AVISPA

www.avispa-project.org

IST-2001-39252

Automated Validation of Internet Security Protocols and Applications

HLPSL Tutorial

A Beginner's Guide
to
Modelling and Analysing Internet Security Protocols

The AVISPA team

Document Version: 1.1
June 30, 2006

Information Society Technologies

Project funded by the European Community under the
Information Society Technologies Programme (1998-2002)

Contents

1 HLPSL Basics 3
1.1 Using the AVISPA Tool . . . 3
1.2 Basic Roles . . . 5
1.3 Transitions . . . 6
1.4 Composed Roles . . . 7

2 HLPSL Examples 9
2.1 Example 1 - from Alice-Bob notation to HLPSL specification . . . 9
2.2 Example 2 - common errors, untrusted agents, attack traces . . . 14
2.2.1 Modelling Tips and Pitfalls . . . 18
2.2.2 Discussion and Analysis Results . . . 22
2.3 Example 3 - security goals . . . 25
2.3.1 Discussion and Analysis Results . . . 28
2.4 Example 4 - Algebraic Operators . . . 34

3 HLPSL Tips 41
3.1 Priming Variables . . . 41
3.2 Witness and Request . . . 41
3.3 Secrecy . . . 42
3.4 Constants and Variables . . . 42
3.5 Concatenation (.) and Commas (,) . . . 43
3.6 Exploring executability of your model . . . 43
3.7 Detecting Replay Attacks . . . 44
3.8 Instantiating Sessions . . . 44
3.9 Function Results . . . 47
3.10 Declaring Channels . . . 47

4 Questions and answers about HLPSL 48

A Symbols and Keywords 49

Introduction

The AVISPA Tool provides a suite of applications for building and analysing formal models of security protocols. Protocol models are written in the *High Level Protocol Specification Language*, or HLPSL [3, 9]. The aim of this tutorial is to provide some advice on constructing protocol models in HLPSL.

In addition to this tutorial, the AVISPA Package User Manual [5] is another useful resource for beginners to HLPSL. Please refer to this manual if you require further information on HLPSL or any of the tools discussed throughout this tutorial.

Organisation of this tutorial: Section 1 contains a very basic introduction to what HLPSL looks like and how it is used.

Section 2 contains four introductory examples that illustrate modelling with HLPSL. In discussing the examples, we attempt to show both correct solutions and possible pitfalls that modellers should be aware of.

Section 3 contains a number of tips useful for writing or reading HLPSL specifications.

Finally, Section 4 provides a list of questions and answers about HLPSL,
followed by an appendix containing a list of HLPSL keywords and symbols.

1 HLPSL Basics

AVISPA provides a language called the *High Level Protocol Specification Language (HLPSL)* [3, 9] for describing security protocols and specifying their intended security properties, as well as a set of tools to formally validate them.

