Chocolat/SMV: A Translator from CafeOBJ into SMV

Kazuhiro Ogata
NEC Software Hokuriku, Ltd.
ogatak@acm.org

Masahiro Nakano, Masaki Nakamura, Kokichi Futatsugi
School of Information Science, JAIST
{m-nakano, masaki-n, kokichi}@jaist.ac.jp

Abstract

Chocolat/SMV is a translator that takes a CafeOBJ specification of a transition system called an OTS and generates an SMV specification of a finite version of the OTS. The primary purpose of the translation is to find errors lurked in CafeOBJ specifications of OTSs with SMV.

1. Introduction

Interactive theorem provers and model checkers are complementary. The former can verify that infinite-state systems have properties, while the latter can find counterexamples that finite-state systems do not have properties.

In the OTS/CafeOBJ method[6], a system is modeled as a transition system called an OTS, the OTS is written in CafeOBJ[1], an algebraic specification language, and it is verified that the OTS has properties by using the CafeOBJ system as an interactive theorem prover. CafeOBJ specifications of OTSs are called OTS/CafeOBJ specifications. We have done case studies to demonstrate its effectiveness.

But, there are no model checking facilities available in the OTS/CafeOBJ method and therefore it may take much time to notice errors in OTS/CafeOBJ specifications. In the course of the verification with the OTS/CafeOBJ method that an e-commerce protocol has a property[5], it took a whole week to notice that there exists a counterexample[4].

That is why we begun designing and implementing Chocolat/SMV that takes an OTS/CafeOBJ specification plus some annotations and generates an SMV one. Annotations are used to tell Chocolat/SMV how to finitize OTSs. Chocolat/SMV uses a simple data abstraction technique to finitize OTSs, which is described in the paper. The primary purpose of Chocolat/SMV is to find counterexamples in OTS/CafeOBJ specifications with the SMV model checker[2]. Therefore, it is crucial for the translation to have the property that any counterexample found in an SMV specification translated from an OTS/CafeOBJ one is also a counterexample in the OTS/CafeOBJ one. The paper describes the proof that the translation has the property. The paper also reports on a case study that Chocolat/SMV is applied to the NSPK authentication protocol[3].

2 Preliminaries

Let Υ be a universal state space. An OTS[6] $\mathcal{S}$ consists of $\langle \mathcal{O}, \mathcal{I}, \mathcal{T} \rangle$ such that 1) $\mathcal{O}$: a set of observers; each $o \in \mathcal{O}$ is a function $o : \Upsilon \rightarrow D$, where D is a data type; given two states $\upsilon_1, \upsilon_2 \in \Upsilon$, the equivalence ($\upsilon_1 =_{\mathcal{S}} \upsilon_2$) between them wrt $\mathcal{S}$ is defined as $\forall o \in \mathcal{O}. o(\upsilon_1) = o(\upsilon_2)$, 2) $\mathcal{I}$: the set of initial states such that $\mathcal{I} \subseteq \Upsilon$, and 3) $\mathcal{T}$: a set of conditional transitions; each $\tau \in \mathcal{T}$ is a function $\tau : \Upsilon \rightarrow \Upsilon$ such that $\tau(\upsilon_1) =_{\mathcal{S}} \tau(\upsilon_2)$ for each $[\upsilon] \in \Upsilon/=_{\mathcal{S}}$ and each $\upsilon_1, \upsilon_2 \in [\upsilon]$; $\tau(\upsilon)$ is called the successor state of $\upsilon \in \Upsilon$ wrt τ; the condition c_τ of τ is called the effective condition. A predicate p is called invariant wrt $\mathcal{S}$ iff $p(\upsilon)$ holds for every reachable state υ wrt $\mathcal{S}$. Observers and transitions may be parameterized, which are generally expressed as $o_{d_{i_1},\ldots,d_{i_m}} : \Upsilon \rightarrow D_i$ and $\tau_{d_{j_1},\ldots,d_{j_n}} : \Upsilon \rightarrow \Upsilon$, provided that $m, n \geq 0$ and there exists a data type D_k such that $d_k \in D_k$ ($k = i_1, \ldots, i_m, i, j_1, \ldots, j_n$).

