Relations between Formal and Semi-formal Specification Systems: a Case Study

Fazio Nelly* fazio@giove.dipmat.unict.it

Nicolosi Antonio* nicolosi@giove.dipmat.unict.it

September 4, 2000

Abstract

The aim of this paper is to investigate the relations between formal and semi-formal specification systems, comparing the models obtained analyzing a cryptographic protocol — the Yahalom key distribution protocol.
We first model the protocol with a distributed Gurevich's Abstract State Machine, and then move to its validation with ASMGofer.
This implementation shows that the attack to Paulson's variant to the Yahalom protocol does not affect the correct version.
Finally, from the ASM model we obtain straightforwardly the UML model: in transposing the former model into the latter, we underline the correspondence between the basic constructing elements of the two specification systems.

1 The Problem: a Cryptographic Protocol

1.1 Introduction

Communication over the net is becoming more and more common. Unfortunately, the Internet is an unreliable environment, since each message has to go through many intermediate computers to work out its way toward its destination.

Cryptographic protocols are sequences of messages that a number of participants exchange with each other to ensure that the subsequent communication satisfies some properties, which are the so called *goals* of the protocol.

An execution of all the steps specified by the cryptographic protocol is typically said a *run* of the protocol.

Typical goals of a cryptographic protocol are:

- *confidentiality*: the communication starting after a run of the protocol must be unintelligible to eavesdroppers;
- *authentication*: at the end of a run, each participant can be sure of the identity of the other parties;
- *key distribution*: the aim of the protocol is to deliver some important data (typically a key) only to the parties involved in the message exchange.

*Dipartimento di Matematica e Informatica - Università di Catania - ITALY

One of the basic tool used in cryptographic protocols is *encryption*, that is the use of a key to conceal information in such a way that the data can only be understood by who knows the decryption key.

For the encryption to be effective, the key must be carefully chosen and kept secret. It also should be used as little as possible, since each encryption reveals some information about the key that an hostile agent can collect to break that key.

It is for these reasons that each agent holds a special key, the so called *master key*, which is used only to exchange temporary keys. Such keys are typically used for one communication session only and are therefore called *session keys*.

1.2 The Yahalom Protocol

The Yahalom protocol is a key distribution protocol that also guarantees authentication. It assumes a shared-key cryptosystem, in which each participant shares a master key with a trusted party, the **Key Distribution Server**.

The role of the Server is to generate new session keys each time it receives a request from an agent: it is "trusted" in the sense that it will always behaves as it is expected to.

The aim of the protocol is to enable every two agents to agree on a session key — *key distribution* — to be used to ensure the secrecy of the subsequent communication. It also guarantees each party that the other one has been involved in current run — *authentication*.

One of the major problem of protocols is that each participant can be involved in several runs at once, possibly with different roles. For the agent to correctly associate every message he/she receives to the right run, it is useful to include in each message an identification number: a nonce.

A *nonce* typically is a long, random number to be used only once: it is assumed that each participant is able to generate his/her own nonces, and keeps track of the nonces so far generated. In some cases each nonce is associated to a recipient — the agent who is assumed to receive the message with that nonce.

Figure 1 presents the Yahalom protocol that distributes a session key K_{ab} to parties A and B with the help of the Server $\mathcal{S}$.

1. $A \longrightarrow B : A, N_a$
2. $B \longrightarrow \mathcal{S} : B, \{A, N_a, N_b\}_{K_b}$
3. $\mathcal{S} \longrightarrow A : \{B, K_{ab}, N_a, N_b\}_{K_a}, \{A, K_{ab}\}_{K_b}$
4. $A \longrightarrow B : \{A, K_{ab}\}_{K_b}, \{N_b\}_{K_{ab}}$

Figure 1: The Yahalom Protocol

1.3 The Attack

Although the protocol has only four messages, it is very difficult to understand and analyze in depth. The complexity of the protocol was pointed out by Paulson [1], who presented a slight variation which is easy to attack by a malicious agent — the *Spy*.

