were able to understand/comprehend that how important the security of authentication protocols are, how they work, which security properties they should enjoy and what threats exist in the communication network. From which we were able to deepen our understanding of what secure communication is.

8.2 Graphical animations of authentication protocols

We were able to learn SMGA, how to design state pictures and how to observe graphical animations of state pictures to guess the characteristics. Designing state pictures is the most intellectual task in graphical animations with SMGA because state picture design affects how we can observe the behavior of the protocol well. We have confirmed that all lemmas used in the formal verification of the two protocols can be discovered by observing graphical animations based on our state picture designs which demonstrates that SMGA has potential to help human users conjecture lemmas needed to conduct formal proofs. We found out some useful and practical tips on designing state pictures of authentication protocols. The tips are summarized as follows:

1. Understand that how the protocol works, how many message exchanges are included, which components are in each message.
2. If the formats of message exchanged are not the same, we have to consider that if it is required to make the formalization of the messages consistent in state picture and if it is possible to do so. If it is not the case, we have to consider how to visualize the different format of messages.
3. Carefully choose the observable components which will be visualized in the state picture.
4. Placement of observable components is also important because it helps us to quickly recognize some relations on the values of multiple observable components. Displaying the observable components that have relations between them nearby together in the stat picture helps us observe the graphical animations easily. For example, `receiver` and `Public-key` in the state picture.
5. State picture design should be visualized as much as possible so that users can visually perceive the value of each observable component and some relations among the values of observable components.
6. If an observable component can have one or more values, for example, `latest` message has three values ($m1$, $m2$ or $m3$). We should prepare some designated area and a specific position in the area for each value. If the observable component has a value, only the visual object for it should be displayed and the other visual objects for the other values should disappear. For example, when the latest message is a message `m2`, there is only one light gray square displayed under m2 as shown in the following picture:

7. Designing sub-networks according to types of messages allows us to immediately recognize the specific message type in each specific network. It makes us more transparent in our visual perception when we observe each specific message or the order/relation between messages.

After we design the state picture for authentication protocols, we prepare finite state sequence input files and observe the graphical animations with SMGA to guess the characteristics of protocols. Let us repeat some useful and practical tips described in the section 6.3 on how to conjecture characteristics of the protocols by observing graphical animations as follows:

1. By concentrating on one observable component, we may find that if the value of that observable component is `intr`, any other observable com-

ponents may have some specific values, from which we may conjecture some characteristics.
2. By concentrating on two different observable components, we may find a relation between them, from which we may conjecture some characteristics.
3. By observing the order of the message in the network, we may find a relation between them, from which we may conjecture some characteristics.
4. By carefully investigating the conditions of some characteristics that have been already conjectured, from which we may conjecture some other characteristics.

Observing such graphical animations allows us to guess the characteristics of authentication protocols. We could observed the lemmas used to conduct formal proofs. However, the characteristics guessed are required to be confirmed whether or not the protocol surely enjoys those characteristics. Hence, we were able to learn Maude search command to confirm the guessed characteristics of the protocols.

8.3 Proof scores, Proof scripts, CiMPA and CiMPG

Model checking is convenient, however, it does not guarantee the correctness of the characteristics in all possible situations because of the state explosion problem. We could confirm the characteristics at a specific depth 5 in a reasonable time. Although some model checking experiments were not completed because of the state space explosion problem, some characteristics of NSLPK have been proved [16], guaranteeing that the characteristics are invariant properties with respect to the state machine formalizing NSLPK protocol. We were able to rer-conduct the formal verification of NSLPK (and IFF) using CiMPA and CiMPG. We were able to deepen our understanding of how to formally specify authentication protocols in CafeOBJ and Maude specification languages, how to write proof scores and how to revise proof scores to use in CafeInMaude. We were able to learn what CiMPA and CiMPG are, how to use them, and the advantages and disadvantages of each method. We were able to double-check the correctness of the properties and other characteristics (invariant properties used as lemmas in formal verification) of the two protocols using CiMPA and CiMPG.