Abstract machines as well as abstract data types can be specified in CafeOBJ[1], which has two kinds of sorts: visible and hidden sorts denoting abstract data types and the state spaces of abstract machines, and two kinds of operators wrt hidden sorts: action and observation operators that denote state transitions of abstract machines and let us know the situation where abstract machines are located. Both an action operator and an observation operator take a state of an abstract machine and zero or more data, and return the successor state and a value that characterizes the situation where the abstract machine is located. Keyword `bop` is used to declare action and observation operators, while keyword `op` is used to declare other operators. Operators are defined with equations. Keyword `eq` is used to declare equations, while keyword `ceq` is used to declare conditional equations; conditions are written after keyword `if`.

An OTS $\mathcal{S}$ is specified in CafeOBJ. Υ is denoted by a hidden sort, say *H*, $o_{d_{i_1},\ldots,d_{i_m}}$ by a CafeOBJ observation operator, say *o*, and $\tau_{d_{j_1},\ldots,d_{j_n}}$ by a CafeOBJ action operator, say *a*. An action operator is basically specified with equations by describing how the value returned by each observation operator changes. A typical form of such equations looks like

$$\texttt{ceq}\ o(a(S, X_{j_1}, \ldots, X_{j_n}), X_{i_1}, \ldots, X_{i_m}) = e\text{-}a(S, X_{j_1}, \ldots, X_{j_n}, X_{i_1}, \ldots, X_{i_m})\ \texttt{if}\ c\text{-}a(S, X_{j_1}, \ldots, X_{j_n})\ .$$

S and each X_k are CafeOBJ variables of H and the visible sort denoting D_k. $a(\ldots)$ denotes the successor state of S wrt $\tau_{j_1,\ldots,j_n}$. e-$a(\ldots)$ denotes the value returned by $o_{i_1,\ldots,i_m}$ in the successor state. c-$a(\ldots)$ denotes $c_{\tau_{j_1,\ldots,j_n}}$.

SMV[2] is a symbolic model checker, checking if a finite-state transition system satisfies a property written in CTL (Computation Tree Logic). A transition system is written as a module, which consists of a VAR section and an ASSIGN section. The typed variables declared in a VAR section denote the state space of a transition system and states are denoted by possible values assigned to the variables. Initial values of variables and state transitions are defined in an ASSIGN section. Initial values of variables are defined with operator `init` such that $\texttt{init}(x) := Exp$. State transitions are defined with operator `next` such that $\texttt{next}(x) := Exp$. Expression *Exp* can be written with guarded commands (using `case` statements). When the value obtained by evaluating expression *Exp* is a set, one value is nondeterministically chosen from the set and assigned to variable x. Modules are regarded as records; variables in a VAR section are members of a record.

3 Translation from OTS/CafeOBJ into SMV

When OTS/CafeOBJ specifications are translated into SMV ones, we have two problems to solve: 1) composed data types such as lists should be encoded in basic data types such as arrays and integers available in SMV, and 2) OTSs should be finitized. For the first problem, data types available in SMV and those straightforwardly made of such available data types are only used in OTS/CafeOBJ specifications that are translated into SMV specifications. Currently available data types are truth values, natural numbers, lists, records and those made of these data types. The second problem is discussed in detail in the coming subsection. In the rest of this section, we also describe how to generate SMV specifications of finite OTSs and the proof that the translation has a desired property.

3.1 Simple Data Abstraction

We describe how to generate a finite OTS $\mathcal{S}' = \langle \mathcal{O}', \mathcal{I}', \mathcal{T}' \rangle$ from an (infinite) OTS $\mathcal{S} = \langle \mathcal{O}, \mathcal{I}, \mathcal{T} \rangle$. For an OTS $\mathcal{S}'$ to be finite means that the coset $\Upsilon'/{=_{\mathcal{S}'}}$ consists of finite elements, where Υ' is the state space of $\mathcal{S}'$.

For each data type D used in $\mathcal{S}$, which is regarded as a set of values, we choose a finite set D' such that $D' \subseteq D$ and define D'' as $D' \cup \{\text{out}_D\}$ where $\text{out}_D \notin D$. out_D denotes an arbitrary element in D but not in D'. We also define the equivalence relation $=_{D''}$ as

$$d_1 =_{D''} d_2 \equiv \begin{cases} d_1 =_D d_2 & \textbf{if } d_1 \in D' \wedge d_2 \in D' \\ \text{true} & \textbf{if } d_1 \notin D' \wedge d_2 \notin D' \\ \text{false} & \textbf{otherwise} \end{cases}$$

The equivalence relation $=_{D''}$ can also be used to check if two elements of D are equal.