The flawed version (figure 2) is different from the original one only for the second message — B's nonce is sent in clear:

To understand why the version in figure 2 is faulty, it is necessary to discuss better the goals of the protocol.

As mentioned above, the Yahalom protocol aims to distribute a session key K_{ab} between parties A and B. But session keys are vulnerable, and after a while the *Spy*

1. $A \longrightarrow B : A, N_a$
2. $B \longrightarrow \mathcal{S} : B, N_b, \{A, N_a\}_{K_b}$
3. $\mathcal{S} \longrightarrow A : \{B, K_{ab}, N_a, N_b\}_{K_a}, \{A, K_{ab}\}_{K_b}$
4. $A \longrightarrow B : \{A, K_{ab}\}_{K_b}, \{N_b\}_{K_{ab}}$

Figure 2: Flawed Version of the Yahalom Protocol

can manage to crack some message, learning some session key K_{ab}. Typically, this happens when the communication between A and B is over — the key K_{ab} is now *old*, and the *Spy* has not earned that much.

But what if the *Spy* could convince B to believe that A wants to start a new communication session with the **old** session key K_{ab}? B will send confidential information to A, and the *Spy* will learn it — a breach of security.

Figure 3 illustrates the sequence of messages that leads to a fake authentication between A and B:

1. $\mathit{Spy} \longrightarrow B : A, N_x$
2. $B \longrightarrow \mathcal{S} : B, N'_b, \{A, N_x\}_{K_b}$
3. $\mathcal{S} \longrightarrow A : \{B, K'_{ab}, N_x, N'_b\}_{K_a}, \{A, K'_{ab}\}_{K_b}$
4. $\mathit{Spy} \longrightarrow B : \{A, K_{ab}\}_{K_b}, \{N'_b\}_{K_{ab}}$

Figure 3: The Attack

Despite the little difference between the two variants, the correct version thwarts this attack. This will be proved through the validation (with ASMGofer) of the ASM model for the protocol.

The key point is that B identifies the freshness of the session key with the freshness of the nonce encrypted with that session key in the fourth message of the sequence.

The *Spy* can always forces B to generate a fresh nonce — sending him/her the first message of the sequence. But only in the flawed version the *Spy* can learn this fresh nonce, and that enables him/her to persuades B to accept the old session key for the subsequent communication with A.

2 The ASM Model

The first specification system used to analyze the Yahalom protocol are the Abstract State Machines of Gurevich [2].

The model is split into 4 main modules:

1. the network environment;
2. the malicious *Spy*;
3. the Agents and the Server $\mathcal{S}$ (for the correct version);
4. the Agents and the Server $\mathcal{S}$ (for the flawed version);

Each component is organized in two parts:

- *the signature*: it is the static description of the model — a snapshot of the system. It is made up by elements, grouped into *universes* and correlated with each other through (dynamic) *functions*.

- *the rules*: they describe the dynamic behavior of the system, specifying how the functions of the signature change over the time. Typically, they are "conditional updates", i.e. rules of the form:

$$\texttt{if } \mathit{Cond} \texttt{ then } \mathit{Update} \texttt{ end-if}$$

2.1 Modeling the Net

In this section we describe the environment in which the Yahalom Protocol is intended to be executed.

This model has been adapted from the one proposed in [3].

2.1.1 Signature

Since the **universes** constitutes the common ground on which all the components of the model are built, they are all presented here, and they will not be repeated in the remaining sections.

