URL: http://www.elsevier.nl/locate/entcs/volume71.html 15 pages

Rewriting-Based Verification of Authentication Protocols

Kazuhiro Ogata [1]

NEC Software Hokuriku, Ltd.
and
Japan Advanced Institute of Science and Technology (JAIST)

Kokichi Futatsugi [2]

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

Abstract

We propose a method of formally analysing security protocols based on rewriting. The method is roughly as follows. A security protocol is modeled as an observational transition system, which is described in CafeOBJ. Proof scores showing that the protocol has safety (security) properties are then written in CafeOBJ and the proof scores are executed (rewritten) by the CafeOBJ system.

1 Introduction

Security protocols such as authentication ones are key technology if we exchange messages secretly and/or authentically over an open network such as the Internet. But, they are subject to subtle faults that are especially difficult to find by testing and usual operation. Even if cryptosystems used are hard to break, there could be attacks to break security protocols that are seemingly well designed such as Lowe's attack[15] to the Needham-Schroeder Public-Key authentication protocol (the NSPK protocol)[18]. Therefore, several methods of formally analysing security protocols have been proposed[2,4,9,16,21,22].

In this paper, we propose a method of formally analysing security protocols based on rewriting. The method is roughly as follows. A security protocol is modeled as an observational transition system[19,20], which is described in CafeOBJ[1,5]. Proof scores showing that the protocol has safety (security)

[1] Email: ogatak@acm.org
[2] Email: kokichi@jaist.ac.jp

properties are then written in CafeOBJ and the proof scores are executed (rewritten) by the CafeOBJ system. The CafeOBJ system can be used as an interactive proof-checker or verifier on several levels[10]. In the proposed method, the CafeOBJ system is used as proof score executor. The NSPK protocol corrected by Lowe[15] is used to show our method.

The rest of the paper is organized as follows. Section 2 mentions CafeOBJ. Observational transition systems and a way of describing them in CafeOBJ are written in Sect 3. Section 4 gives a brief description of the NSPK protocol corrected by Lowe. Section 5 describes the observational transition system modeling the protocol and its specification in CafeOBJ. Section 6 shows (part of) the proof that the protocol has a safety property. Section 7 gives related work, and we conclude the paper in Sect 8.

2 CafeOBJ in a Nutshell

CafeOBJ[1,5] is mainly based on two logical foundations: *initial* and *hidden* algebra. Initial algebra is used to specify abstract data types such as integers, and hidden algebra[6,11] to specify abstract machines. There are two kinds of sorts (corresponding to types in programming languages) in CafeOBJ. They are *visible* and *hidden* sorts. A visible sort represents an abstract data type, and a hidden sort the state space of an abstract machine. There are basically two kinds of operations to hidden sorts. They are *action* and *observation* operations. An action operation can change a state of an abstract machine. It takes a state of an abstract machine and zero or more data, and returns another (possibly the same) state of the abstract machine. Only observation operations can be used to observe the inside of an abstract machine. An observation operation takes a state of an abstract machine and zero or more data, and returns a value corresponding to the state. An action operation is basically specified with equations by describing how the value of each observation operation changes relatively based on the values of observation operations in a state after executing the action operation in the state.

Declarations of visible sorts are enclosed with `[` and `]`, and those of hidden ones with `*[` and `]*`. Declarations of observation and action operations start with `bop` or `bops`, and those of other operations with `op` or `ops`. After `bop` or `op` (or `bops` or `ops`), an operator is written (or more than one operator is written), followed by `:` and a sequence of sorts (i.e. sorts of the operators' arguments), and ended with `->` and one sort (i.e. sort of the operators' results). Definitions of equations start with `eq`, and those of conditional ones with `ceq`. After `eq`, two expressions, or terms connected by `=` are written, ended with a full stop. After `ceq`, two terms connected by `=` are written, followed by `if` and a term denoting a condition, and ended with a full stop.

The CafeOBJ system, an implementation of CafeOBJ, rewrites (reduces) a given term by regarding equations as left-to-right rewrite rules. This executability makes it possible to simulate described systems and to verify that

they possess some desired properties.

3 Observational Transition Systems

We assume that there exists a universal state space called Υ. When we describe a system, the system is basically modeled by observing only quantities that are relevant to the system and that interest us from the outside of each state of Υ. An observational transition system (OTS)[19,20] can be used to model a system in this way. UNITY[3] is an ancestor of OTS's, which are reformalized by adopting the concept of hidden algebra[6,11].

An OTS $\mathcal{S} = \langle \mathcal{O}, \mathcal{I}, \mathcal{T} \rangle$ consists of:

- $\mathcal{O}$: A set of observations. Each observation $o \in \mathcal{O}$ is a function $o : \Upsilon \rightarrow D$ mapping each $v \in \Upsilon$ into some typed value in D (D may be different for each observation). The value returned by an observation (in a state) is called the value of the observation (in the state).

  Given an OTS $\mathcal{S}$ and two states $v_1, v_2 \in \Upsilon$, the equality between two states, denoted by $v_1 =_{\mathcal{S}} v_2$, with respect to $\mathcal{S}$ is defined as follows:

  $$v_1 =_{\mathcal{S}} v_2 \text{ iff } \forall o \in \mathcal{O}. o(v_1) = o(v_2),$$

  where '=' in $o(v_1) = o(v_2)$ is supposed to be well defined for the range of each $o \in \mathcal{O}$. $\mathcal{S}$ may be removed from $=_{\mathcal{S}}$ if it is clear from the context.
- $\mathcal{I}$: The initial condition. This condition specifies the initial value of each observation that defines initial states of the OTS.
- $\mathcal{T}$: A set of conditional transition rules. Each transition rule $\tau \in \mathcal{T}$ is a relation between states provided that, for each state $v \in \Upsilon$, there exists a state $v' \in \Upsilon$, called a successor state, such that $\tau(v, v')$ and moreover, for each state $v_1, v_2, v_1', v_2' \in \Upsilon$ such that $v_1 =_{\mathcal{S}} v_2$, $\tau(v_1, v_1')$ and $\tau(v_2, v_2')$, $v_1' =_{\mathcal{S}} v_2'$. τ can be regarded as a function on equivalent classes of Υ with respect to $=_{\mathcal{S}}$. Therefore, we assume that $\tau(v)$ denotes the representative element of the equivalent class the successor states of v with respect to τ belong to, and $\tau(v)$ is called *the* successor state of v with respect to τ.