1.1 Using the AVISPA Tool

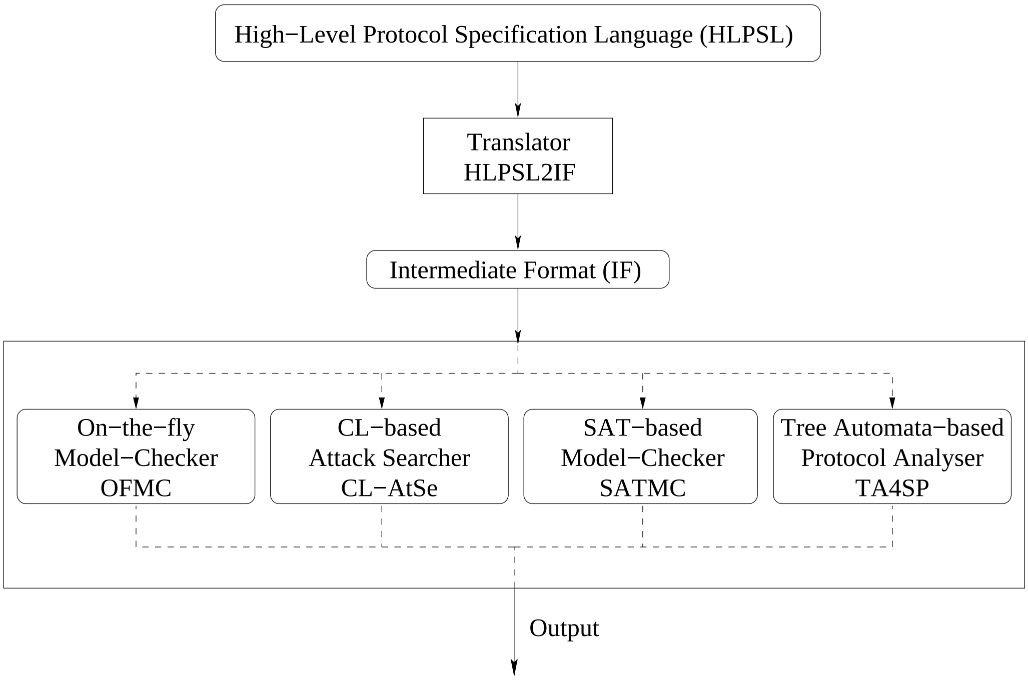

Figure 1: Architecture of the AVISPA Tool

The structure of the AVISPA Tool [2] is shown in Fig. 1. A HLPSL specification is translated into the *Intermediate Format (IF)*, using a translator called `hlpsl2if`. IF is a lower-level language than HLPSL and is read directly by the back-ends to the AVISPA Tool. Note that this intermediate translation step is transparent to the user, as the translator is called automatically. The interested reader can find more about the IF in the AVISPA User Manual and in the AVISPA deliverable which discusses IF [4, 5]. The IF specification of a protocol is then input to the back-ends of the AVISPA Tool to analyse if the security goals are satisfied or violated.

At the time of this writing, the AVISPA Tool comprises four back-ends: OFMC [6], CL-AtSe [16], SATMC [1], and TA4SP[7]; this list may later be extended with new back-ends. The Intermediate Format IF is designed with the objective that it should be simple for developers of other analysis tools to use IF as their input language. Because the analysis methods of the four current back-ends are complementary (at least partially, in the sense that some basic techniques are common to some of the back-ends) and not equivalent, situations might arise in which the back-ends return different results.

Downloading and Running The AVISPA Tool The AVISPA Tool, as well as a very helpful XEmacs mode for editing HLPSL specifications with syntax highlighting etc., and tools for documenting HLPSL specifications in LaTeXand HTML format, is available for download at `http://www.avispa-project.org/download.html`. See the INSTALL and README files contained in the package for further information.

There is also a web interface available at `http://www.avispa-project.org/web-interface/` which allows users to experiment with HLPSL and the AVISPA Tool without having to install anything. Through the web interface, you can select one of the protocols of the AVISPA library, modify it if you like, or write a protocol on your own; you can use one of the four back-ends to check the given protocol, or even use all of them and then compare their outputs.

If a security goal of the specification is violated, the back-ends provide a trace which shows the sequence of events leading up to the violation and displays which goal was violated. The command-line AVISPA Tool outputs attack traces in a textual form we will see later. The web interface can also display an attack trace in the form of a Message Sequence Chart.

The AVISPA tool is called simply `avispa`. The `-h` flag returns usage information as follows:

```
% avispa -h
```

Given an HLPSL file called, for instance, `example.hlpsl`, we can invoke the AVISPA tool with its default options as shown here:

```
% avispa example.hlpsl
```

By default, the AVISPA Tool invokes the OFMC back-end, also called a *sub-module* of the tool. An alternative sub-module can be specified on the command-line in order to invoke a different back-end. At the time of writing, the four valid sub-module arguments, corresponding to the four back-ends listed above, are `--ofmc`, `--satmc`, `--cl-atse` and `--ta4sp`. For instance, we can analyse the HLPSL file using SATMC as follows:

```
% avispa example.hlpsl --satmc
```

In this tutorial, we will focus on invoking the AVISPA Tool with the default options. The usage information, which is printed when invoking `avispa -h`, gives a more complete description of the options not discussed here.

We now discuss the HLPSL language itself.

