Electronic Notes in Theoretical Computer Science 71 (2003)

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

Rewriting-Based Veriﬁcation of Authentication Protocols

Kazuhiro Ogata [a](#_bookmark0)*,*[b](#_bookmark1)*,*[1](#_bookmark2) and Kokichi Futatsugi [b](#_bookmark1)*,*[2](#_bookmark2)

a *NEC Software Hokuriku, Ltd.*

b *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.

*Key words:* Authentication protocols, CafeOBJ, the NSPK protocol, observational transition systems, rewriting, verification

# Introduction

Security protocols such as authentication ones are key technology if we ex- change 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 attac[k[15]](#_bookmark21) to the Needham-Schroeder Public-Key authentication protocol (the NSPK proto[col)[18].](#_bookmark28) Therefore, several methods of formally analysing security protocols have been prop[osed[2,4,9,16,21,22].](#_bookmark32)

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],](#_bookmark30) which is described in [CafeOBJ[1,5].](#_bookmark17) 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. The CafeOBJ system can be used as

1 Email: [ogatak@acm.org](mailto:ogatak@acm.org)

2 Email: [kokichi@jaist.ac.jp](mailto:kokichi@jaist.ac.jp)

*◯*c *2003 Published by Elsevier Science B. V. Open access under* [*CC BY-NC-ND license.*](http://creativecommons.org/licenses/by-nc-nd/3.0/)

an interactive proof-checker or verifier on several lev[els[10].](#_bookmark22) In the proposed method, the CafeOBJ system is used as proof score executor. The NSPK protocol corrected by Low[e[15]](#_bookmark21) is used to show our method.

The rest of the paper is organized as follows. Section [2](#_bookmark3) mentions CafeOBJ. Observational transition systems and a way of describing them in CafeOBJ are written in Sect [3.](#_bookmark4) Section [4](#_bookmark5) gives a brief description of the NSPK protocol corrected by Lowe. Section [5](#_bookmark6) describes the observational transition system modeling the protocol and its specification in CafeOBJ. Section [6](#_bookmark9) shows (part of) the proof that the protocol has a safety property. Section [7](#_bookmark12) gives related work, and we conclude the paper in Sect [8.](#_bookmark13)

# CafeOBJ in a Nutshell

[CafeOBJ[1,5]](#_bookmark17) 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]](#_bookmark22) 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 obser- vation 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 basi- cally 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 exe- cutability makes it possible to simulate described systems and to verify that they possess some desired properties.

# Observational Transition Systems

We assume that there exists a universal state space called Υ. When we de- scribe 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]](#_bookmark30) can be used to model a system in this way. [UNITY[3]](#_bookmark19) is an ancestor of ots’s, which are reformalized by adopting the concept of hidden [algebra[6,11].](#_bookmark22)

An ots *S* = *⟨O, I, T⟩* consists of:

* + *O* : A set of observations. Each observation *o ∈O* is a function *o* :Υ *→ D* mapping each *υ ∈* Υ 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 *S* and two states *υ*1*, υ*2 *∈* Υ, the equality between two states, denoted by *υ*1 =*S υ*2, with respect to *S* is defined as follows:

*υ*1 =*S υ*2 iff *∀o ∈ O.o*(*υ*1)= *o*(*υ*2)*,*

where ‘=’ in *o*(*υ*1) = *o*(*υ*2) is supposed to be well defined for the range of each *o ∈ O*. *S* may be removed from =*S* if it is clear from the context.

* + *I* : The initial condition. This condition specifies the initial value of each observation that defines initial states of the ots.
  + *T* : A set of conditional transition rules. Each transition rule *τ ∈ T* is a relation between states provided that, for each state *υ ∈* Υ, there exists a state *υ' ∈* Υ, called a successor state, such that *τ* (*υ, υ'*) and moreover,

for each state *υ*1*, υ*2*, υ' , υ' ∈* Υ such that *υ*1 =*S υ*2, *τ* (*υ*1*, υ'* ) and *τ* (*υ*2*, υ'* ),

1 2 1 2

*'* =*S υ'* . *τ* can be regarded as a function on equivalent classes of Υ with

*υ*

2

1

respect to =*S* . Therefore, we assume that *τ* (*υ*) denotes the representative element of the equivalent class the successor states of *υ* with respect to *τ* belong to, and *τ* (*υ*) is called *the* successor state of *υ* with respect to *τ* .