  The condition c_τ for a transition rule $\tau \in \mathcal{T}$ is called the *effective condition*. Given a state, its truth value can be determined by only the values of observations in the state. Predicates of this kind are called *state predicates*. Given a state $v \in \Upsilon$, c_τ is true in v, namely τ is *effective* in v, iff $v \neq_{\mathcal{S}} \tau(v)$.

Multiple similar observations or transition rules may be indexed. Generally, observations and transition rules are denoted by $o_{i_1,\ldots,i_m}$ and $\tau_{j_1,\ldots,j_n}$, respectively, provided that $m, n \geq 0$ and we assume that there exist data types D_k such that $k \in D_k$ ($k = i_1, \ldots, i_m, j_1, \ldots, j_n$). For example, an integer array a possessed by a process p may be denoted by an observation a_p, and the increment of the ith element of the array may be denoted by a transition rule $inca_{p,i}$.

Given an OTS, a set of infinite sequences of states is obtained. The infinite

sequence of states is called an execution of the OTS. More specifically, an execution of an OTS $\mathcal{S}$ is an infinite sequence $s_0, s_1, \ldots$ of states satisfying:

- *Initiation*: For each $o \in \mathcal{O}$, $o(s_0)$ satisfies $\mathcal{I}$.
- *Consecution*: For each $i \in \{0, 1, \ldots\}$, $s_{i+1} =_{\mathcal{S}} \tau(s_i)$ for some $\tau \in \mathcal{T}$.
- *Fairness*: For each $\tau \in \mathcal{T}$, there exist an infinite number of indexes $i \in \{0, 1, \ldots\}$ such that $s_{i+1} =_{\mathcal{S}} \tau(s_i)$.

A state is called *reachable* with respect to $\mathcal{S}$ if it appears in an execution of $\mathcal{S}$.

Important properties that an OTS may have are basically classified into two classes: *safety* and *liveness* (or *progress*) properties. We only describe safety properties and how to prove that an OTS has a safety property in this paper. Safety properties are defined as follows: a predicate $p : \Upsilon \to \{\text{true}, \text{false}\}$ is a safety property with respect to $\mathcal{S}$ iff p is a state predicate and $p(v)$ holds for every reachable $v \in \Upsilon$.

If we prove that an OTS has a safety property p, the following induction is mainly used:

- Base case: For any state $v \in \Upsilon$ in which each observation $o \in \mathcal{O}$ satisfies $\mathcal{I}$, we show that $p(v)$ holds.
- Inductive step: Given any reachable state $v \in \Upsilon$ such that $p(v)$ holds, we show that, for any transition rule $\tau \in \mathcal{T}$, $p(\tau(v))$ also holds.

An OTS $\mathcal{S}$ is described in CafeOBJ. The universal state space Υ is denoted by a hidden sort, say `Sys`, by declaring `*[Sys]*`.

An observation $o_{i_1,\ldots,i_m} \in \mathcal{O}$ is denoted by a CafeOBJ observation operation. We assume that data types $D_k\,(k = i_1, \ldots, i_m)$ and D are described in initial algebra and there exist visible sorts $S_k\,(k = i_1, \ldots, i_m)$ and S corresponding to the data types. The CafeOBJ observation operation denoting $o_{i_1,\ldots,i_m}$ is declared as follows:

```
bop o : Sys S_{i_1} ... S_{i_m} -> S
```

The initial condition $\mathcal{I}$, the value of each observation in any initial state, is described by declaring a constant (an operator without any arguments) denoting any initial state and specifying the value of each observation in the state with equations. First, the constant `init` denoting any initial state is declared as follows:

```
op init : -> Sys
```

Suppose that the initial value of $o_{i_1,\ldots,i_m}$ is $f(i_1, \ldots, i_m)$, this can be described in CafeOBJ as follows:

```
eq o(init,X_{i_1},...,X_{i_m}) = f(X_{i_1},...,X_{i_m}) .
```

where `X`$_k$ $(k = i_1, \ldots, i_m)$ is a CafeOBJ variable with `S`$_k$, and `f(X`$_{i_1}$`,...,X`$_{i_m}$`)` means a term (consisting of `X`$_{i_1}, \ldots,$ `X`$_{i_m}$) corresponding to $f(i_1, \ldots, i_m)$.

A transition rule $\tau_{j_1,\ldots,j_n} \in \mathcal{T}$ is denoted by a CafeOBJ action operation. We assume that data types $D_k\,(k = j_1, \ldots, j_n)$ are described in initial algebra

and there exist visible sorts $S_k\,(k = j_1, \ldots, j_n)$ corresponding to the data types. The CafeOBJ action operation denoting $\tau_{j_1,\ldots,j_n}$ is declared as follows:

```
bop a : Sys S_j1 ... S_jn -> Sys
```

If $\tau_{j_1,\ldots,j_n}$ is executed in a state in which it is effective, the value of $o_{i_1,\ldots,i_m}$ may be changed, which can be described in CafeOBJ generally as follows:

```
ceq o(a(S,X_j1,...,X_jn),X_i1,...,X_im)
    = e-a(S,X_j1,...,X_jn,X_i1,...,X_im) if c-a(S,X_j1,...,X_jn) .
```

where `e-a(S,`X_{j_1}`,...,`X_{j_n}`,`X_{i_1}`,...,`X_{i_m}`)` means a term (consisting of `S,`$X_{j_1}, \ldots,$ $X_{j_n}, X_{i_1}, \ldots, X_{i_m}$) corresponding to the value of $o_{i_1,\ldots,i_m}$ in the successor state, and `c-a(S,`X_{j_1}`,...,`X_{j_n}`)` means a term (consisting of `S,`$X_{j_1}, \ldots, X_{j_n}$) corresponding to $c_{\tau_{j_1,\ldots,j_n}}$.