1.2 Basic Roles

It is easiest to translate a protocol into HLPSL if it is first written in Alice-Bob (A-B) notation. For example, below we illustrate A-B notation with the well-known Wide Mouth Frog protocol [8]:

```
A -> S : {Kab}_Kas
S -> B : {Kab}_Kbs
```

This simple protocol illustrates A-B notation as well as some of the naming conventions we adopt throughout this document (and in general). In this protocol, `A` wishes to set up a secure session with `B` by exchanging a new shared session key with the help of a trusted server `S` with which `A` and `B` each share a key. We denote with `Kas` (respectively `Kbs`) the key shared between `A` (respectively `B`) and `S`. `A` starts by generating a new session key, `Kab`, which is intended for `B`. She encrypts this key with `Kas` and sends it to `S` in the first message (note that encryption is denoted using curly brackets and the encryption key is identified following an `_`). `S`, in turn, decrypts the message, re-encrypts `Kab` with `Kbs`, and sends the result on to `B`. After this exchange, `A` and `B` share the new session key and can use it to communicate with one another.

A-B notation is convenient, as it gives us a clear illustration of the messages exchanged in a normal run of a given protocol. Several protocol specification languages, including an older version of HLPSL, are based on A-B notation. In practise, however, A-B notation is not expressive enough to capture the sequence of events that need to be specified when considering large-scale Internet protocols. For instance, such protocols often call for control-flow constructs such as if-then-else branches, looping and other features. A-B notation, which shows only message exchanges, is too high level to capture such constructs, which talk about the execution of actions by a single participant of a protocol run. That's why we need a more expressive language like HLPSL.

HLPSL is a role-based language, meaning that we specify the actions of each kind of participant in a module, which is called a *basic role*. Later we instantiate these roles and we specify how the resulting participants interact with one another by "gluing" multiple basic roles together into a *composed role*. When modelling a protocol, it can be helpful to begin with an illustration of the flow of messages in A-B notation, and then proceed with the specification of the basic roles. For each (type of) participant in a protocol, there will be one basic role specifying his sequence of actions. This specification can later be instantiated by one or more agents playing the given role. In the case of the WMF protocol, for instance, there are three basic roles, which we call `alice`, `bob`, and `server`. We note that role names always begin with lower-case letters. We use, for instance, the name `alice` to denote the role itself, while the name of the agent playing the role will be called `A`, as in the A-B notation above.

Each basic role describes what information the participant can use initially (*parameters*), its *initial state*, and ways in which the state can change (*transitions*). For instance, the role of `alice` in the protocol above might look like this:

```
role alice(A,B,S : agent,
           Kas : symmetric_key,
           SND, RCV : channel (dy))
played_by A def=
  local State: nat, Kab: symmetric_key
  init  State := 0
  transition
    ...
end role
```

This is (a fragment of) a role known as alice, with parameters A, B and S of type agent, and Kas of type symmetric_key. The RCV and SND parameters are of type channel, indicating that these are channels upon which the agent playing role alice will communicate. The attribute to the channel type, in this case (dy), denotes the *intruder model* to be considered for this channel. Intruder models are discussed further below.

All variables in HLPSL begin with a capital letter, and all constants begin with a lower-case letter. Moreover, all variables and constants are typed. For the moment, assume that these parameter values are passed to role alice from elsewhere. The parameter A appears in the played_by section, which means, intuitively, that A denotes the name of the agent who plays role alice. Also note the local section which declares local variables of alice: in this case, one called State which is a nat (a natural number) and another called Kab, which will represent the new session key. The local State variable is initialised to 0 in the init section.

For information about the different types available in HLPSL and other details, please see the AVISPA User Manual [5].

1.3 Transitions

The transition section of a HLPSL specification contains a set of transitions. Generally, each one represents the receipt of a message and the sending of a reply message. A transition consists of a trigger, or precondition, and an action to be performed when the trigger event occurs. An example belonging to role server of our running example is shown here:

```
step1. State  = 0 /\ RCV({Kab'}_Kas) =|>
       State':= 2 /\ SND({Kab'}_Kbs)
```

This is a transition called step1, though the names of the transitions serve merely to distinguish them from one another. It specifies that if the value of State is equal to zero and a message is received on channel RCV which contains some value Kab' encrypted with Kas, then a transition fires which sets the new value of State to 2 and sends the same value Kab' on channel SND, but this time encrypted with Kbs.