Let Υ' be the state space of $\mathcal{S}'$ such that $\Upsilon' \equiv \Upsilon \cup \{v_{\text{out}}\}$ and $v_{\text{out}} \notin \Upsilon$. Using the equivalence relation $=_{D''}$ on each data type D, a finite OTS $\mathcal{S}'$ is defined as follows:

• $\mathcal{O}'$ is $\{o'_{d_{i_1},\ldots,d_{i_m}} \mid o_{d_{i_1},\ldots,d_{i_m}} \in \mathcal{O}, d_{i_1} \in D'_{i_1}, \ldots, d_{i_m} \in D'_{i_m}\}$. Each $o'_{d_{i_1},\ldots,d_{i_m}}$ is defined as

$$o'_{d_{i_1},\ldots,d_{i_m}}(v) \equiv \begin{cases} o_{d_{i_1},\ldots,d_{i_m}}(v) \\ \quad \textbf{if } o_{d_{i_1},\ldots,d_{i_m}}(v) \in D'_i \wedge v \in \Upsilon \\ \text{out}_{D_i} \qquad\qquad \textbf{otherwise} \end{cases}$$

By defining $\mathcal{O}'$, the equivalence relation $=_{\mathcal{S}'}$ on Υ' is also defined, and the coset $\Upsilon'/{=_{\mathcal{S}'}}$ is finite.

• $\mathcal{I}'$ is $\mathcal{I}$. Note that although $\mathcal{I}/{=_{\mathcal{S}}}$ may consist of infinite elements, $\mathcal{I}/{=_{\mathcal{S}'}}$ consists of finite elements.

• $v \in \Upsilon'$ is called being inside wrt $\mathcal{S}'$ if $o'_{d_{i_1},\ldots,d_{i_m}}(v) \in D'_i$ for every $o'_{d_{i_1},\ldots,d_{i_m}} \in \mathcal{O}'$. For $v \in \Upsilon'$ to be inside wrt $\mathcal{S}'$ is denoted by the predicate $\text{inside}_{\mathcal{S}'}$ that is defined as

$$\text{inside}_{\mathcal{S}'}(v) \equiv \forall o'_{d_{i_1},\ldots,d_{i_m}} \in \mathcal{O}'.(o'_{d_{i_1},\ldots,d_{i_m}}(v) \in D'_i)$$

$\mathcal{S}'$ may be omitted from $\text{inside}_{\mathcal{S}'}$. $\mathcal{T}'$ is $\{\tau'_{d'_{j_1},\ldots,d'_{j_n}} \mid \tau_{d_{j_1},\ldots,d_{j_n}} \in \mathcal{T}\}$, where each d'_{j_k} is d_{j_k} if $d'_{j_k} \in D'_{j_k}$ and $\text{out}_{D_{j_k}}$ otherwise. Each $\tau'_{d'_{j_1},\ldots,d'_{j_n}}$ is defined as

$$\tau'_{d'_{j_1},\ldots,d'_{j_n}}(v) \equiv \begin{cases} \tau_{d_{j_1},\ldots,d_{j_n}}(v) & \textbf{if } d'_{j_1} \in D'_{j_1} \wedge \\ \quad \ldots \wedge d'_{j_n} \in D'_{j_n} \wedge \text{inside}_{\mathcal{S}'}(v) \\ v_{\text{out}} & \textbf{otherwise} \end{cases}$$

According to the definition of $\mathcal{S}'$, a predicate p over Υ to be proved invariant wrt $\mathcal{S}$ is redefined as

$$p'(v) \equiv (\text{inside}_{\mathcal{S}'}(v) \Rightarrow p(v))$$

3.2 Generating SMV Specifications

An OTS $\mathcal{S}'$ translated from an (infinite) OTS $\mathcal{S}$ is finite-state (precisely the coset $\Upsilon'/{=_{\mathcal{S}'}}$ is finite) and the data types used in $\mathcal{S}'$ as well as $\mathcal{S}$ are available in SMV from the assumption. Therefore, there is a clear correspondence between $\mathcal{S}'$ and an SMV transition system up to data types. An SMV specification of $\mathcal{S}'$ is generated in the way: 1) $\mathcal{S}'$ is represented as an SMV module, 2) a non-parameterized observation $o' : \Upsilon' \rightarrow D'' \in \mathcal{O}'$ is represented by a variable x whose type is D''; a parameterized observation $o'_{d_{i_1},\ldots,d_{i_m}}$:

$\Upsilon' \to D_i'' : \mathcal{O}'$ is represented by an m-dimensional array a : **array** $0 .. (|D'_{i_1}|-1)$ **of** ... **array** $0 .. (|D'_{i_m}|-1)$ **of** D_i'', where $|D'_{i_k}|$ is the number of the elements in D'_{i_k}, 3) the initial states are defined by setting the variables and arrays denoting the observations to values with operator `init`, and 4) the transitions are defined wrt each of the variables and arrays denoting the observers with operator `next`.

3.3 Soundness of the Translation

Let $\mathcal{M}$ be the SMV transition system into which an OTS $\mathcal{S}$ is translated via a finite OTS $\mathcal{S}'$. Since there is a clear correspondence between $\mathcal{M}$ and $\mathcal{S}'$, any counterexample found in $\mathcal{M}$ is clearly a counterexample in $\mathcal{S}'$. Therefore, it suffices to prove that any counterexample found in $\mathcal{S}'$ is a counterexample in $\mathcal{S}$. We assume that $\mathcal{O}'$ is not empty.

Theorem 1 *Any counterexample in $\mathcal{S}'$ is also a counterexample in $\mathcal{S}$.*

We need two lemmas, which are as follows:

Lemma 1 $\forall \upsilon'_k, \upsilon'_{k+1} \in \Upsilon'. \forall \tau'_{d'_{j_1},\ldots,d'_{j_n}} \in \mathcal{T}'.$
$(\text{inside}_{\mathcal{S}'}(\upsilon'_{k+1}) \wedge (\upsilon'_{k+1} =_{\mathcal{S}'} \tau'_{d'_{j_1},\ldots,d'_{j_n}}(\upsilon'_k)))$
$\Rightarrow \text{inside}_{\mathcal{S}'}(\upsilon'_k)$

Lemma 2 $\forall \upsilon'_k, \upsilon'_{k+1} \in \Upsilon'. \forall \tau'_{d'_{j_1},\ldots,d'_{j_n}} \in \mathcal{T}'. \exists \upsilon_{k+1} \in \Upsilon.$
$(\text{inside}_{\mathcal{S}'}(\upsilon'_{k+1}) \wedge (\upsilon'_{k+1} =_{\mathcal{S}'} \tau'_{d'_{j_1},\ldots,d'_{j_n}}(\upsilon'_k)))$
$\Rightarrow ((\upsilon_{k+1} =_{\mathcal{S}} \tau_{d_{j_1},\ldots,d_{j_n}}(\upsilon'_k)) \wedge (\upsilon_{k+1} =_{\mathcal{S}'} \upsilon'_{k+1}))$

Proof (Theorem 1) Let υ'_n be an arbitrary reachable state wrt $\mathcal{S}'$ such that $\neg p'(\upsilon'_n)$. Let $\upsilon'_0, \upsilon'_1, \ldots, \upsilon'_n$ be an arbitrary execution fragment starting from an arbitrary initial state $\upsilon'_0 \in \mathcal{I}'$ and ending with υ'_n. From the definition of p', we have

$$\neg p'(\upsilon'_n) \Leftrightarrow (\text{inside}_{\mathcal{S}'}(\upsilon'_n) \wedge \neg p(\upsilon'_n)) \qquad (1)$$

From Lemma 1 and (1), $\text{inside}_{\mathcal{S}'}(\upsilon'_k)$ holds for every $k \in \{0, 1, \ldots, n\}$. Because $\mathcal{I}'$ is the same as $\mathcal{I}$, $\upsilon'_0 \in \mathcal{I}$. Besides, from Lemma 2, we can construct an execution fragment $\upsilon_0, \upsilon_1, \ldots, \upsilon_n$ of $\mathcal{S}$ such that $\upsilon_k =_{\mathcal{S}'} \upsilon'_k$ for every $k \in \{0, 1, \ldots, n\}$. From (1), $\neg p(\upsilon_n)$. □