We were able to understand a difference between the results of formal verification using the same proof scores with CafeOBJ and CafeInMaude.

The latter may return false even if the former returns true. This is because CafeOBJ used the equality rewrite rule in which left and right are distinguished). Based on the fact that rewriting in CafeInMaude can be done smoothly by paying attention to the case splitting method and rewrite rules. From which we understand the advantages and disadvantages of CafeOBJ and CafeInMaude.

Once we have completely conducted the formal verification by proof scores, it is rather straightforward to develop the verification with CiMPG. CiMPG takes time to automatically generate proof scripts for CiMPA. Although CiMPG can automatically generate proof scripts for CiMPA, we found out that any proof scores cannot be converted into proof scripts even if the proof score is correct. Let us repeat the following example: (i) `nw(s) = (m , nw')` and (ii) `m \in nw(s) = false` in which CiMPG cannot generate proof script correctly. Two equations used for case splitting should be `m \in nw(s) = true` and `m \in nw(s) = false`. Therefore, there are many times in which case splitting is "flexibly" applied in the same way as based on two equations (i) and (ii) mentioned above. One more thing about CiMPG is that it cannot generate proof scripts from proof scores that use trivial lemmas. If that is the case, we need to write the manual proof script for proof score using trivial lemmas.

In this thesis, we have confirmed that all characteristics found by observing graphical animations are the lemmas needed to complete the formal verification. It demonstrates that SMGA has potential to help human users conjecture lemmas needed to conduct formal proofs and how effectively use SMGA and CiMPG(and CiMPA) together to complete the formal verification of the two protocols.

Chapter 9

Related work

Bui, et al. [4] have graphically animated a distributed mutual exclusion protocol called Suzuki-Kasami protocol with SMGA. Messages are exchanged in the network and the network can be very huge when many messages have been put into the network. The authors have found that the messages that have been put into the network and have been deleted from the work are crucial information. Their proposed solution is to prepare two places for containing those messages. In case that the size network is larger, their approach is to display the incoming messages to the text "...". Their solutions help them guess some characteristics/properties of the protocol, and are summarized as tips to design the state picture for the distributed protocol. Their research and ours can share working flow. We, however, cannot apply those tips to our work because of the different architecture of the types of protocol.

ShiViz [17] has been developed to visualize logs generated by distributed systems. Logs in this context are sequences of events, hosts that carry out the events, and timestamps when the hosts carried out the events. Message sending and receiving are considered important events. Based on the log as an input of ShiViz, ShiViz generates a diagram that is similar to a sequence diagram. The authors have tried to demonstrate that their visualization helps human users comprehend what events exchange and/or succeed to what events., some patterns of message passings, etc. One functionality of ShiViz is to find three typical patterns of message passings: Request-Response, Broadcast, and Gather. The visualization used by ShiViz (still visualization) and the one (graphical animation) used by SMGA can be complementary. One possible future direction is to find a well-balanced combination of the two approaches to (authentication) system visualization.

VA4JVM [18] is a tool that can visualize outputs generated by Java Pathfinder (JPF). JPF outputs are often long and hard to read, especially when JPF finds something wrong, such as race-condition and deadlock. VA4JVM supports some functionalities, such as zooming, filtering, highlighting some specific parts of JPF outputs. Those functionalities can help human users observe some fragments that look interesting to be able to better

comprehend JPF outputs. Counterexamples can be graphically animated by SMGA in which they are generated by Maude LTL model checker [19]. Although Maude LTL model checker is a classical model checker and JPF is a software model checker, it would be worth considering some VA4JVM functionalities, such as zooming, filtering, and highlighting, to apply them to the future version of SMGA.