Here we see an example of *priming*: `X'` means *the new value of the variable X*. We say *"X prime"* (the notation stems from the temporal logic TLA [12, 13], upon which HLPSL is based). It is important to realise that the value of the variable will not be changed until the current transition is complete. So, the right-hand-side tells us that the value of the `State` variable, after transition `step1` fires, will be `2`.

A more interesting example, however, is the primed variable that is within the `RCV`. In this case, we bind the variable to whatever is received. As in the example, we can specify a structure of the message that is expected: in this case, we expect an encrypted message. The message must be encrypted with key `Kas`: the fact that this variable is not primed indicates that the received message must have the same value as the current value of the variable. The *contents* of the encrypted message, however, can be arbitrary. Whatever is in there, it will be bound to the variable `Kab` in the next step, because it is primed.

This is how one may model the way in which the information available to a role may change.

1.4 Composed Roles

Composed roles instantiate one or more basic roles, "gluing" them together so they execute together, usually in parallel (with interleaving semantics). Once you have defined your basic roles, you need to define composed roles which describe sessions of the protocol. If we assume, in addition to the `alice` role we've already discussed, that we also have a `bob` and a `server` role with the arguments one would expect, then we can define a composed role which instantiates one instance of each basic role and thus describes one whole protocol session. By convention, we generally call such a composed role `session`.

```
role session(A,B,S    : agent,
             Kas, Kbs : symmetric_key) def=

local SA, RA, SB, RB SS, RS: channel (dy)

composition
    alice (A, B, S, Kas, SA, RA)
/\  bob   (B, A, S, Kbs, SB, RB)
/\  server(S, A, B, Kas, Kbs, SS, RS)

end role
```

Composed roles have no `transition` section, but rather a `composition` section in which the basic roles are instantiated. The `/\` operator indicates that these roles should execute in parallel.

In the `session` role, one usually declares all the channels used by the basic roles. These variables are not instantiated with concrete constants. The `channel` type takes an additional attribute, in parentheses, which specifies the intruder model one assumes for that channel. Here, the type declaration `channel (dy)` stands for the Dolev-Yao intruder model [11]. Under this model, the intruder has full control over the network, such that all messages sent by agents will go to the intruder. He may intercept, analyse, and/or modify messages (as far as he knows the required keys), and send any message he composes to whoever he pleases, posing as any other agent. As a consequence, the agents can send and receive on whichever channel they want; the intended connection between certain channel variables (e.g. `alice` sends on `SA` some messages to `bob` who receives them on `RB`) is irrelevant because the intruder *is* the network.

Finally, a top-level role is always defined. This role contains global constants and a composition of one or more sessions, where the intruder may play some roles as a legitimate user. There is also a statement which describes what knowledge the intruder initially has. Typically, this includes the names of all agents, all public keys, his own private key, any keys he shares with others, and all publicly known functions. Note that the constant `i` is used to refer to the intruder. For example:

```
role environment()
def=

  const a, b, s       : agent,
        kas, kbs, kis : symmetric_key

  intruder_knowledge = {a, b, s, kis}

  composition

       session(a,b,s,kas,kbs)
    /\ session(a,i,s,kas,kis)
    /\ session(i,b,s,kis,kbs)

end role
```

The final statement in a specification is always an instantiation of the top level role:

```
environment()
```

This section has given a basic understanding of the structure of HLPSL specifications. Readers new to HLPSL are recommended to continue reading to the next section of this tutorial. More detailed information on HLPSL specifications can be found in the AVISPA Package User Manual [5], which describes the full syntax and semantics of HLPSL.

2 HLPSL Examples

2.1 Example 1 - from Alice-Bob notation to HLPSL specification

Suppose A and B share a secret key K (i.e. K is a value known only to A and B). Consider the following protocol for producing a new shared key K1:

```
A -> B: {Na}_K
B -> A: {Nb}_K
A -> B: {Nb}_K1, where K1=Hash(Na.Nb)
```

In Alice-Bob notation, this reads: A sends to B a nonce Na, encrypted with K. B then sends to A another nonce Nb also encrypted with K. Finally A calculates a new key K1 by hashing the value of Na and Nb concatenated together, and sends back to B the value of Nb encrypted with K1.