If $\tau_{j_1,\ldots,j_n}$ is executed in a state in which it is not effective, the value of any observation is not changed. Therefore, all we have to do is to declare the following equation:

```
ceq a(S,X_j1,...,X_jn) = S if not c-a(S,X_j1,...,X_jn) .
```

If the value of $o_{i_1,\ldots,i_m}$ is not affected by executing $\tau_{j_1,\ldots,j_n}$ in any state (regardless of the truth value of $c_{\tau_{j_1,\ldots,j_n}}$), the following equation is declare:

```
eq o(a(S,X_j1,...,X_jn),X_i1,...,X_im) = o(S,X_i1,...,X_im) .
```

4 The NSLPK Protocol

Needham and Schroeder[18] proposed an authentication protocol, called the NSPK protocol, with public-key cryptosystems in 1978. Lowe[15] found out 17 years later that there was a serious attack on the protocol that an intruder could impersonate another agent to establish a session with yet another agent. He also proposed one possible correction, which is called the NSLPK protocol in this paper.

The NSLPK protocol uses public-key cryptosystems in order to establish mutual authentication between two principals. For each principal p, there is a public key denoted by $k(p)$, which any principal can obtain from a key server. Each principal p also has a private key that is the inverse of $k(p)$. A message m encrypted with a public key k is denoted by $\{m\}_k$. Any principal can encrypt a message m with p's public key to generate $\{m\}_{k(p)}$, while only p can decrypt this message, which ensures secrecy. The protocol also uses nonces that can be represented by large random numbers.

The NSLPK protocol could be described as follows:

$$\begin{array}{lllll} \text{Message 1} & p_1 \to p_2 & : & p_1.p_2.\{n_{p_1}.p_1\}_{k(p_2)} \\ \text{Message 2} & p_2 \to p_1 & : & p_2.p_1.\{n_{p_1}.n_{p_2}.p_2\}_{k(p_1)} \\ \text{Message 3} & p_1 \to p_2 & : & p_1.p_2.\{n_{p_2}\}_{k(p_2)} \end{array}$$

p_1 is an initiator that tries establishing a session with a responder p_2. p_1 starts a run of the protocol by creating a nonce n_{p_1} and sending it along with its identity to p_2, encrypted with p_2's public key. This kind of messages are called messages of type 1. When p_2 receives the message, it decrypts the message with its private key to obtain the nonce n_{p_1}. It then returns n_{p_1} along with a new nonce n_{p_2} as well as its identity to p_1, encrypted with p_1's public key. This kind of messages are called messages of type 2. When p_1 receives the message, it decrypts the message with its private key to confirm that the message contains n_{p_1} and p_2, which should make p_1 assured that p_1 is talking to p_2 because only p_2 should be able to decrypt the message of type 1 to obtain n_{p_1}. p_1 then returns n_{p_2} to p_2, encrypted with p_2's public key. This kind of messages are called messages of type 3. Receiving the message should make p_2 assured that p_2 is talking to p_1 because only p_1 should be able to decrypt the message of type 2 to obtain n_{p_2}.

The first and second fields of messages are called the source and destination fields respectively. The last field of encrypted parts of messages of type 1 and 2 is called the principal field.

5 Modeling

Let us model a system in which an arbitrary number of principals take part in the NSLPK protocol as an OTS. One of the principals is assumed to be an intruder. The intruder obeys the basic assumptions of the Dolev-Yao model[7]. All it can do illegally is enumerated as follows:

- It can intercept any message that is being delivered. If nonces included in the message are encrypted with the intruder's public key, the nonces are gleaned, and otherwise the message is stored as it is.
- It can make fake messages of nonces or messages that are kept in store. If a fake message is made of nonces, the nonces are encrypted with any principal's public key and any other field of the message is filled with any principal. If it is made of a message, only the source and destination fields are changed with any principal.

The following assumption on nonce creation is also used:

Nonce Creation Every time a principal creates a nonce, the nonce is really fresh, which has never appeared in the system so far.

The following operations on messages are used in the rest of the paper: *isMsg1*, *isMsg2* and *isMsg3* are predicates checking if a message is type 1, type 2 and type 3, respectively, *getS* and *getD* return the source and destination fields of a message respectively, *getP* returns the principal field of a message if the message is either type 1 or type 2, *getK* returns the public key used in a message, *getN1* returns the (first) nonce of a message, and *getN2* returns the second nonce of a message if the message is type 2.