The condition *cτ* for a transition rule *τ ∈ T* is called the *effective condi- tion*. 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 *υ ∈* Υ, *cτ* is true in *υ*, namely *τ* is *effective* in *υ*, iff *υ /*=*S τ* (*υ*).

Multiple similar observations or transition rules may be indexed. Gen- erally, observations and transition rules are denoted by *oi*1*,...,im* and *τj*1*,...,jn* , respectively, provided that *m, n ≥* 0 and we assume that there exist data types *Dk* such that *k ∈ Dk* (*k* = *i*1*,..., im, j*1*,..., jn*). For example, an integer array *a* possessed by a process *p* may be denoted by an observation *ap*, and the increment of the *i*th element of the array may be denoted by a transition rule *incap,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 *S* is an infinite sequence *s*0*, s*1*,...* of states satisfying:

* + *Initiation* : For each *o ∈ O*, *o*(*s*0) satisfies *I*.
  + *Consecution* : For each *i ∈ {*0*,* 1*,.. .}*, *si*+1 =*S τ* (*si*) for some *τ ∈T* .
  + *Fairness* : For each *τ ∈ T* , there exist an infinite number of indexes *i ∈*

*{*0*,* 1*,.. .}* such that *si*+1 =*S τ* (*si*).

A state is called *reachable* with respect to *S* if it appears in an execution of *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* :Υ *→ {*true*,* false*}* is a safety property with respect to *S* iff *p* is a state predicate and *p*(*υ*) holds for

every reachable *υ ∈* Υ.

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

* + Base case: For any state *υ ∈* Υ in which each observation *o ∈O* satisfies *I*, we show that *p*(*υ*) holds.
  + Inductive step: Given any reachable state *υ ∈* Υ such that *p*(*υ*) holds, we show that, for any transition rule *τ ∈T* , *p*(*τ* (*υ*)) also holds.

An ots *S* is described in CafeOBJ. The universal state space Υ is denoted by a hidden sort, say Sys, by declaring \*[Sys]\*.

An observation *oi*1*,...,im ∈ O* is denoted by a CafeOBJ observation oper- ation. We assume that data types *Dk* (*k* = *i*1*,..., im*) and *D* are described in initial algebra and there exist visible sorts *Sk* (*k* = *i*1*,..., im*) and *S* cor- responding to the data types. The CafeOBJ observation operation denoting *oi*1*,...,im* is declared as follows:

bop o : Sys S*i*1 ... S*im* -> S

The initial condition *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 *oi*1*,...,im* is *f* (*i*1*,..., im*), this can be described in CafeOBJ as follows:

eq o(init,X*i*1 ,... ,X*im* ) = f(X*i*1 ,... ,X*im* ) .

where X*k* (*k* = *i*1*,..., im*) is a CafeOBJ variable with S*k*, and f(X*i*1 ,... ,X*im* )

means a term (consisting of X*i*1 *,...,* X*im* ) corresponding to *f* (*i*1*,..., im*).

A transition rule *τj*1*,...,jn ∈ T* is denoted by a CafeOBJ action operation. We assume that data types *Dk* (*k* = *j*1*,..., jn*) are described in initial algebra and there exist visible sorts *Sk* (*k* = *j*1*,..., jn*) corresponding to the data types. The CafeOBJ action operation denoting *τj*1*,...,jn* is declared as follows:

bop a : Sys S*j*1 ... S*jn* -> Sys

If *τj*1*,...,jn* is executed in a state in which it is effective, the value of *oi*1*,...,im*

may be changed, which can be described in CafeOBJ generally as follows:

ceq o(a(S,X*j*1 ,... ,X*jn* ),X*i*1 ,... ,X*im* )

= e-a(S,X*j*1 ,... ,X*jn* ,X*i*1 ,... ,X*im* ) if c-a(S,X*j*1 ,... ,X*jn* ) .

where e-a(S,X*j*1 ,... ,X*jn* ,X*i*1 ,... ,X*im* ) means a term (consisting of S*,* X*j*1 *,...,* X*jn,* X*i*1 *,...,* X*im* ) corresponding to the value of *oi*1*,...,im* in the successor state, and c-a(S,X*j*1 ,... ,X*jn* ) means a term (consisting of S*,* X*j*1 *,...,* X*jn* ) corre- sponding to *cτj*1*,...,jn* .