Goals of the Example Protocol We will discuss the concrete modelling of security goals in a coming example, but we summarise the intended security goals of this example protocol here. This is a (toy) key exchange protocol in which the first two messages serve to establish key agreement, and the last one serves as proof that A has the new key. Our first goal for this example is *unilateral authentication*: that is, that B authenticates A on Nb (on the last message), in other words: when B receives the third message, he can be sure that Nb was sent by A. Furthermore, we require *strong authentication*, an extension of so-called *weak authentication* which precludes replay attacks. We can thus also conclude that, if strong authentication is achieved, then Nb has not been previously received by B. As a second security goal, the new key K1 should be kept secret.

Below is a fragment of role alice modelled in HLPSL. Note that comments in a HLPSL specification begin with the % symbol and continue to the end of the line.

```
role alice(..,
           K: symmetric_key,  % K and Hash must be passed to each
           Hash: hash_func,   % role, so that A and B agree on them.
           ..)
... def=

  local
    ...
    State : nat         % This variable is typically defined in all roles.

  init
    State := 1
```

```
transition

1. State  = 1 /\ RCV(start) =|>
   State':= 2 /\ Na' := new() /\ SND({Na'}_K)

2. State  = 2 /\ RCV({Nb'}_K) =|>
   State':= 3 /\ SND({Nb'}_Hash(Na.Nb'))
```

Discussion: Modelling Shared Knowledge Recall that A and B are assumed to agree beforehand on the value of K. They must also agree on which cryptographic hash function they will use in order to generate K1. We model such pre-shared knowledge by passing the same values to those instances of roles alice and bob who should participate in a session together.[1] These values are passed from the calling composition role, as can be seen in the full example below: the first session between a and b uses key kab for the value of K, while the second between a and the intruder i uses kai. (Recall that lower case identifiers denote constants.) All three sessions use the same hash function h. Although the variable names in the sharing roles need not necessarily be the same, by convention one generally gives them the same names.

Discussion: Transitions The first transition is relatively clear, but it illustrates an important feature of HLPSL: namely, how one models the generation of fresh data. start is a signal for alice to begin the protocol run. She creates a fresh value for nonce Na by assigning it to new(), which intuitively means that value is generated randomly.[2] We note also that new() can be applied to data of arbitrary types, for instance to generate fresh values of type symmetric_key. She encrypts this value using key K before inserting the encrypted value into the channel called SND. After this transition, alice is in state 2.

The second transition is trickier. Firstly, alice receives a message {Nb'}_K. Provided that alice is in state 2 and that this message is of the form {*}_K, for some value *, alice sets Nb to be the received value encrypted under K. In the same transition, the newly received value is stored as the new value of Nb and sent out again, encrypted with the key Hash(Na.Nb') which is computed by hashing the concatenation of the two values Na and Nb'.

The full solution for this example is provided below. Note that it contains a number of aspects yet to be explained. For example, this specification contains the terms secret, witness and request (all of which are related to describing security goals). As these concepts will be covered later in the tutorial, it is safe to ignore them for now.

[1] Note that the value of K known by two instances of alice who participate in different sessions can, of course, be different.

[2] More precisely, each freshly generated value is unique and has never been generated before.