- $AGENT$: set of active agents over the network
- $NONCE$: set of nonces
- $SHRKEY$: set of master keys
- $SESKEY$: set of session keys
- KEY: set of both the possible kinds of keys:

$$KEY \triangleq SHRKEY \cup SESKEY$$

- $COMPONENT$: set of elements that can appear in a message. It is defined as

$$COMPONENT \triangleq AGENT \cup NONCE \cup SESKEY$$

- $MESSAGE$: set of all the possible messages that can be sent over the network. It is defined inductively as the smallest possible set that includes $COMPONENT$ and that is closed under encryption and message concatenation:

$$COMPONENT \subset MESSAGE$$

$$\frac{m \in MESSAGE,\ k \in KEY}{(m\#k) \in MESSAGE}$$

$$\frac{m1 \in MESSAGE,\ m2 \in MESSAGE}{(m1\hat{\ }m2) \in MESSAGE}$$

- $PACKET$: set of packages routed over the net, collectively referred to as the "traffic". Each packet contains just one message, and specifies a *sender* and a *receiver*.

Beside these problem-specific universes, the functions in the model refer to a pre-defined ASM universe — the set *Boole*, with the two elements `True` and `False` .

To model the capability of each Agent to generate nonces to be sent in messages, define the monitored function

$$nonceGenerator : (AGENT \times AGENT) \longrightarrow NONCE$$

which is expected to return a different nonce each time it is invoked (*non-clashing constraint*).

Since each Agent has to record all the nonces he/she has generated along with the intended recipient of each of them, we need the following two dynamic functions (controlled by the Agent):

$$nonce : AGENT \longrightarrow \mathcal{P}(NONCE)$$

$$recipient : NONCE \longrightarrow AGENT$$

To have a secure communication with the Server $\mathcal{S}$, Agents share a key with this trusted party:

$$shrkey : AGENT \longrightarrow SHRKEY$$

To model message routing, each Agent needs a channel in which he/she can find incoming messages:

$$receive : AGENT \longrightarrow PACKET$$

For Agents to fully take advantage of the use of nonces in messages, it is important to distinguish fresh nonces from old ones: for this purpose we introduce the following function (controlled by Agents):

$$isFresh : NONCE \longrightarrow Boole$$

To create fresh session keys every time it is asked to, the Server $\mathcal{S}$ needs the monitored function

$$sesKeyGenerator : (AGENT \times AGENT) \longrightarrow SESKEY$$

which is expected to satisfy the *non-clashing constraint* just as its counterpart for nonces.

Both kinds of keys are suitable to encrypt and decrypt messages; we model such operations through the functions:

$$crypt : (MESSAGE \times KEY) \longrightarrow MESSAGE$$

$$decrypt : (MESSAGE \times KEY) \longrightarrow MESSAGE$$

The function

$$traffic : PACKET \longrightarrow (AGENT \times AGENT \times MESSAGE)$$

models the network traffic as a set of packets as discussed above.

Finally, the following function extrapolates the content of a message:

$$cont : (AGENT \times AGENT \times MESSAGE) \longrightarrow MESSAGE$$

2.1.2 Rules

To complete the modeling of the net, we just need to give some macros and a rule.

The `routeMsg` rule in figure 4 delivers a random message in the traffic to each Agent.

```
Rule routeMsg:
do forall $X \in AGENT$
    choose $p \in PACKET$
        do $receive(X) := p$
    end-choose
end-do
```

Figure 4: The routeMsg rule

Macros related to the managing of the network are given in figure 5: they are useful for the rules in the other sections.

```
send (X, Y, msg) ≡
    extend $PACKET$ with $p$
        $traffic(p) := (X, Y, msg)$
    end-extend

clear (p) ≡
    $receive(self) :=$ undef

addNonce (A, N) ≡
    $nonce(A) := nonce(A) \cup \{N\}$
```

Figure 5: Macros for the net

2.2 Modeling the *Spy*

For a model of a cryptographic protocol to be able to fully analyze the protocol itself, it is crucial the way the *Spy* is modeled. The more powerful the *Spy*, the finer the analysis.

The *Spy* model we describe here is inspired from [3].

2.2.1 Signature

Almost all the functions used by the *Spy* are, in fact, nullary functions, i.e. variables.