If *τj*1*,...,jn* 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*j*1 ,... ,X*jn* ) = S if not c-a(S,X*j*1 ,... ,X*jn* ) .

If the value of *oi*1*,...,im* is not affected by executing *τj*1*,...,jn* in any state (regardless of the truth value of *cτj*1*,...,jn* ), the following equation is declare:

eq o(a(S,X*j*1 ,... ,X*jn* ),X*i*1 ,... ,X*im* ) = o(S,X*i*1 ,... ,X*im* ) .

# The NSLPK Protocol

Needham and Schro[eder[18]](#_bookmark28) proposed an authentication protocol, called the NSPK protocol, with public-key cryptosystems in 1978. Low[e[15]](#_bookmark21) 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:

|  |  |  |
| --- | --- | --- |
| Message 1 | *p*1 *→ p*2 : | *p*1*.p*2*.{np*1 *.p*1*}k*(*p*2) |
| Message 2 | *p*2 *→ p*1 : | *p*2*.p*1*.{np*1 *.np*2 *.p*2*}k*(*p*1) |
| Message 3 | *p*1 *→ p*2 : | *p*1*.p*2*.{np*2 *}k*(*p*2) |

*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 *np*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 *np*1 . It then returns *np*1 along with a new nonce *np*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 *np*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 *np*1 . *p*1 then returns *np*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 *np*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.

# 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 mo[del[7].](#_bookmark20) 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 ∈ {*Ini*,* Res*}*, we have the following observa- tions: *lp*1*,p*2*,r* , *n*1*p*1*,p*2*,r* and *n*2*p*1*,p*2*,r* . *lp*1*,p*2*,*Ini is used for *p*1 as initiator, having

one of the four possible values i1, i2, i3 and i4, while *lp*1*,p*2*,*Res is used for *p*1 as responder, having one of the four possible values r1, r2, r3 and r4. If *lp*1*,p*2*,*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 *lp*1*,p*2*,*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. *n*1*p*1*,p*2*,*Ini and *n*2*p*1*,p*2*,*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. *n*1*p*1*,p*2*,*Res and *n*2*p*1*,p*2*,*Res are used for *p*1 so as to record the nonce received from *p*2 and the nonce created by *p*1 re- spectively when *p*1 as responder tries establishing or has a session with *p*2 as initiator. Initially *lp*1*,p*2*,*Ini is i1, *lp*1*,p*2*,*Res is r1, and *n*1*p*1*,p*2*,r* and *n*2*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.](#_bookmark7) *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 ∈ {*1*,* 2*,* 3*}*. *end{p*1*,p*2*,r}* finishes a session of *p*1 as *r* with *p*2 as *¬r*, where *¬*Ini = Res and *¬*Res = Ini. Their effective conditions are shown in Table [1.](#_bookmark7)

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 | *lp*1*,p*2*,*Ini = i1 *Λ p*1 */*= *p*2 |
| *msg2* +*p*1*,p*2 | *lp*1*,p*2*,*Res = r2 *Λ p*1 */*= *p*2 |
| *msg3* +*p*1*,p*2 | *lp*1*,p*2*,*Ini = i3 *Λ p*1 */*= *p*2 |
| *msg1 —p*1*,p*2*,m* | *lp*1*,p*2*,*Res = r1 *Λ p*1 */*= *p*2 *Λ m ∈ nw Λ isMsg1* (*m*) *Λ getS* (*m*)= *p*2*Λ*  *getD* (*m*)= *p*1 *Λ getK* (*m*)= *k*(*p*1) *Λ getP* (*m*)= *p*2 |
| *msg2 —p*1*,p*2*,m* | *lp*1*,p*2*,*Ini*}* = i2 *Λ p*1 */*= *p*2 *Λ m ∈ nw Λ isMsg2* (*m*) *Λ getS* (*m*)= *p*2*Λ*  *getD* (*m*)= *p*1 *Λ getK* (*m*)= *k*(*p*1) *Λ getP* (*m*)= *p*2 *Λ getN1* (*m*)= *n*1*p*1*,p*2*,*Ini |
| *msg3 —p*1*,p*2*,m* | *lp*1*,p*2*,*Res = r3 *Λ p*1 */*= *p*2 *Λ m ∈ nw Λ isMsg3* (*m*) *Λ getS* (*m*)= *p*2*Λ*  *getD* (*m*)= *p*1 *Λ getK* (*m*)= *k*(*p*1) *Λ getN1* (*m*)= *n*2*p*1*,p*2*,*Res |
| *end p*1*,p*2*,*Ini | *lp*1*,p*2*,*Ini = i4 *Λ p*1 */*= *p*2 |
| *end p*1*,p*2*,*Res | *lp*1*,p*2*,*Res = r4 *Λ p*1 */*= *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 ∈ {*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.](#_bookmark8) *interceptm* intercepts message *m* if *m ∈ 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 ,p ,n* [*,n* ] uses nonces *n*1 [and *n*2] in *nonces* to generate a