Proof (Lemma 1) Let us consider arbitrary $\upsilon'_k, \upsilon'_{k+1} \in \Upsilon'$ and $\tau'_{d'_{j_1},\ldots,d'_{j_n}} \in \mathcal{T}'$ such that the premise holds. From the assumption, we have $\text{inside}_{\mathcal{S}'}(\tau'_{d'_{j_1},\ldots,d'_{j_n}}(\upsilon'_k))$. From this and the definitions of $\text{inside}_{\mathcal{S}'}$ and $\tau'_{d'_{j_1},\ldots,d'_{j_n}}$, we have

$\forall o'_{d_{i_1},\ldots,d_{i_m}} \in \mathcal{O}'.$
(**if** $d'_{j_1} \in D'_{j_1} \wedge \ldots \wedge d'_{j_n} \in D'_{j_n} \wedge \text{inside}_{\mathcal{S}'}(\upsilon'_k)$
then $o'_{d_{i_1},\ldots,d_{i_m}}(\tau_{d_{j_1},\ldots,d_{j_n}}(\upsilon'_k)) \in D'_i$
else $o'_{d_{i_1},\ldots,d_{i_m}}(\upsilon_{\text{out}}) \in D'_i$)

From this, since $o'_{d_{i_1},\ldots,d_{i_m}}(\upsilon_{\text{out}}) \notin D'_i$ and $\mathcal{O}'$ is not empty, we have $\text{inside}_{\mathcal{S}'}(\upsilon'_k)$. □

Proof (Lemma 2) Let us consider arbitrary $\upsilon'_k, \upsilon'_{k+1} \in \Upsilon'$ and $\tau'_{d'_{j_1},\ldots,d'_{j_n}} \in \mathcal{T}'$ such that the premise holds. From the assumption, we have $\text{inside}_{\mathcal{S}'}(\tau'_{d'_{j_1},\ldots,d'_{j_n}}(\upsilon'_{k+1}))$. From this and the definition of $\tau'_{d'_{j_1},\ldots,d'_{j_n}}$, $\tau'_{d'_{j_1},\ldots,d'_{j_n}}(\upsilon'_k)$ must be $\tau_{d_{j_1},\ldots,d_{j_n}}(\upsilon'_k)$. Let υ_{k+1} be $\tau_{d_{j_1},\ldots,d_{j_n}}(\upsilon'_k)$ as a witness. Clearly, $\upsilon_{k+1} =_{\mathcal{S}} \tau_{d_{j_1},\ldots,d_{j_n}}(\upsilon'_k)$, and from the assumption, we have $\upsilon_{k+1} =_{\mathcal{S}'} \upsilon'_{k+1}$. □

4 A Case Study: The NSPK Protocol

The NSPK authentication protocol[3] can be written as

Msg1	$p \to q$	:	$\mathcal{E}_q(n_p, p)$
Msg2	$q \to p$	:	$\mathcal{E}_p(n_p, n_q)$
Msg3	$p \to q$	:	$\mathcal{E}_q(n_q)$

We suppose that each principal is given a private/public key pair, and the public key is available to all principals but the private key to its owner only. A message m encrypted with the principal p's public key is denoted by $\mathcal{E}_p(m)$.

4.1 Modeling and Description of the Protocol

We suppose that there exist the intruder as well as trustable principals. In addition to the actions specified by the protocol, the intruder gleans as many nonces as possible and fakes messages based on gleaned nonces.

Principals are denoted by visible sort `Nat` for natural numbers; 0 denotes the intruder and positive integers trustable principals. Nonces are denoted by `Nat`. Messages are denoted by visible sort `Msg`; the data constructor of messages is `m`; $\mathtt{m}(t, k, n1, n2, p)$ denotes $\mathcal{E}_k(n1, p)$ if $t = 0$, $\mathcal{E}_k(n1, n2)$ if $t = 1$ and $\mathcal{E}_k(n1)$ if $t = 2$; given $\mathtt{m}(t, k, n1, n2, p)$, projections `tag`, `key`, `nonce1`, `nonce2` and `id` return t, k, $n1$, $n2$ and p. Used nonces are denoted by visible sort `UNonce`; the constructor of used nonces are `n`; $\mathtt{n}(n, p, q)$ denotes the used nonce n that has been generated by principal p to send to principal q; given $\mathtt{n}(n, p, q)$, `nonce`, `creator` and `receiver` return n, p and q. We use three kinds of lists, which are those of natural numbers (denoted by visible sort `Natlist`), those of messages (by visible sort `Msglist`) and those of used nonces (by visible sort `Nlist`); for either kind of lists `empty` denotes the empty list, `c` is the data constructor of non-empty lists and `isin` is the membership predicate; for lists of unused nonces `used` takes a nonce n and a list of unused nonces and checks if there exists a used nonce whose first argument is n in the list.