To be as realistic as possible, we consider, beside the *Spy*, the presence of *compromised* Agents — people whose key has fallen in the hands of the *Spy*:

$$compr : \mathcal{P}(AGENT)$$

As a consequence, the set of keys known to *Spy* contains not only the key the *Spy* shares with the Server $\mathcal{S}$, but also all the compromised Agents' keys:

$$keyspy : \mathcal{P}(KEY)$$

The *Spy* we are considering is able to intercept every single packet in the traffic. From all these pieces of data, the *Spy* tries to extract as much information as possible, breaking every encryption he/she can through the use of keys in *keyspy*. This operation is modeled through the function

$$newSpyKnw : \mathcal{P}(MESSAGE) \times \mathcal{P}(KEY) \longrightarrow \mathcal{P}(COMPONENT) \times \mathcal{P}(MESSAGE)$$

that, given a set of messages and a set of keys, returns both the set of intelligible message components, and the set of message fragments encrypted under unknown keys.

We will see in the Rule section how the first part of this "knowledge" is used to augment the set of known keys (*keyspy*), while the second part is collected in the following variable, containing all "not-understood" message fragments:

$$coanalz : \mathcal{P}(MESSAGE)$$

The most dangerous activity of the *Spy* is synthesizing new messages from the components he/she had stolen from the traffic, the messages in *coanalz* and the keys in *keyspy*:

$$synth : \mathcal{P}(COMPONENT) \times \mathcal{P}(MESSAGE) \times \mathcal{P}(KEY) \longrightarrow \mathcal{P}(MESSAGE)$$

Finally, we need to model the possibility of a burglary — the *Spy* could occasionally manage to corrupt the Server $\mathcal{S}$, obtaining from it some session keys. This possible channel of communication is modeled with the variable:

$$notes : \mathcal{P}(KEY)$$

2.2.2 Rules

To be as generic as possible, we thin of the *Spy* as a corrupted insider, i.e. an Agent who is trusted by the other Agents, which don't know his/her malicious intents. Consequently, the *Spy* can also act as a normal Agent, but here we consider only his/her illegal activity. This activity is described in figure 6.

```
Rule spyIllegal:
do in-parallel
    updateSpyKnw
    destroyInfo
    addFakeInfo
end-par
```

Figure 6: The spyIllegal rule

The `updateSpyKwn` macro model the *Spy* trying to learn new information from the traffic. The relevant informations are, of course, the session keys and the nonces. These two kinds of data are collected in two different set, respectively *keyspy* and *nonce(Spy)*. While nonces can only be discovered through the *newSpyKnw* function, new session keys can be obtained in two additional ways:

- from the Server $\mathcal{S}$, as explained in presenting the variable *notes*;
- through "brute force" cracking of session keys' certificate.

This last point is crucial: since we are dealing with a *key distribution* protocol, it is normal to foresee that old session keys will, before or after, fall in the hands of the *Spy*. To model this, we let the *notes* variable be monitored by the Server $\mathcal{S}$ and by the *Spy*, who can, every now and then, break one of the "certificate" issued by the Server , obtaining the session key thereby specified.

The `destroyInfo` macro simply chooses at random a message in the traffic and destroys it. This macro is also useful to model the unreliability of the network.

The `addFakeInfo` macro builds all the fake messages it can and sends them to random Agents.

These macros are illustrated in figure 7.

2.3 Modeling the Protocol

In this section we introduce the modules for the **Key Distribution Server** and for Agents. Note that, as mentioned discussing the *Spy* model, the *Spy* is an agent too; therefore, the module thereby described is an extension of the agent's module illustrated below.

2.3.1 Signature

In modeling the Server $\mathcal{S}$, we must consider the possibility of a burglary (see the discussion about the *notes* variable in the *Spy* section). We do this through the monitored nullary predicate:

$$burglary : Boole$$

As for the agents, we need two special predicates.