First the genuine parts of the system are modeled. For any pair of different principals p_1, p_2 and any role $r \in \{\mathrm{Ini}, \mathrm{Res}\}$, we have the following observations: $l_{p_1,p_2,r}$, $n1_{p_1,p_2,r}$ and $n2_{p_1,p_2,r}$. $l_{p_1,p_2,\mathrm{Ini}}$ is used for p_1 as initiator, having one of the four possible values i1, i2, i3 and i4, while $l_{p_1,p_2,\mathrm{Res}}$ is used for p_1 as responder, having one of the four possible values r1, r2, r3 and r4. If $l_{p_1,p_2,\mathrm{Ini}}$ is i1, i2, i3 and i4, p_1 as initiator is ready for starting a new run of the protocol with p_2, ready for receiving a message of type 2 from p_2, ready for sending a message of type 3 to p_2 and has a session with p_2, respectively. If $l_{p_1,p_2,\mathrm{Res}}$ is r1, r2, r3 and r4, p_1 as responder is ready for receiving a message of type 1 from p_2, ready for sending a message of type 2 to p_2, ready for receiving a message of type 3 from p_2 and has a session with p_2, respectively. $n1_{p_1,p_2,\mathrm{Ini}}$ and $n2_{p_1,p_2,\mathrm{Ini}}$ are used for p_1 so as to record the nonce created by p_1 and the nonce received from p_2 respectively when p_1 as initiator tries establishing or has a session with p_2 as responder. $n1_{p_1,p_2,\mathrm{Res}}$ and $n2_{p_1,p_2,\mathrm{Res}}$ are used for p_1 so as to record the nonce received from p_2 and the nonce created by p_1 respectively when p_1 as responder tries establishing or has a session with p_2 as initiator. Initially $l_{p_1,p_2,\mathrm{Ini}}$ is i1, $l_{p_1,p_2,\mathrm{Res}}$ is r1, and $n1_{p_1,p_2,r}$ and $n2_{p_1,p_2,r}$ are an arbitrary value that is never used as nonce.

We have two more observations: nw and n. nw denotes the underlying computer network connecting the principals. It is a multiset, or a bag of messages. n denotes the nonce created next. Initially nw is empty and n is an arbitrary value that can be used as nonce.

We have the transition rules shown in Table 1. $msgi+_{\{p_1,p_2\}}$ corresponds to that p_1 sends a message of type i to p_2, while $msgi-_{\{p_1,p_2,m\}}$ to that p_1 receives message m of type i sent by p_2, where $i \in \{1,2,3\}$. $end_{\{p_1,p_2,r\}}$ finishes a session of p_1 as r with p_2 as $\neg r$, where $\neg\mathrm{Ini} = \mathrm{Res}$ and $\neg\mathrm{Res} = \mathrm{Ini}$. Their effective conditions are shown in Table 1.

Table 1
Transition rules for any pair of different principals p_1, p_2 and any message m and their effective conditions.

Trans. rules	Effective conditions
$msg1+_{p_1,p_2}$	$l_{p_1,p_2,\mathrm{Ini}} = \mathrm{i1} \wedge p_1 \neq p_2$
$msg2+_{p_1,p_2}$	$l_{p_1,p_2,\mathrm{Res}} = \mathrm{r2} \wedge p_1 \neq p_2$
$msg3+_{p_1,p_2}$	$l_{p_1,p_2,\mathrm{Ini}} = \mathrm{i3} \wedge p_1 \neq p_2$
$msg1-_{p_1,p_2,m}$	$l_{p_1,p_2,\mathrm{Res}} = \mathrm{r1} \wedge p_1 \neq p_2 \wedge m \in nw \wedge isMsg1(m) \wedge getS(m) = p_2 \wedge$ $getD(m) = p_1 \wedge getK(m) = k(p_1) \wedge getP(m) = p_2$
$msg2-_{p_1,p_2,m}$	$l_{p_1,p_2,\mathrm{Ini}\}} = \mathrm{i2} \wedge p_1 \neq p_2 \wedge m \in nw \wedge isMsg2(m) \wedge getS(m) = p_2 \wedge$ $getD(m) = p_1 \wedge getK(m) = k(p_1) \wedge getP(m) = p_2 \wedge getN1(m) = n1_{p_1,p_2,\mathrm{Ini}}$
$msg3-_{p_1,p_2,m}$	$l_{p_1,p_2,\mathrm{Res}} = \mathrm{r3} \wedge p_1 \neq p_2 \wedge m \in nw \wedge isMsg3(m) \wedge getS(m) = p_2 \wedge$ $getD(m) = p_1 \wedge getK(m) = k(p_1) \wedge getN1(m) = n2_{p_1,p_2,\mathrm{Res}}$
$end_{p_1,p_2,\mathrm{Ini}}$	$l_{p_1,p_2,\mathrm{Ini}} = \mathrm{i4} \wedge p_1 \neq p_2$
$end_{p_1,p_2,\mathrm{Res}}$	$l_{p_1,p_2,\mathrm{Res}} = \mathrm{r4} \wedge p_1 \neq p_2$

Next the inherent parts of the intruder, say I, are modeled. For intruder I, we have the four observations: *nonces*, *msg1s*, *msg2s* and *msg3s*. *nonces* is a set of nonces that the intruder has gleaned. *msgis* where $i \in \{1,2,3\}$ is a set of messages of type i that the intruder has gleaned. The initial values of the four observations are empty.

For intruder I, we have the transition rules shown in Table 2. $intercept_m$ intercepts message m if $m \in nw$, and gleans nonces in the message if the nonces are encrypted with the intruder's public key and the message as it is otherwise. $fakei+_{p_1,p_2,m}$ uses message m in *msgis* to generate a message of type i, and $fakei'+_{p_1,p_2,n_1[,n_2]}$ uses nonces n_1 [and n_2] in *nonces* to generate a message of type i, where $i \in \{1,2,3\}$. Their effective conditions are shown in Table 2.

Table 2
Intruder's inherent transition rules for any pair of different principals p_1, p_2, any message m and any nonces n_1, n_2 and their effective conditions.