The OTS modeling the protocol consists of three observations and six parameterized transitions. The corresponding CafeOBJ observation and action operators are

```
bop usedNonces    : Sys -> Nlist
bop gleanedNonces : Sys -> Natlist
bop network       : Sys -> Msglist
bop sendMsg1 : Sys Nat Nat Nat -> Sys
bop sendMsg2 : Sys Nat Nat Msg -> Sys
bop sendMsg3 : Sys Nat Msg Msg -> Sys
bop fakeMsg1 : Sys Nat Nat Nat -> Sys
bop fakeMsg2 : Sys Nat Nat Nat -> Sys
bop fakeMsg3 : Sys Nat Nat     -> Sys
```

`Sys` is the hidden sort denoting the state space. Given a state s, `usedNonces`(s) is the list of used nonces in state s, `gleanedNonces`(s) is the list of nonces gleaned by the intruder in state s, and `network`(s) is the list of messages sent until state s, denoting the network. The first three action operators formalize sending messages following to the protocol, and the remaining the intruder's faking messages. `sendMsg2` is defined as

```
ceq usedNonces(sendMsg2(S,Q,N,M))
  = c(n(N,Q,id(M)),usedNonces(S))
  if c-sendMsg2(S,Q,N,M) .
ceq gleanedNonces(sendMsg2(S,Q,N,M))
  = c(N,gleanedNonces(S)) if c-sendMsg2(S,Q,N,M)
  and (Q == 0 or id(M) == 0).
ceq gleanedNonces(sendMsg2(S,Q,N,M))
  = gleanedNonces(S) if c-sendMsg2(S,Q,N,M)
  and not(Q == 0 or id(M) == 0).
ceq network(sendMsg2(S,Q,N,M))
  = c(m(1,id(M),nonce1(M),N,0),network(S))
  if c-sendMsg2(S,Q,N,M) .
ceq sendMsg2(S,Q,N,M) = S
  if not c-sendMsg2(S,Q,N,M) .
```

`c-sendMsg2(S,Q,N,M)` denotes the effective condition of the transition denoted by `sendMsg2`, defined as

```
isin(M,network(S)) and tag(M) == 0 and
key(M) == Q and not(used(N,usedNonces(S)))
```

`c-sendMsg2`(s, q, n, m) means that in a state s, there exists a Msg1 m in the network that is addressed to q, m is encrypted with q's public key, and a nonce n generated by q for replying to m is really fresh. If this condition holds, the Msg2 denoted by the term `m`$(1, \ldots)$ is put into the network and the nonce `N` used is put into the list of unused nonces. In addition to the condition, if the principal `Q` who generated `N` is the intruder or the public key `id(M)` used for `m`$(1, \ldots)$ is the intruder's, then `N` is put into the list of gleaned nonces. `fakeMsg2` is defined as

```
eq  usedNonces(fakeMsg2(S,Q,N1,N2))
  = usedNonces(S) .
eq  gleanedNonces(fakeMsg2(S,Q,N1,N2))
  = gleanedNonces(S) .
ceq network(fakeMsg2(S,Q,N1,N2))
  = c(m(1,Q,N1,N2,0),network(S))
  if c-fakeMsg2(S,Q,N1,N2) .
ceq fakeMsg2(S,Q,N1,N2) = S
  if not c-fakeMsg2(S,Q,N1,N2) .
```

`c-fakeMsg2(S,Q,N1,N2)` denotes the effective condition of the transition denoted by `fakeMsg2`, defined as

```
isin(N1,gleanedNonces(S))
and isin(N2,gleanedNonces(S))
```

The equations say that if two nonces are available to the intruder, the intruder can fake and send a Msg2. The remaining action operators are defined likewise.