1 2 1 2

message of type *i*, where *i ∈ {*1*,* 2*,* 3*}*. Their effective conditions are shown in

Table [2](#_bookmark8).

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 |
| *interceptm* | *m ∈ nw* |
| *fakei* +*p*1*,p*2*,m*, *i ∈ {*1*,* 2*,* 3*}* | *M ∈ msgis Λ p*1 */*= *p*2 |
| *fakei'*+*p*1*,p*2*,n*1[*,n*2], *i ∈ {*1*,* 2*,* 3*}* | *n*1 *∈ nonces Λ* [*n*2 *∈ noncesΛ*]*p*1 */*= *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 denot- ing 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* +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 trou- blesome 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.

# 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.](#_bookmark10) *✷*

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*,*

*pr*1 *∧ pr*2 *∧ pr*3 *∧ pr*4 *∧ pr*5 *∧ pr*6 *∧ pr*7*∧*

*pr*8 *∧ pr*9 *∧ pr*10 *∧ pr*11 *∧ pr*12 *∧ pr*13 *∧ pr*14

*where*

*pr*1 *≡ ч*(n *∈* nonces(S))*, pr*2 *≡ ч*(msg1(P1,P2,K,n,I) *∈* msg1s(S))*, pr*3 *≡ ч*(msg2(P1,P2,K,N,n,I) *∈* msg2s(S))*, pr*4 *≡ ч*(msg1(P1,P2,K,n,I) *∈* nw(S))*,*

*pr*5 *≡ ч*(msg2(P1,P2,K,N,n,I) *∈* nw(S))*, pr*6 *≡ ч*(msg1(P1,P2,k(I),n,P3) *∈* nw(S))*,*

*pr*7 *≡ ч*(msg2(P1,P2,k(I),n,N,P3) *∈* nw(S))*, pr*8 *≡ ч*(msg2(P1,P2,k(I),N,n,P3) *∈* nw(S))*, pr*9 *≡ ч*(msg3(P1,P2,k(I),n) *∈* nw(S))*, pr*10 *≡* n1(S,P1,I,Res) */*= n*,*

*pr*11 *≡* n2(S,P1,I,Ini) */*= n*, pr*12 *≡* M *∈* msg1s(S) *⇒* getK(M) */*= k(I)*, pr*13 *≡* M *∈* msg2s(S) *⇒* getK(M) */*= k(I)*, pr*14 *≡* M *∈* msg3s(S) *⇒* getK(M) */*= k(I)*.*

**Proof.** The lemma is proved with the CafeOBJ system as proof score execu- tor. The proof is done by induction described in Sect. [3.](#_bookmark4)

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 opera- tions, 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 a` la [Hsiang[14]](#_bookmark23) 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](#_bookmark7) and Table [2.](#_bookmark8)

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* +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 *ч*(*p V q*) = *p Λ q ⊕ p ⊕ q ⊕* 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 p1 */*= I and p2 */*= I, the second one in which p1 = I, and the last one in which p2 = 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.](#_bookmark11)

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) | l(s,p1,p2,Ini) = i1 *Λ* p1 */*= p2 | p1 */*= I *Λ* p2 */*= I |
| 2 | p1 = I |
| 3 | p2 = I |
| 4 | *ч*(l(s,p1,p2,Ini) = i1 *Λ* p1 */*= p2) | — |

We show the proof corresponding to case 2 in Table [3.](#_bookmark11) 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](#_bookmark11) 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.