Trans. rules	Effective conditions
$intercept_m$	$m \in nw$
$fakei+_{p_1,p_2,m}$, $i \in \{1,2,3\}$	$M \in msgis \wedge p_1 \neq p_2$
$fakei'+_{p_1,p_2,n_1[,n_2]}$, $i \in \{1,2,3\}$	$n_1 \in nonces \wedge [n_2 \in nonces \wedge] p_1 \neq p_2$

The OTS is described in CafeOBJ. The signature is as follows:

```
pr(PRINCIPAL + NONCE + MSG + LOCATION + ROLE + SET(NONCE)*{sort Set -> SetOfNonces})
pr(SET(MSG)*{sort Set -> SetOfMsg} + BAG(MSG)*{sort Bag -> Network})
*[Sys]*
-- any initial state
op   init                : -> Sys
-- observation operations
bop  l                   : Sys Prin Prin Role        -> Loc
bops n1 n2               : Sys Prin Prin Role        -> Nonce
bop  nonces              : Sys                       -> SetOfNonces
bop  msg1s msg2s msg3s   : Sys                       -> SetOfMsg
bop  nw                  : Sys                       -> Network
bop  n                   : Sys                       -> Nonce
-- action operations
bop  msg1+ msg2+ msg3+   : Sys Prin Prin             -> Sys
bop  msg1- msg2- msg3-   : Sys Prin Prin Msg         -> Sys
bop  end                 : Sys Prin Prin Role        -> Sys
bop  intercept           : Sys Msg                   -> Sys
bop  fake1+ fake2+ fake3+ : Sys Prin Prin Msg        -> Sys
bop  fake1'+ fake3'+     : Sys Prin Prin Nonce       -> Sys
bop  fake2'+             : Sys Prin Prin Nonce Nonce -> Sys
```

A comment starts with `--` and terminates at the end of the line.

`PRINCIPAL`, `NONCE`, `MSG`, `LOCATION` and `ROLE` are modules for principals, nonces, messages, locations such as i1 and r1, and roles. `Prin`, `Nonce`, `Msg`, `Loc` and `Role` are visible sorts denoting these data. These modules are imported so that the data can be used. `SET` is a parameterized module for sets with one parameter. Two instances of `SET` are imported. One is instantiated with `NONCE`, and the other with `Msg`. `SetOfNonces` and `SetOfMsg` are visible sorts denoting a set of nonces and a set of messages. `BAG` is also a parameterized module for bags with one parameter. One instance is imported, instantiated

with MSG. Network is a visible sort denoting a bag of messages, namely the underlying computer network. Note that BOOL that is a module for Boolean values is implicitly imported and Bool is a visible sort denoting the values.

In the specification, msg1, msg2 and msg3 are the data constructors for messages of type 1, 2 and 3, respectively. Suppose that p1, p2, p3 are terms denoting principals p_1, p_2, p_3, n1, n2 denoting nonces n_1, n_2, and k denoting a public key k, terms msg1(p1,p2,k,n1,p3), msg2(p1,p2,k,n1,n2,p3) and msg3(p1,p2,k,n1) denote messages $p_1.p_2.\{n_1.p_3\}_k$, $p_1.p_2.\{n_1.n_2.p_3\}_k$ and $p_1.p_2.\{n_1\}_k$, respectively.

In the specification, basically we have 15 sets of equations: one for any initial state and the others for 14 action operations. In this paper, we show two sets of equations for msg1+ and intercept.

In the rest of the section, S is a CafeOBJ variable for Sys, P1, P2, P3 and P4 for Prin, N1 and N2 for Nonce, R1 and R2 for Role, and M for Msg.

The following is the equations for msg1+:

```
op c1-msg1+ : Sys Prin Prin -> Bool
eq c1-msg1+(S,P1,P2)          = l(S,P1,P2,Ini) = i1 and not(P1 = P2) .
--
ceq l(msg1+(S,P1,P2),P3,P4,R1)
    = (if P1 = P3 and P2 = P4 and R1 = Ini then i2 else l(S,P3,P4,R1) fi)
      if c1-msg1+(S,P1,P2) .
ceq n1(msg1+(S,P1,P2),P3,P4,R1)
    = (if P1 = P3 and P2 = P4 and R1 = Ini then n(S) else n1(S,P3,P4,R1) fi)
      if c1-msg1+(S,P1,P2) .
eq n2(msg1+(S,P1,P2),P3,P4,R1) = n2(S,P3,P4,R1) .
eq nonces(msg1+(S,P1,P2))      = nonces(S) .
eq msg1s(msg1+(S,P1,P2))       = msg1s(S) .
eq msg2s(msg1+(S,P1,P2))       = msg2s(S) .
eq msg3s(msg1+(S,P1,P2))       = msg3s(S) .
ceq nw(msg1+(S,P1,P2))         = msg1(P1,P2,k(P2),n(S),P1),nw(S) if c1-msg1+(S,P1,P2) .
ceq n(msg1+(S,P1,P2))          = new(n(S))                       if c1-msg1+(S,P1,P2) .
ceq msg1+(S,P1,P2)             = S if not c1-msg1+(S,P1,P2) .
```

The term c1-msg1+(S,P1,P2) denotes the effective condition of transition rule $msg1+_{\mathtt{P1,P2}}$ in state (denoted by) S. Comma ',' is the data constructor for bags. The term msg1(P1,P2,k(P2),n(S),P1) , nw(S) denotes the computer network after putting the message of type 1 into the computer network denoted by nw(S).

CafeOBJ provides built-in operator _==_, but it could be sometimes troublesome unless you are certain that the CafeOBJ specification regarded as a term rewriting system is confluent. Therefore, for each data structure used, we define operator _=_ that checks if two values are equal. The operator is given operator attribute comm declaring that the operator is commutative. Necessary equations for defining operator _=_ should be described.