Example 1:

```
role alice(
  A,B     : agent,
  K       : symmetric_key,
  Hash    : hash_func,
  SND,RCV : channel(dy))
played_by A def=

  local
    State   : nat,
    Na,Nb   : text,
    K1      : message

  init
    State := 0

  transition

  1. State  = 0 /\ RCV(start) =|>
     State’:= 2 /\ Na’ := new()
                /\ SND({Na’}_K)

  2. State  = 2 /\ RCV({Nb’}_K) =|>
     State’:= 4 /\ K1’ := Hash(Na.Nb’)
                /\ SND({Nb’}_K1’)
                /\ witness(A,B,bob_alice_nb,Nb’)

end role
```

```
role bob(
    A,B     : agent,
    K       : symmetric_key,
    Hash    : hash_func,
    SND,RCV : channel(dy))
played_by B def=

  local
```

```
    State   : nat,
    Nb,Na   : text,
    K1      : message

  init
    State := 1

  transition

  1. State  = 1 /\ RCV({Na'}_K) =|>
     State':= 3 /\ Nb' := new()
                /\ SND({Nb'}_K)
                /\ K1':= Hash(Na'.Nb')
                /\ secret(K1',k1,{A,B})

  2. State  = 3 /\ RCV({Nb}_K1) =|>
     State':= 5 /\ request(B,A,bob_alice_nb,Nb)

end role
```

```
role session(
    A,B  : agent,
    K    : symmetric_key,
    Hash : hash_func)
def=

  local SA, SB, RA, RB : channel (dy)

  composition

     alice(A,B,K,Hash,SA,RA)
  /\ bob  (A,B,K,Hash,SB,RB)

end role
```

```
role environment()
def=
  const
    bob_alice_nb,
    k1          : protocol_id,
    kab,kai,kib : symmetric_key,
    a,b         : agent,
    h           : hash_func

  intruder_knowledge = {a,b,h,kai,kib}

  composition
     session(a,b,kab,h)
  /\ session(a,i,kai,h)
  /\ session(i,b,kib,h)

end role
```

```
  goal
    secrecy_of k1
    authentication_on bob_alice_nb
  end goal
```

```
  environment()
```

Running the AVISPA Tool on this example returns the following output:

```
% avispa ex1.hlpsl

% OFMC
% Version of 2005/06/07
SUMMARY
  SAFE
DETAILS
  BOUNDED_NUMBER_OF_SESSIONS
```

```
PROTOCOL
  ./ex1.if
GOAL
  as_specified
BACKEND
  OFMC
COMMENTS
STATISTICS
  parseTime: 0.00s
  searchTime: 0.16s
  visitedNodes: 105 nodes
  depth: 8 plies
```

As we remarked above, the tool calls the OFMC back-end when called with the default options. We can see that OFMC found no attacks. In other words, the stated security goals were satisfied for a bounded number of sessions as specified in `environment` role. The AVISPA Tool supports alternatives to a bounded scenario (that is, a bounded number of concrete sessions), but these go beyond the scope of this tutorial. We refer the interested reader to the AVISPA User Manual [5].

2.2 Example 2 - common errors, untrusted agents, attack traces

This example considers a Kerberos-style protocol with 3 principals: `A`, `B` and `S`. `A` wishes to establish a secret key `K` with `B`, but both have only secret keys with `S`. `A` asks `S` for such a key, giving his identity and the identity of `B`. Figure 2 provides a graphical representation of this protocol.

Alice-Bob Notation:

```
A -> S: (A.B.{Na}_Ka)        % Ka is a key shared by A and S
A <- S: (A.B.{K.Na.Ns}_Ka.   % S generates new key K
             {K.Na.Ns}_Kb)   % A cannot decrypt the contents of {K.Na.Ns}_Kb
                             % but he is able to forward that to B
A -> B: (A.B.{K.Na.Ns}_Kb.
             {Na.Ns}_K)      % The last part is
                             % a key confirmation: B knows K
A <- B: (A.B.{Ns.Na}_K)
```

We give a full HLPSL specification of this protocol below and then discuss the protocol model and a number pitfalls that HLPSL modellers should be aware of.

```
role alice (A, S, B: agent,
```