One of the desired properties to verify for the protocol is the (nonce) secrecy property, which is expressed as

```
(isin(n(N,P,Q),usedNonces(S)) and isin(N,
gleanedNonces(S))) implies (P == 0 or Q == 0)
```

This means that an arbitrary nonce that the intruder can glean has been generated by the intruder or by trustable principals for sending to the intruder.

4.2 Model-Checking the Protocol

We specify that the number of principals involved is 3 (0, 1 and 2 are used), the number of nonces generated is 3 (0, 1 and 2 are used), the length of the list (for `usedNonces`) of used nonces is 2, the length of the list (for `gleanedNonces`) of gleaned nonces is 2 and the length of the list (for `network`) of messages is 4, and then translate the OTS/CafeOBJ specification of the protocol into an SMV specification with Chocolat/SMV.

The CafeOBJ module where visible sort `Msg` and its related operators and equations are declared is translated into this SMV module

```
MODULE Msg
VAR tag : 0 .. 2;  key : 0 .. 2;
 nonce1 : 0 .. 2;  nonce2 : 0 .. 2;  id : 0 .. 2;
```

The variables in module `Msg` correspond to the arguments of data constructor `m` for messages in the OTS/CafeOBJ specification. We also have the predicate `Msg_eq` that checks if two instances of module `Msg` are the same; `Msg_eq` is defined as

```
#define Msg_eq(m, n)  (m.tag = n.tag &
  m.key = n.key & m.nonce1 = n.nonce1 &
  m.nonce2 = n.nonce2 & m.id = n.id)
```

The CafeOBJ module where visible sort `Msglist` and its related operators and equations are declared is translated into this SMV module

```
MODULE Msglist
VAR  _c : 0 .. 5;  _ar : array 1 .. 4 of Msg;
```

Variable `_c` records how many elements of the array `_ar` are occupied; if `_c` is 0, no elements are used and if `_c` is 4, an overflow has occurred. Predicate `Msglist_isin_eq` checking if two instances of module `Msglist` are the same and predicate `Msglist_isin` checking if there exists an instance of module `Msg` in the array `_ar` of an instance of module `Msglist` are defined as

```
#define Msglist_isin_eq(v, e)  (Msg_eq(v, e))
#define Msglist_isin(e, a)
 (a._c >= 1 & Msglist_isin_eq(e, a._ar[1]) |
  a._c >= 2 & Msglist_isin_eq(e, a._ar[2]) |
  a._c >= 3 & Msglist_isin_eq(e, a._ar[3]) |
  a._c >= 4 & Msglist_isin_eq(e, a._ar[4]))
```

The CafeOBJ modules for visible sorts UNonce, Natlist and Nlist are translated into SMV modules likewise. Visible sorts Nat and Bool (the visible sort for truth values) are translated into (bounded) integers and truth values in SMV.

The CafeOBJ module where the OTS modeling the protocol is written is translated into the SMV module NSPK whose VAR section is

```
MODULE NSPK
VAR gleanedNonces : Natlist;  network : Msglist;
  usedNonces : Nlist;
  _va0 : 0 .. 2;  _va1 : 0 .. 2;  _va2 : Msg;
  _va3 : Msg;     _va4 : Msg;     _va5 : 0 .. 2;
  action : {fakeMsg1, fakeMsg2, fakeMsg3,
            sendMsg1, sendMsg2, sendMsg3};
```

The three CafeOBJ observation operators are translated into variables gleanedNonces, network and usedNonces. Variables _va0, ..., _va5 are used in assignments implementing state transitions as arbitrary values of their corresponding types. Variable action corresponds to an arbitrary one of the six action operators.

In the ASSIGN section of module NSPK, assignments implementing state transition are written. The assignments for variable action are

```
init(action) := fakeMsg1;
next(action) := {fakeMsg1, fakeMsg2, fakeMsg3,
                 sendMsg1, sendMsg2, sendMsg3};
```

which mean that the initial value of action is fakeMsg1 and a sate transition arbitrarily chooses one among the six values as its value. The assignments for variable _va2 are

```
init(_va2.tag) := 0; init(_va2.key) := 0;
init(_va2.nonce1) := 0; init(_va2.nonce2) := 0;
init(_va2.id) := 0;
next(_va2.tag) := {0, 1, 2};
next(_va2.key) := {0, 1, 2};
next(_va2.nonce1) := {0, 1, 2};
next(_va2.nonce2) := {0, 1, 2};
next(_va2.id) := {0, 1, 2};
```

Assignments for variables _va0, _va1, _va3, _va4 and _va5 are made likewise.