The following is the equations for intercept:

```
eq l(intercept(S,M),P1,P2,R1)   = l(S,P1,P2,R1) .
eq n1(intercept(S,M),P1,P2,R1)  = n1(S,P1,P2,R1) .
eq n2(intercept(S,M),P1,P2,R1)  = n2(S,P1,P2,R1) .
ceq nonces(intercept(S,M))
    = (if getK(M) = k(I) and (isMsg1(M) or isMsg3(M))
       then getN1(M) nonces(S) else nonces(S) fi) if M \in nw(S) .
ceq nonces(intercept(S,M))
```

```
    = (if getK(M) = k(I) and isMsg2(M)
       then getN1(M) getN2(M) nonces(S) else nonces(S) fi) if M \in nw(S) .
ceq msg1s(intercept(S,M))
    = (if not(getK(M) = k(I)) and isMsg1(M)
       then M msg1s(S) else msg1s(S) fi) if M \in nw(S) .
ceq msg2s(intercept(S,M))
    = (if not(getK(M) = k(I)) and isMsg2(M)
       then M msg2s(S) else msg2s(S) fi) if M \in nw(S) .
ceq msg3s(intercept(S,M))
    = (if not(getK(M) = k(I)) and isMsg3(M)
       then M msg3s(S) else msg3s(S) fi) if M \in nw(S) .
ceq nw(intercept(S,M))          = nw(S) - M  if M \in nw(S) .
eq n(intercept(S,M))            = n(S) .
ceq intercept(S,M)              = S if not M \in nw(S) .
```

Juxtaposition operation is the data constructor for sets. The term `M msg1s(S)` denotes the set obtained by putting `M` into the set denoted by `msg1s(S)`.

The specification has 10 modules, and is of about 400 lines. The main module is `NSLPK` in which the signature and equations that have been just described are written, and is about of 300 lines.

6 Verification

Claim 6.1 *In any reachable state, the intruder cannot impersonate another principal* `p` *to establish a session with yet another principal* `q`.

Proof. All we have to do is to show that, in any reachable state, the intruder never obtains nonces generated by either `p` or `q` to establish sessions with each other, which immediately follows from Lemma 6.2. □

Let `n` be an arbitrary one of nonces generated by either `p` or `q` to establish sessions with each other.

Lemma 6.2 *For any reachable state* `S`*, any principals* `P1`, `P2`, `P3`*, any public key* `K`*, any nonce* `N`*, any message* `M`*,*

$$pr1 \wedge pr2 \wedge pr3 \wedge pr4 \wedge pr5 \wedge pr6 \wedge pr7 \wedge$$

$$pr8 \wedge pr9 \wedge pr10 \wedge pr11 \wedge pr12 \wedge pr13 \wedge pr14$$

where

$pr1 \equiv \neg(\texttt{n} \in \texttt{nonces(S)})$, $pr2 \equiv \neg(\texttt{msg1(P1,P2,K,n,I)} \in \texttt{msg1s(S)})$,

$pr3 \equiv \neg(\texttt{msg2(P1,P2,K,N,n,I)} \in \texttt{msg2s(S)})$, $pr4 \equiv \neg(\texttt{msg1(P1,P2,K,n,I)} \in \texttt{nw(S)})$,

$pr5 \equiv \neg(\texttt{msg2(P1,P2,K,N,n,I)} \in \texttt{nw(S)})$, $pr6 \equiv \neg(\texttt{msg1(P1,P2,k(I),n,P3)} \in \texttt{nw(S)})$,

$pr7 \equiv \neg(\texttt{msg2(P1,P2,k(I),n,N,P3)} \in \texttt{nw(S)})$, $pr8 \equiv \neg(\texttt{msg2(P1,P2,k(I),N,n,P3)} \in \texttt{nw(S)})$,

$pr9 \equiv \neg(\texttt{msg3(P1,P2,k(I),n)} \in \texttt{nw(S)})$, $pr10 \equiv \texttt{n1(S,P1,I,Res)} \neq \texttt{n}$,

$pr11 \equiv \texttt{n2(S,P1,I,Ini)} \neq \texttt{n}$, $pr12 \equiv \texttt{M} \in \texttt{msg1s(S)} \Rightarrow \texttt{getK(M)} \neq \texttt{k(I)}$,

$pr13 \equiv \texttt{M} \in \texttt{msg2s(S)} \Rightarrow \texttt{getK(M)} \neq \texttt{k(I)}$, $pr14 \equiv \texttt{M} \in \texttt{msg3s(S)} \Rightarrow \texttt{getK(M)} \neq \texttt{k(I)}$.

Proof. The lemma is proved with the CafeOBJ system as proof score executor. The proof is done by induction described in Sect. 3.

First we write a module in which the predicate to be proved is defined. The module looks like as follows:

```
mod PRED1 {
  pr(NSLPK)
  op pr1 : Sys -> Bool
  op pr2 : Sys Prin Prin Key -> Bool
  ...
  op pr  : Sys Prin Prin Prin Key Nonce Msg -> Bool
  op n   : -> Nonce
  ...
  eq (non = n)            = false . -- non is any value never used as nonces.
  eq pr1(S)               = not(n \in nonces(S)) .
  eq pr2(S,P1,P2,K)       = not(msg1(P1,P2,K,n,I) \in msg1s(S)) .
  ...
  eq pr14(S,M)            = M \in msg3s(S) implies not(getK(M) = k(I)) .
  eq pr(S,P1,P2,P3,K,N,M) = p1(S) and p2(S,P1,P2,K) and ... and p14(S,M) .
}
```

In this section, `S` is a CafeOBJ variable for `Sys`, `P1`, `P2` and `P3` for `Prin`, `N` for `Nonce`, `K` for `Key`, and `M` for `Msg`.

For the base case, all we have to do is to have the CafeOBJ system execute the following proof score:

```
open PRED1
  red pr(init,P1,P2,P3,K,N,M) .
close
```

By opening a module with CafeOBJ command `open`, we can use the operations, variables and equations declared in the module.