The first one is meant to model an agent wishing to initiate a new run of the protocol with another agent:

$$wish2init : AGENT \times AGENT \longrightarrow Boole$$

```
**updateSpyKnw** ≡
    let $netTraffic = cont(traffic(PACKET))$
       $(analz, notAnalz) = newSpyKwn(netTraffic, keyspy)$,
       $newNonces = analz \cap NONCE$,
       $newKeys = (analz \cap KEY) \cup notes$,
        do in-parallel
           $nonce(Spy) := nonce(Spy) \cup newNonces$
           $keyspy := keyspy \cup newKeys$
           $coanalz := coanalz \cup notAnalz$
           $notes := \emptyset$
        end-par
    end-let

**destroyInfo** ≡
    choose $p \in PACKET$
       $clear(p)$
    end-choose

**addFakeInfo** ≡
    let $fakeMsg = synth(AGENT \cup keyspy \cup nonce(Spy)$,
           $coanalz, keyspy)$
        do forall $X \in AGENT$
            choose $msg \in fakeMsg$
               $send(Spy, X, msg)$
            end-choose
        end-do
    end-let
```

Figure 7: Macros for the spy

The second predicate models the successful end of a protocol run between two agents:

$$authOK : AGENT \times AGENT \longrightarrow Boole$$

2.3.2 Rules

The Server module is made up of a single rule. Briefly, the Server $\mathcal{S}$ keeps waiting for requests, and answers each incoming message creating a new session key and issuing key *certificates* (i.e. encrypted messages that guarantee the authenticity of the session key) — one for each of the two parties involved in that run of the protocol. In doing this, $\mathcal{S}$ occasionally hands out the session key to the *Spy* — a *burglary*. This is illustrated in figure 8.

The Agents activity is essentially the parallel execution of the four rules in figure 9.

Each rule checks if it is necessary to send a new message or if a protocol run has just been successfully completed.

These rules make use of the following predicate to determine which kind of message they have received (if any):

$$received(A, X.Y.msg) \iff traffic(receive(A)) = (X, Y, msg)$$

```
Rule serverActivity:
    if $received(Self, \_.Self.\{Y, msg\}) \wedge$
      $decrypt(msg, shrKey(Y)) = \{X, N_1, N_2\}$ then
          let $K_{xy} = sesKeyGenerator(X, Y)$,
            $K_x = shrKey(X)$,
            $K_y = shrKey(Y)$
            $send(Self, X, (\{Y, K_{xy}, N_1, N_2\}_{K_x}, \{X, K_{xy}\}_{K_y}))$
            if $burglary$ then
              $notes := notes \cup \{K_{xy}\}$
            end-if
          end-let
      $clear(receive(Self))$
    end-if
```

Figure 8: The serverActivity rule

```
Rule yahalom1:
if $wish2init(Self, X)$ then
    let $NewNonce = nonce(Self, X)$
        do in-parallel
            $recipient(NewNonce) := X$
            $isFresh(NewNonce) :=$ True
            $addNonce(Self, NewNonce)$
            $send(Self, X, \{Self, NewNonce\})$
            $wish2init(Self, X) :=$ False
        end-par
    end-let
end-if

Rule yahalom2:
if $received(Self, \_.Self.\{X, N_0\})$ then
    let $NewNonce = nonce(Self, X)$
        $K_{self} = shrKey(Self)$
        do in-parallel
            $recipient(NewNonce) := X$
            $isFresh(NewNonce) :=$ True
            $addNonce(Self, NewNonce)$
            $send(Self, Server, \{Self,$
                $\{X, N_0, NewNonce\}_{K_{self}}\})$
        end-par
    end-let
    $clear(receive(Self))$
end-if

Rule yahalom3:
if $received(Self, \_.Self.\{msg1, msg2\}) \wedge$
    $decrypt(msg1, shrKey(Self)) =$
        $\{X, K_{xy}, N_1, N_2\}$ then
    if $N_1 \in nonce(Self) \wedge$
        $recipient(N_1) = X \wedge$
        $isFresh(N_1)$ then
        $send(Self, X, \{msg2, \{N_2\}_{K_{xy}}\})$
    end-if
    $clear(receive(Self))$
end-if

Rule yahalom4:
if $received(Self, \_.Self.\{msg1, msg2\}) \wedge$
    $decrypt(msg1, shrKey(Self)) = \{X, K_{xy}\}$
    $\wedge\ decrypt(msg2, K_{xy}) = \{N_0\}$ then
    if $N_0 \in nonce(Self) \wedge$
        $recipient(N_0) = X \wedge$
        $isFresh(N_0)$ then
            do in-parallel
                $isFresh(N_0) :=$ False
                $authOK(X, Self) :=$ True
            end-par
    end-if
    $clear(receive(Self))$
end-if
```