Souri, et al. [20] have presented Graphical Symbolic Modeling Toolkit (GSMT) to design and verify the behavioral models of distributed systems. It consists of the behavioral modeling in form of Kripke structure (KS) and Labeled Transition System (LTS), generating a graphical state exploration diagram of the behavioral model, generating the expected specification rules automatically, translating the behavioral model to the SMV codes, and reduction of the state space. The important functionality of the GSMT is the implementation of the syntactic reduced approach that ameliorates the state space explosion. It uses NuSMV symbolic model checker for evaluating the constructed temporal logic formulas. In our work, Maude LTL model checker can be used to generate counterexamples from which the counterexamples can be animated using SMG. However, the only requirement of the LTL model checker is the finiteness of the reachable states and the reachable state spaces of our case studies are infinite. Hence, their recursive reduced model is worth considering as one possible approach to reduce the state space of the system.

In [21], the authors analyzed Needham-Schroeder Public Key Protocol. The protocol is modeled using Promela. They found a well-known attack successfully, but they informally illustrated the procedure to construct the intruder model. In [22], the authors proposed a general method to model the intruder's behavior. In their method, the intruder can analyze all the messages intercepted by itself dynamically and make responses correspondingly. Although the method works well, the redundant model is vulnerable to the state space explosion.

The paper [23] used the model checking method to formally verify security protocols using Spin model checker. The paper exampled NSPK protocol and DS protocol. The experimental results showed that the number of the protocol model states has been decreased by a wide margin, and the efficiency of protocol verification has been improved properly. The security protocol models mainly include the protocol instance model and the intruder model. It is described with Promela Model. The property of security protocols is described using LTL. An attack is found in each protocol. The difference between their method and our method is that we use OTS/CafeOBJ method to formalize the protocols as state machines. The Dolev-Yao intruder model is used in both methods. Furthermore, this paper only verified the authentication property of security protocols while

we verified authentication and secrecy properties.

Riesco, et al. [24] have implemented CafeInMaude. In that paper, the author has shown the improvement of the performance of some commands through several case studies with CafeOBJ specifications. The experiments illustrate that some analyses that could not be performed in CafeOBJ become possible in CafeInMaude. Riesco, et al. [11] continued to develop two extension tools CiMPA and CiMPG. In that paper, the algorithm behind CiMPA and CiMPG is presented and Qlock protocol which is an abstract version of Dijkstra's binary semaphore has been used as a case study to demonstrate how to use CiMPA and CiMPG. The paper [11] also shows the performance of CiMPG in generating proof scripts from the correct proof scores for some protocols.

Palombo, et al. [25] have formally analyzed several examples of a particular class of cryptographic protocols (NSPK, DS, TMN, and DH) using three alternative tools. They used SPIN to analyze explicit state models written in Promela; symbolic models in the pi-calculus were verified using Proverif, and finally, we proved lemmas that show protocol insecurity using the theorem prover Coq. They have presented that choosing the right tool always implies a trade-off. It is generally easier to model a system using SPIN's state representation than using the pi-calculus or logical inference rules. However, SPIN can only verify a bounded number of sessions and message space. Proverif verifies an unbounded number of sessions and message space at the cost of potentially not terminating or reporting false attacks. On the other hand, Coq theories with security proofs are not bounded and can represent an unrestricted symbolic attacker. They have shown that protocol insecurity can be proved in Coq. On the other hand, proof that demonstrates that the properties hold may be harder to construct. In comparison with our research, we proved lemmas that show protocol security using proof assistant CiMPA and proof generator CiMPG.

In addition to CafeOBJ, several languages allow us to conduct formal verification. A well-known formal specification and verification language is ACL2 [26]. It has been successfully used in many formal verification problems. However, it does not seem as flexible as CafeOBJ. For example, CafeOBJ used mix-fix syntax, whereas ACL2 allows only pre-fix expression. Lieno [27] has developed a language and verifier called Dafny. The author illustrated its features through a case study. Dafny is dedicated to formal verification of programs while our research is dedicated to formal verification of designs or specifications.

Chapter 10

Conclusion

This thesis has presented the graphical animation of a particular class of authentication protocols using SMGA and formal verification of the protocols with CiMPG (and CiMPA). We have summarized some lessons learned through IFF and NSLPK case studies. We have suggested useful and practical tips on how to design state pictures and how to conjecture the characteristics of the protocols. The most intellectual task of this study is to design state pictures because guessing characteristics largely rely on state picture design. It is presented in Chapter 5 and Chapter 6. Our state picture design allows us to confirm that all lemmas used in the formal verification of the two protocols can be discovered by observing graphical animations which demonstrates the usefulness of SMGA to help human users conjecture lemmas to conduct formal proofs. The characteristics discovered are confirmed by Maude search command at a specific depth.