For the inductive step, given an arbitrary reachable state `s` in which the predicate holds, for any transition rule, we show that the predicate is still true in the successor state `s'`. We write a module describing what state `s` looks like. The module looks like as follows:

```
mod ISTEP1 {
  pr(PRED1)
  ops s s' : -> Sys
  ...
-- inductive hypothesis
  eq n \in nonces(s)                    = false .
  eq msg1(P1,P2,K,n,I) \in msg1s(s)     = false .
  ...
  eq M \in msg3s(s) and (getK(M) = k(I)) = false .
}
```

If a logical formula is described as an equation, the formula is converted into an exclusive-or canonical form à la Hsiang[14] because the CafeOBJ system reduces a logical formula into such an exclusive-or canonical form.

One of the crucial activities in the inductive step is doing case analysis. Case analysis is done based on the effective conditions of the transition rules shown in Table 1 and Table 2.

We describe the proof that the predicate `pr` is still true in the the successor state `msg1+(s,p1,p2)` for any principals `p1, p2`. We first consider two cases. One corresponds to states in which transition rule $msg1+_{\texttt{p1,p2}}$ is effective, and the other to ones in which it is not. The proof score for the former case is as follows:

```
open ISTEP1
-- arbitrary chosen objects
  ops p1 p2 : -> Prin .
```

```
-- assumption
  eq l(s,p1,p2,Ini) = i1 .
  eq (p1 = p2)      = false .
-- facts, etc.
-- the successor state
  eq s' = msg1+(s,p1,p2) .
-- check if the predicate is true in s'.
  red pr(s',P1,P2,P3,K,N,M) .
close
```

Having the CafeOBJ system execute the proof score, it returns the following term:

```
msg1(P1,P2,k(I),n,P3) \in (msg1(p1,p2,k(p2),n(s),p1) , nw(s)) and
msg1(P1,P2,K,n,I) \in (msg1(p1,p2,k(p2),n(s),p1) , nw(s)) xor
msg1(P1,P2,k(I),n,P3) \in (msg1(p1,p2,k(p2),n(s),p1) , nw(s)) xor
msg1(P1,P2,K,n,I) \in (msg1(p1,p2,k(p2),n(s),p1) , nw(s)) xor true
```

The term means that neither the message `msg1(P1,P2,k(I),n,P3)` nor the message `msg1(P1,P2,K,n,I)` is in the network `(msg1(p1,p2,k(p2),n(s),p1) , nw(s))` because $\neg(p \vee q) = p \wedge q \oplus p \oplus q \oplus \text{true}$. Therefore, if neither `p1` nor `p2` equals intruder `I`, the term should be true. Hence, the case is split into three subcases: the first one in which $\texttt{p1} \neq \texttt{I}$ and $\texttt{p2} \neq \texttt{I}$, the second one in which $\texttt{p1} = \texttt{I}$, and the last one in which $\texttt{p2} = \texttt{I}$. The result of the case analysis for checking if the predicate is still true in the successor state `msg1+(s,p1,p2)` is shown in Table 3.

Table 3
Case analysis for checking if the predicate is still true in the successor state `msg1+(s,p1,p2)` for any reachable state `s` in which the predicate holds and any principals `p1`,`p2`.

no.	the successor state	cases	subcases
1	`msg1+(s,p1,p2)`	$\texttt{l(s,p1,p2,Ini)} = \texttt{i1} \wedge \texttt{p1} \neq \texttt{p2}$	$\texttt{p1} \neq \texttt{I} \wedge \texttt{p2} \neq \texttt{I}$
2			$\texttt{p1} = \texttt{I}$
3			$\texttt{p2} = \texttt{I}$
4		$\neg(\texttt{l(s,p1,p2,Ini)} = \texttt{i1} \wedge \texttt{p1} \neq \texttt{p2})$	—

We show the proof corresponding to case 2 in Table 3. The proof score is as follows:

```
open ISTEP1
-- arbitrary chosen objects
  ops p1 p2 : -> Prin .
-- assumption
  eq l(s,I,p2,Ini) = i1 .    -- for p1 = I
  eq (I = p2)      = false . -- for P1 = I
  eq p1            = I .
-- facts, etc. (n(s) is created by I and I cannot create the same nonce as n
-- due to Nonce Creation. So, it must be different from n.)
  eq (n = n(s)) = false .
-- the successor state
  eq s' = msg1+(s,p1,p2) .
-- check if the predicate is true in s'.
  red p(s',P1,P2,P3,K,N,M) .
close
```

In the proof score, we use the assumption on **Nonce Creation**. Having the CafeOBJ system execute the proof score, it returns `true`.

The proof corresponding to case 4 in Table 3 is not necessary so long as the specification is intentionally and correctly written because there must be no difference between `s` and `msg1+(s,p1,p2)` in this case. However, it is helpful to do the proof corresponding to this case so as to find errors in the specification.

We can prove that any other transition rule preserves the predicate in a similar way. We have considered 39 cases all together for the inductive step.□

All the proof scores are of about 800 lines. It took about 12 seconds to have the CafeOBJ system load the specification and execute the proof scores on a laptop with 850MHz Pentium III processor and 512MB memory.

7 Related Work

Several methods of formally analysing authentication protocols have been proposed. Among them are methods using model checkers[4,16], ones using theorem provers[21,22], ones based on strand spaces[8,9] and ones based on multiset rewriting[2,4].

Our approach is similar to methods using theorem provers, especially Paulson's Inductive Method[21]. Inductive Method models an authentication protocols by inductively defining traces of messages from a set of rules that correspond to the possible actions of the principals including the intruder, and security properties can be stated as predicates over the traces. You can inductively prove that a certain property holds of all possible traces for an authentication protocol with the theorem prover Isabelle/HOL. In our approach, sequences of states (of an authentication protocol) instead of messages are defined, and an implementation (the CafeOBJ system) of an algebraic specification language instead of a general theorem prover is used to support verification. Since our approach uses only rewriting to prove that an authentication protocol has a safety property and does not use any heavy and slow operation such as (higher-order) unification, we may expect that our approach executes proof scores faster than methods using general theorem provers.