Some of the assignments for variable network look like

```
next(network._c) :=
case
 ...
 action = fakeMsg2 & Natlist_isin(_va1,gleanedNonces) &
  Natlist_isin(_va5,gleanedNonces) & TRUE :
  case network._c < 4 : network._c + 1; 1 : 5; esac;
 action = fakeMsg2 & ! (Natlist_isin(_va1,
  gleanedNonces) & Natlist_isin(_va5,gleanedNonces)) &
  TRUE : network._c;
 ...
 action = sendMsg2 & Msglist_isin(_va2,network)
  & _va2.tag = 0 & _va2.key = _va0 & ! Nlist_used(_va1,
  usedNonces) & TRUE : case network._c < 4 : network._c
  + 1; 1 : 5; esac;
 action = sendMsg2 & ! (Msglist_isin(_va2,network) &
  _va2.tag = 0 & _va2.key = _va0 & ! Nlist_used(_va1,
  usedNonces)) & TRUE : network._c;
 ...
esac;
next(network._ar[1].tag) :=
case
 ...
 action = fakeMsg2 & Natlist_isin(_va1,gleanedNonces) &
  Natlist_isin(_va5,gleanedNonces) & TRUE :
  case network._c = 4 : 0; network._c = 0 : 1 + 0; 1 :
  network._ar[1].tag; esac;
 action = fakeMsg2 & ! (Natlist_isin(_va1,
  gleanedNonces) & Natlist_isin(_va5,
  gleanedNonces)) & TRUE : network._ar[1].tag;
 ...
 action = sendMsg2 & Msglist_isin(_va2,network)
  & _va2.tag = 0 & _va2.key = _va0 & ! Nlist_used(_va1,
  usedNonces) & TRUE : case network._c = 4 : 0;
  network._c = 0 : 1 + 0; 1 : network._ar[1].tag; esac;
 action = sendMsg2 & ! (Msglist_isin(_va2,network)
  & _va2.tag = 0 & _va2.key = _va0 & ! Nlist_used(_va1,
  usedNonces)) & TRUE : network._ar[1].tag;
 ...
esac;
```

The omitted parts are made likewise and so are the remaining assignments for variable network. Assignments for the remaining variables are also made likewise.

We used NuSMV (see nusmv.irst.itc.it), an implementation of SMV, to model-check the SMV specification translated from the OTS/CafeOBJ specification of the protocol wrt the secrecy property. It took about 15 minutes for NuSMV to model-check the SMV specification on a computer with 4.3GHz Pentium 4 and 1GB memory and report on a counterexample. The counterexample says that four state transitions lead to a state where the secrecy property does not hold.

5 Conclusion

We have described how OTS/CafeOBJ specifications are translated into SMV ones and the proof that the translation has the desired property that any counterexample found in the SMV specification translated from an OTS/CafeOBJ one is also a counterexample in the OTS/CafeOBJ one. We have also reported on the case study that Chocolat/SMV is applied to the NSPK authentication protocol.

References

[1] R. Diaconescu and K. Futatsugi. *CafeOBJ report*. AMAST Series in Computing, 6. World Scientific, 1998.

[2] K. L. McMillan. *Symolic Model Checking: An Approach to the State Explosion Problem*. Kluwer, 1993.

[3] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *CACM*, 21(12):993–999, 1978.

[4] K. Ogata and K. Futatsugi. Flaw and modification of the iKP electronic payment protocols. *IPL*, 86:57–62, 2003.

[5] K. Ogata and K. Futatsugi. Formal analysis of the iKP electronic payment protocols. In *1st ISSS*, LNCS 2609, pages 441–460. Springer, 2003.

[6] K. Ogata and K. Futatsugi. Proof scores in the OTS/CafeOBJ method. In *6th FMOODS*, LNCS 2884, pages 170–184. Springer, 2003.