Figure 9: The four rules of the agent module

2.4 Modeling the Flawed Version

To complete the ASM model for the Yahalom protocol we need to present the flawed version.

2.4.1 Signature

All the universes and functions needed in this subsection have already been introduced above.

2.4.2 Rules

The only difference between the flawed version and the correct one is the second message of the sequence (see the section about the attack for more details).

As a consequence, we just need to slightly modify the rules that manage this kind of message, to obtain the two rules in figure 10 and 11.

```
Rule flawedServerActivity:
    if $received(Self, \_.Self.\{Y, N_2, msg\}) \wedge$
      $decrypt(msg, shrKey(Y)) = \{X, N_1\}$ then
          let $K_{xy} = sesKeyGenerator(X, Y)$,
              $K_x = shrKey(X)$,
              $K_y = shrKey(Y)$
              $send(Self, X, (\{Y, K_{xy}, N_1, N_2\}_{K_x}, \{X, K_{xy}\}_{K_y}))$
               if $burglary$ then
                  $notes := notes \cup \{K_{xy}\}$
               end-if
          end-let
      $clear(receive(Self))$
    end-if
```

Figure 10: The flawedServerActivity rule

```
Rule flawedYahalom2:
if $received(Self, \_.Self.\{X, N_0\})$ then
    let $NewNonce = nonce(Self, X)$
        $K_{self} = shrKey(Self)$
         do in-parallel
            $recipient(NewNonce) := X$
            $isFresh(NewNonce) :=$ True
            $addNonce(Self, NewNonce)$
            $send(Self, Server, \{Self, NewNonce,$
                $\{X, N_0,\}_{K_{self}}\})$
         end-par
    end-let
   $clear(receive(Self))$
end-if
```

Figure 11: The flawedYahalom2 rule

3 The ASMGofer Implementation

To validate the model obtained in the previous section, we have implemented it using the ASMGofer system.

Beside some minor problems, pointed out later on, this passage has been quite straightforward.

The implementation has been split into several different files (`.gs` stands for "gofer script").

- `univs.gs`: contains the definition of all the universes;
- `net.gs`: contains all the functions for the net module;
- `Spy.gs`: implements the *Spy*, just as presented above;
- `WinnerSpy.gs`: implements a specialized version of the *Spy*, useful to illustrate the attack to the flawed version of the protocol;
- `Yahalom.gs`: is the ASMGofer code for the Agents' and Server's modules;
- `YahFlawed.gs`: contains the flawed version of the protocol;
- `scenario_.gs`: contains a possible scenario of the network, specifying the number of agent, the set of compromised agents and other parameters.

3.1 Implementation details

We now come to discuss some problematic points of the implementation.

In transposing the ASM universes in their ASMGofer counterpart — `data types` — we needed to model a sort of subtyping, to express such relation between universes like

$$KEY \triangleq SHRKEY \cup SESKEY$$

and

$$COMPONENT \triangleq AGENT \cup NONCE \cup SESKEY$$

This has been accomplished through the use of `data type constructors`: adding an uppercase letter before each subtype, the system is able to correctly recognize the relation between the various types. Here is the code:

```
data AGENT = AGENT Int | Server | Spy
data NONCE = NONCE Int
data SHRKEY = SHRKEY Int | SHRSpy
data SESKEY = SESKEY Int
data KEY = H SHRKEY | E SESKEY
data COMPONENT = A AGENT | N NONCE | S SESKEY
```

The ASMGofer type also comes in hands in implementing the *MESSAGE* universe. Since it supports recursive types, the definition is simply:

```
data MESSAGE = C COMPONENT | MESSAGE :#: KEY | MESSAGE :^: MESSAGE
```

3.2 Additional functions of the implementation

Here follows a list of the functions which have been included in the implementation for technical reasons.