However, it does not guarantee that two authentication protocols surely enjoy the desired properties because standard model checking techniques cannot handle state machines in which there are an arbitrary number of entities, such as principle. We have conducted the formal verification of two protocols with proof assistant CiMPA and with proof generator CiMPG, which is described in Chapter 7. Through the two case studies of formal verification, we have summarized some lessons learned. Once we have complete proof scores, it is rather straightforward to manually write proof scripts for CiMPA. We need to minimally annotate proof scores to use CiMPG. Although CiMPG can automatically generate proof scripts for CiMPA from annotated proof scores, it will take time to do so when the size of the input proof score is large. However, the time taken to generate the proof scripts for the two protocols is – here On the other hand, writing manually proof scripts will not be a good idea when the size of proof score is large since it tends to cause human errors.

In comparison with the proof score approach, each verification method has advantages as well as disadvantages. While proof scores are flexible to write, they are subject to human errors since human users can overlook some cases during the verification. The proof scripts are reliable, but they are

not easy to develop for non-expert users. CiMPG combines the flexibility of the proof score approach and the reliability of CiMPA. We have conducted the two case studies to confirm the effectiveness of CiMPG (and CiMPA). However, it often takes time for CiMPG to generate proof scripts when the size of input proof scores is large. In the two case studies, we have formally verified that IFF protocol enjoys the identifiable property, and NSLPK enjoys the nonce secrecy and one-to-many correspondence (authentication) properties.

One piece of our future work is to graphically animate state machines that formalize other authentication protocols such as TLS [28] to demonstrates the usefulness SMGA. Another piece of our future work is to improve SMGA tool so that it can be more interactive by supporting some more interesting features such as: zooming in/out, or pattern matching by integrating SMGA into Maude. It will be useful if we can develop a tool that can revise CafeOBJ proof scores to make them obey the syntax of CafeInMaude. We also aim to come up better annotations to proof scores for CiMPG to efficiently and correctly generate proof scripts from annotated proof scores in a reasonable amount of time even if the size of the proof score is huge. And in the future, we will verify more authentication protocols to confirm the effectiveness of CiMPG.

References

[1] T. T. T. Nguyen and K. Ogata, "Graphical animations of state machines," in *15th DASC*, 2017, pp. 604–611.

[2] M. T. Aung, T. T. T. Nguyen, and K. Ogata, "Guessing, model checking and theorem proving of state machine properties – a case study on Qlock," *IJSECS*, vol. 4, no. 2, pp. 1–18, 2018.

[3] T. T. T. Nguyen and K. Ogata, "Graphically perceiving characteristics of the MCS lock and model checking them," in *7th SOFL+MSVL*, 2017, pp. 3–23.

[4] D. D. Bui and K. Ogata, "Better state pictures facilitating state machine characteristic conjecture," in *DMSVIVA 2020*, 2020, pp. 7–12.

[5] ——, "Graphical animations of the Suzuki-Kasami distributed mutual exclusion protocol," *JVLC*, vol. 2019, no. 2, pp. 105–115, 2019.

[6] K. W. Brodlie, et al., Ed., *Scientific Visualization: Techniques and Applications*. Springer, 1992.

[7] M. Clavel, et al., Ed., *All About Maude*, ser. LNCS. Springer, 2007, vol. 4350.

[8] K. Ogata and K. Futatsugi, "Proof scores in the OTS/CafeOBJ method," in *FMOODS 2003*, 2003, pp. 170–184.

[9] R. Diaconescu and K. Futatsugi, *Cafeobj Report*, ser. AMAST Series in Computing. World Scientific, 1998, vol. 6.