# Related Work

Several methods of formally analysing authentication protocols have been pro- posed. Among them are methods using model check[ers[4,16],](#_bookmark26) ones using theo- rem prov[ers[21,22],](#_bookmark32) ones based on strand [spaces[8,9]](#_bookmark22) and ones based on multiset [rewriting[2,4].](#_bookmark16)

Our approach is similar to methods using theorem provers, especially Paul- son’s Inductive Metho[d[21].](#_bookmark31) Inductive Method models an authentication pro- tocols by inductively defining traces of messages from a set of rules that corre- spond to the possible actions of the principals including the intruder, and secu- rity 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].](#_bookmark22) 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 af- fected by object-oriented specification in [Maude[17].](#_bookmark27) Maude is also a member of OBJ language family as CafeOBJ. G. Denker, et [al.[4]](#_bookmark16) describe a finite state system of the NSPK protocol in Maude and automatically finds Lowe’s attack[[15]](#_bookmark21) using the Maude rewrite engine as a model checker.

We should notice that writing proof scores in algebraic specification lan- guages was first advocated by Goguen’s group and developed for more than 15 years in OBJ communit[y[12].](#_bookmark24) This paper also shows that the approach can be applied to analysing security protocols.

# 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 auto- matically. 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 sup- porting writing proof scores. A proof assistant such as the Kumo [system[13]](#_bookmark25) 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/>

1. I. Cervesato, H. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. *A meta- notation for protocol analysis*, in: *12th IEEE CSFW*, 1999, pp. 55–69.
2. K. M. Chandy and J. Misra. “Parallel program design: a foundation,” Addison- Wesley, Reading, MA, 1988.
3. G. Denker, 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/>

1. R. Diaconescu and K. Futatsugi. “CafeOBJ report,” AMAST Series in Computing, 6, World Scientific, Singapore, 1998.
2. R. Diaconescu and K. Futatsugi. *Behavioural coherence in object-oriented algebraic speciﬁcation*, J. Universal Computer Science **6** (2000), pp. 74–96.
3. D. Dolev and A. C. Yao. *On the security of public key protocols*, IEEE Trans. Inform. Theory **IT-29** (1983), pp. 198–208.
4. F. J. T. Fa´brega, 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/>

1. F. J. T. Fa´brega, J. C. Herzog, and J. D. Guttman. *Strand spaces: Proving security protocols correct*, J. Computer Security **7** (1999), pp. 191–230.
2. K. Futatsugi and K. Ogata. *Rewriting can verify distributed real-time systems*, in: *Int’l Symposium on Rewriting, Proof, and Computation*, 2001, pp. 60–79.
3. J. Goguen and G. Malcolm. *A hidden agenda*, Theor. Comput. Sci. **245** (2000),

pp. 55–101.

1. J. Goguen and G. Malcolm, editors. “Software Engineering with OBJ: algebraic specification in action,” Kluwer Academic Publishers, 2000.
2. J. A. Goguen and K. Lin. *Web-based support for cooperative software engineering*, Annals of Software Engineering **12** (2001), pp. 167–191.
3. J. Hsiang. “Refutational Theorem Proving using Term Rewriting Systems,” Ph.D. thesis, University of Illinois at Champaign-Urbana (1981).
4. G. Lowe. *An attack on the Needham-Schroeder public-key authentication protocol*, Inf. Process. Lett. **56** (1995), pp. 131–133.
5. G. Lowe. *Breaking and ﬁxing the Needham-Schroeder public-key protocol using FDR*, in: *TACAS ’96*, LNCS 1055 (1996), pp. 147–166.
6. *Maude web page*.

URL <http://maude.csl.sri.com/>

1. R. M. Needham and M. D. Schroeder. *Using encryption for authentication in large networks of computers*, Comm. ACM **21** (1978), pp. 993–999.
2. K. Ogata and K. Futatsugi. *Modeling and veriﬁcation of distributed real-time systems based on CafeOBJ*, in: *ASE ’01* (2001), pp. 185–192.
3. K. Ogata and K. Futatsugi. *Formal analysis of Suzuki&Kasami distributed mutual exclusion algorithm*, in: *FMOODS ’02* (2002), pp. 181–195.
4. L. C. Paulson. *The inductive approach to verifying cryptographic protocols*, J. Computer Security **6** (1998), pp. 85–128.
5. S. Schneider. *Verifying authentication protocols in CSP*, IEEE Trans. on Softw. Eng. **24** (1998), pp. 741–758.