- **shrHolder** (in `net.gs`): inverse of the shrKey function;
- **sesHolder** (in `net.gs`): associates each session key generated by the Server $\mathcal{S}$ to its intended recipients;
- **dest** (in `net.gs`): extrapolates the receiver of a message from an *extended message* (i.e. the result of the $traffic$ function);
- **startProtocol** (in `net.gs`): macro used within the ASMGofer interpreter to start a new protocol run between the two Agents specified as arguments;

- **probActivity** (in `net.gs`): randomized function used to implement monitored predicates that every now and then should be `True` (e.g. *burglary*);
- **yahalomMain** (in `Yahalom.gs` and `YahFlawed.gs`): macro used within the interpreter to let the whole system evolve;
- **agentActivity** (in `Yahalom.gs` and `YahFlawed.gs`): macro to parallelize the activity of the Agents;
- **synth_** (in `Spy.gs`): these rules are used to code the `addFakeInfo` rule of the *Spy* module. Instead of generating the whole *fakeMsg* set and then extracting a random element from it, we just generate a *single* random element for each kinds of possible fake messages;
- **synth_** (in `WinnerSpy.gs`): these rules make the difference between the general *Spy* and the specialized version for the attack. Instead of synthesizing all the possible kinds of messages, we generate only the two kinds of messages necessary for the attack. The remaining job (choosing the right moment to send each message, ...) is done by the different implementation of *spyIllegal*.

3.3 The simulation of the attack

We now illustrate how to use the ASMGofer interpreter. We will use the following project files:

- `yahalom.p`: project file to test the Yahalom protocol with the full module for the *Spy*:

```
yahalom.p ≡
        univs.gs
        scenario1.gs
        net.gs
        Spy.gs
        Yahalom.gs
```

- `yahflawed.p`: project file to test the flawed version of the protocol with the full module for the *Spy*:

```
yahflawed.p ≡
        univs.gs
        scenario1.gs
        net.gs
        Spy.gs
        YahFlawed.gs
```

- `attack2yahalom.p`: project file to try (without success) to attack the correct version of the protocol:

```
attack2yahalom.p ≡
        univs.gs
        scenario2.gs
        net.gs
        WinnerSpy.gs
        Yahalom.gs
```

- `attack2yahflawed.p`: project file to try (successfully) to attack the flawed version of the protocol:

```
attack2yahflawed.p ≡
        univs.gs
        scenario2.gs
        net.gs
        WinnerSpy.gs
        YahFlawed.gs
```

Let's see how to proceed to simulate the attack to the flawed version. From within the interpreter, type:

```
? :p attack2yahflawed.p
```

Now set AGENT 1 wishing to initiate a protocol run with AGENT 2:

```
? fire1 (startProtocol (AGENT 1, AGENT 2))
```

Then execute the main rule a few times:

```
? fire1 yahalomMain
```

To see what is going on, we can inspect the traffic over the net, typing:

```
? assocs traffic
[(PACKET 3,(AGENT 2,Server,
            (C (A (AGENT 2)) :^: C (N (NONCE 2))) :^: ((C (A (AGENT 1)) :^:
             C (N (NONCE 0))) :#: H (SHRKEY 2))))]
```

Next, continue to fire the `yahalomMain` rule until we get authentication:

```
? fire1 yahalomMain
"DISTRIBUTION OK
Initiator: AGENT 1
Responder: AGENT 2
Session key: SESKEY 4
Session-key created for: (AGENT 1,AGENT 2)"
```