As concerns modeling sending and receiving messages, our approach is similar to the method based on strand spaces[8,9]. We model each of sending and receiving a message as an independent atomic action as the strand space-based method.

We model a computer network as a bag of messages, which has been affected by object-oriented specification in Maude[17]. Maude is also a member of OBJ language family as CafeOBJ. G. Denker, et al.[4] describe a finite state system of the NSPK protocol in Maude and automatically finds Lowe's attack[15] using the Maude rewrite engine as a model checker.

We should notice that writing proof scores in algebraic specification languages was first advocated by Goguen's group and developed for more than 15 years in OBJ community[12]. This paper also shows that the approach can

be applied to analysing security protocols.

8 Conclusion

A system in which an arbitrary number of principals, one of which is an intruder, take part in the NSLPK protocol has been modeled as an OTS and the OTS has been specified in CafeOBJ. we have proved that the intruder cannot impersonate another principal to establish a session with yet another principal by writing proof scores and having the CafeOBJ system execute them. We expect that our approach may model and verify other authentication protocols adequately.

In this case study, writing the proof scores was done by hand, which was less time-consuming than expected though. It took a couple of days to write the proof scores. Since the proof scores are very stylized as you have seen, however, we hope that writing proof scores can be automated to some extent. The point of writing proof scores for a proof is case analysis and to find lemmas to make progress on the proof. The former can be done based on the effective condition of each transition rule, which is expected to be performed automatically. We may have to split the case corresponding to states in which a transition rule is effective into multiple subcases, which is related to the latter and done by repeatedly writing proof scores and having the CafeOBJ system execute them. We are going to design and implement a software tool supporting writing proof scores. A proof assistant such as the Kumo system[13] developed by Goguen's group could also be used to generate proof scores.

References

[1] *CafeOBJ web page.*
URL `http://www.ldl.jaist.ac.jp/cafeobj/`

[2] Cervesato, I., H. Durgin, P. Lincoln, J. Mitchell and A. Scedrov, *A meta-notation for protocol analysis*, in: *12th IEEE CSFW*, 1999, pp. 55–69.

[3] Chandy, K. M. and J. Misra, "Parallel program design: a foundation," Addison-Wesley, Reading, MA, 1988.

[4] Denker, G., J. Meseguer and C. Talcott, *Protocol specification and analysis in Maude*, in: *Formal Methods and Security Protocols Workshop*, 1998.
URL `http://www.cs.bell-labs.com/who/nch/fmsp/`

[5] Diaconescu, R. and K. Futatsugi, "CafeOBJ report," AMAST Series in Computing, 6, World Scientific, Singapore, 1998.

[6] Diaconescu, R. and K. Futatsugi, *Behavioural coherence in object-oriented algebraic specification*, J. Universal Computer Science **6** (2000), pp. 74–96.

[7] Dolev, D. and A. C. Yao, *On the security of public key protocols*, IEEE Trans. Inform. Theory **IT-29** (1983), pp. 198–208.

[8] Fábrega, F. J. T., J. C. Herzog and J. D. Guttman, *Strand space pictures*, in: *Formal Methods and Security Protocols Workshop*, 1998.
URL `http://www.cs.bell-labs.com/who/nch/fmsp/`

[9] Fábrega, F. J. T., J. C. Herzog and J. D. Guttman, *Strand spaces: Proving security protocols correct*, J. Computer Security **7** (1999), pp. 191–230.

[10] Futatsugi, K. and K. Ogata, *Rewriting can verify distributed real-time systems*, in: *Int'l Symposium on Rewriting, Proof, and Computation*, 2001, pp. 60–79.

[11] Goguen, J. and G. Malcolm, *A hidden agenda*, Theor. Comput. Sci. **245** (2000), pp. 55–101.

[12] Goguen, J. and G. Malcolm, editors, "Software Engineering with OBJ: algebraic specification in action," Kluwer Academic Publishers, 2000.

[13] Goguen, J. A. and K. Lin, *Web-based support for cooperative software engineering*, Annals of Software Engineering **12** (2001), pp. 167–191.

[14] Hsiang, J., "Refutational Theorem Proving using Term Rewriting Systems," Ph.D. thesis, University of Illinois at Champaign-Urbana (1981).

[15] Lowe, G., *An attack on the Needham-Schroeder public-key authentication protocol*, Inf. Process. Lett. **56** (1995), pp. 131–133.

[16] Lowe, G., *Breaking and fixing the Needham-Schroeder public-key protocol using FDR*, in: *TACAS '96*, LNCS 1055 (1996), pp. 147–166.

[17] *Maude web page.*
URL `http://maude.csl.sri.com/`

[18] Needham, R. M. and M. D. Schroeder, *Using encryption for authentication in large networks of computers*, Comm. ACM **21** (1978), pp. 993–999.

[19] Ogata, K. and K. Futatsugi, *Modeling and verification of distributed real-time systems based on CafeOBJ*, in: *ASE '01* (2001), pp. 185–192.

[20] Ogata, K. and K. Futatsugi, *Formal analysis of Suzuki&Kasami distributed mutual exclusion algorithm*, in: *FMOODS '02* (2002), pp. 181–195.

[21] Paulson, L. C., *The inductive approach to verifying cryptographic protocols*, J. Computer Security **6** (1998), pp. 85–128.

[22] Schneider, S., *Verifying authentication protocols in CSP*, IEEE Trans. on Softw. Eng. **24** (1998), pp. 741–758.