[10] A. Riesco, K. Ogata, and K. Futatsugi, *CafeInMaude: A CafeOBJ Interpreter in Maude*, ser. Lecture Notes in Computer Science, S. P. and W. A, Eds. Springer, Berlin, Heidelberg, 2016, vol. 9633.

[11] A. Riesco and K. 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.

[12] R. J. Anderson, *Security engineering - A guide to building dependable distributed systems*. Wiley, 2001.

[13] R. M. Needham and M. D. Schroeder, "Using Encryption for Authentication in Large Networks of Computers," *Commun. ACM*, vol. 21, no. 12, pp. 993–999, 1978.

[14] G. Lowe, "An Attack on the Needham-Schroeder Public-Key Authentication Protocol," *Inf. Process. Lett.*, vol. 56, no. 3, pp. 131–133, 1995.

[15] S. FUJII, "An investigation of formal verification of authentication protocols with proof score and a new case study," Master's Research Project Report, Japan Advanced Institute of Science and Technology, 2021.

[16] K. Ogata and K. Futatsugi, "Rewriting-based verification of authentication protocols," *ENTCS*, vol. 71, pp. 208–222, 2002.

[17] I. Beschastnikh, et al., "Visualizing distributed system executions," *ACM TOSEM.*, vol. 29, no. 2, pp. 9:1–9:38, 2020.

[18] C. Artho, et al., "Visualization of concurrent program executions," in *31st COMPSAC*, 2007, pp. 541–546.

[19] T. T. T. Nguyen and K. Ogata, "A way to comprehend counterexamples generated by the Maude LTL model checker," in *SATE 2017*, 2017, pp. 53–62.

[20] A. Souri, A. M. Rahmani, N. J. Navimipour, and R. Rezaei, "A symbolic model checking approach in formal verification of distributed systems," *Human-centric Computing and Information Sciences*, vol. 9, no. 1, p. 4, 2019.

[21] P. Maggi and R. Sisto, "Using spin to verify security properties of cryptographic protocols," in *In LNCS*. Springer-Verlag, 2002, pp. 187–204.

[22] N. Ben Henda, "Generic and efficient attacker models in spin," in *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, ser. SPIN 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 77–86. [Online]. Available: https://doi.org/10.1145/2632362.2632378

[23] S. Chen, H. Fu, and H. Miao, "Formal verification of security protocols using spin," in *2016 IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS)*, 2016, pp. 1–6.

[24] A. Riesco, K. Ogata, and K. Futatsugi, "A maude environment for cafeobj," *Form. Asp. Comput.*, vol. 29, no. 2, p. 309–334, Mar. 2017. [Online]. Available: https://doi.org/10.1007/s00165-016-0398-7

[25] H. Palombo, "A comparative study of formal verification techniques for authentication protocols," 2015.

[26] S. Ray, *Scalable Techniques for Formal Verification*, 1st ed. Springer, Boston, MA.

[27] K. Leino, "Dafny: An automatic program verifier for functional correctness," in *LPAR 2010*, ser. Lecture Notes in Computer Science, V. A. Clarke E.M., Ed., vol. 6355, 2010.

[28] T. Dierks and C. Allen, "The TLS protocol version 1.0," *RFC*, vol. 2246, pp. 1–80, 1999.

Publications

[1] Thet Wai Mon, Shuho Fujii, Duong Dinh Tran, and Kazuhiro Ogata. Formal verification of IFF & NSLPK authentication protocols with CiMPG. In *The 33rd International Conference on Software Engineering and Knowledge Engineering, SEKE 2021, KSIR Virtual Conference Center, Pittsburgh, USA, July 1-10, 2021*, pages 120-125. KSI Research Inc. and Knowledge Systems Institute Graduate School, 2021.

[2] Thet Wai Mon, Dang Duy Bui, Duong Dinh Tran, and Kazuhiro Ogata. Graphical animations of NSLPK authentication protocol. In *The 27th International DMS Conference on Visualization and Visual Languages, DMSVIVA 2021, KSIR Virtual Conference Center, Pittsburgh, USA, June 29-30, 2021.*