The set of messages that the *Spy* has not understood so far is showed by entering:

```
? coanalz
[(C (A (AGENT 1)) :^: C (N (NONCE 0))) :#: H (SHRKEY 2),
(C (A (AGENT 1)) :^: C (S (SESKEY 4))) :#: H (SHRKEY 2),
(((C (A (AGENT 2)) :^: C (S (SESKEY 4))) :^:
   C (N (NONCE 0))) :^: C (N (NONCE 2))) :#: H (SHRKEY 1),
C (N (NONCE 2)) :#: E (SESKEY 4)]
```

On the other hand, so far the *Spy* has been able to intercept the following nonces:

```
? nonce Spy
[NONCE 0, NONCE 2]
```

and knows the following keys:

```
? keyspy
[H SHRSpy]
```

Now, we have to wait until the *Spy* obtains the old session key:

```
? myUntil (not (null [E k| (E k) <- keyspy])) yahalomMain
"Oops: The Spy obtained the session key SESKEY 4 breaking the message
(C (A (AGENT 1)) :^: C (S (SESKEY 4))) :#: H (SHRKEY 2)"
```

Inspecting *keyspy* now we get:

```
? keyspy
[E (SESKEY 4), H SHRSpy]
```

Finally, execute the main rule until we reobtain a (fake) authentication:

```
? fire1 yahalomMain
"DISTRIBUTION OK
Initiator: AGENT 1
Responder: AGENT 2
Session key: SESKEY 4
Session-key created for: (AGENT 1,AGENT 2)"
```

To see that the correct version thwarts this attack, begin with

```
? :p attack2yahalom.p
```

Then proceed as above, but . . . we won't never reach the fake authentication!

The point is that the *Spy* can not obtain, in the correct version, the fresh nonce of the AGENT 2. If we add this nonce to *nonceSpy* and then fire again the main rule, we will reach the fake authentication.

```
? fire1 (do nonce Spy := (NONCE 8) : nonce Spy)
? myUntil (authOK(AGENT 1, AGENT 2)) yahalomMain
"DISTRIBUTION OK
Initiator: AGENT 1
Responder: AGENT 2
Session key: SESKEY 4
Session-key created for: (AGENT 1,AGENT 2)"
```

4 The UML Model

The last model of our protocol is a UML model.

It is made up of seven diagrams, presented in the following subsections.

4.1 The class diagram

This diagram gives the static aspect of the system. Therefore, it serves the same purpose as the signature in ASM models.

Consequently, it is natural to work out the class diagram for the Yahalom protocol from the ASM signature presented early in the paper.

In particular, each universe maps to a class and the relations between the universes induce the class hierarchy. Besides, each function is transposed either in an attribute of the class corresponding to its domain, or in a suitable association between classes.

4.2 The sequence diagrams

Sequence diagrams are used to sketch the typical events flow for a system.

For our problem, we point out two such diagrams: the *YahalomSequence* diagram, to illustrate a generic protocol run, and the *WinnerSpy* diagram presenting the attack to the flawed version.

4.3 The statechart and activity diagrams

Statechart diagrams and activity diagrams both models the dynamic aspects of the system. Essentially, they are duals point of view for the analysis of the problem at hand.

Consequently, they are obtained elaborating, in different ways, the rules of the ASM model.

The **statechart diagram** is obtained focusing on the states that each component of the system can reach, and visualizing the ASM rules as transitions between states of the form :

monitored predicate [*other conditions*] /*body of the rule*

The **activity diagram** is obtained focusing on the activities performed by each component of the system in its own life cycle. Essentially, they are obtained from the body of the ASM rules splitting actions into coherent activities.

References

[1] L. C. Paulson “Relations between secrets: Two formal analyses of the Yahalom protocol” *Journal of Computer Security.*

[2] Y. Gurevich “The ASM Guide” *Guide.*

[3] G. Bella, E. Riccobene “A realistic environment for crypto-protocol analyses by ASMs” *Informatik.*