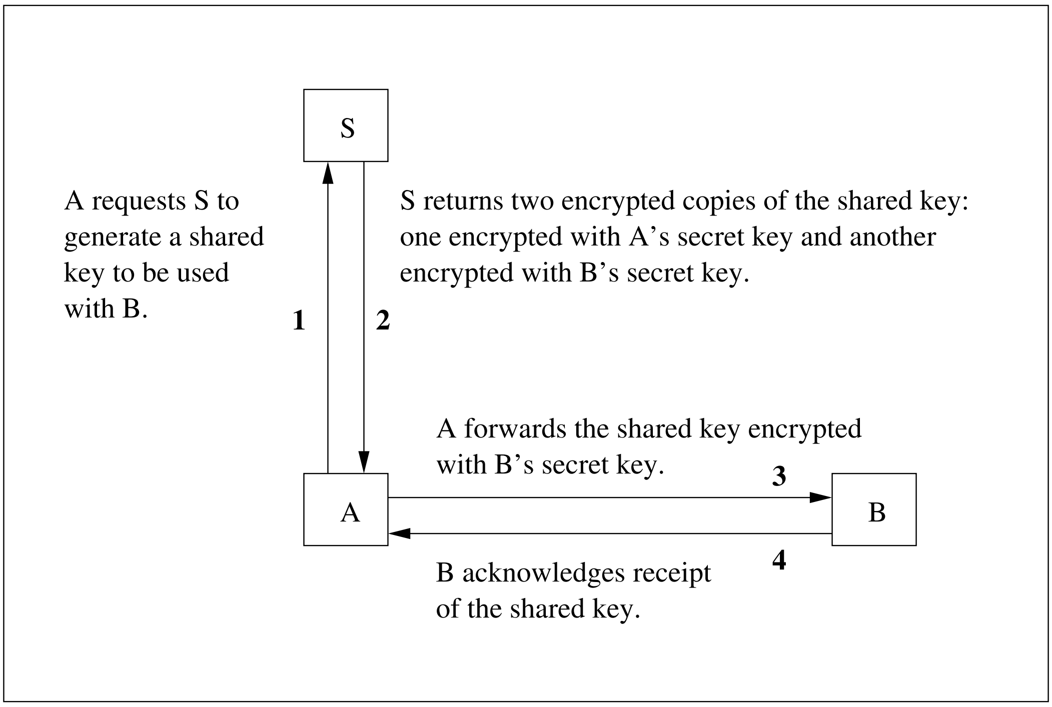
S
A requests S to generate a shared key to be used with B.
1
2
S returns two encrypted copies of the shared key: one encrypted with A's secret key and another encrypted with B's secret key.
A forwards the shared key encrypted with B's secret key.
3
A
B
4
B acknowledges receipt of the shared key.

Figure 2: Representation of the Kerberos-style protocol of Example 2

```
            Ka     : symmetric_key,
            SND_SA, RCV_SA, SND_BA, RCV_BA: channel(dy))
played_by A
def=

  local State : nat,
        Na,Ns : text,
        K     : symmetric_key,
        X     : {symmetric_key.text.text}_symmetric_key

  init   State := 0

  transition

  1. State  = 0 /\ RCV_BA(start) =|>
     State':= 2 /\ Na' := new()
                /\ SND_SA(A.B.{Na'}_Ka)

  2. State  = 2 /\ RCV_SA(A.B.{K'.Na.Ns'}_Ka.X') =|>
```

```
      State':= 4 /\ SND_BA(A.B.X'.{Na.Ns'}_K')

   3. State  = 4 /\ RCV_BA(A.B.{Ns.Na}_K) =|>
      State':= 6 /\ request(A,B,alice_bob_na,Na)

 end role
```

```
 role server (A, S, B  : agent,
              Ka, Kb   : symmetric_key,
              SND_AS, RCV_AS: channel(dy))
 played_by S
 def=

   local State : nat,
         Ns,Na : text,
         K     : symmetric_key

   init  State := 1

   transition

   1. State  = 1 /\ RCV_AS(A.B.{Na'}_Ka) =|>
      State':= 3 /\ Ns' := new() /\ K' := new()
                 /\ SND_AS(A.B.{K'.Na'.Ns'}_Ka.{K'.Na'.Ns'}_Kb)
                 /\ secret(K',k,{A,B,S})

 end role
```

```
 role bob (A, S, B: agent,
           Kb     : symmetric_key,
           SND_AB, RCV_AB: channel(dy))
 played_by B
 def=

   local State     : nat,
         Ns, Na    : text,
         K         : symmetric_key
```

```
  init  State := 5

  transition

  1. State  = 5 /\ RCV_AB(A.B.{K'.Na'.Ns'}_Kb.{Na'.Ns'}_K') =|>
     State':= 7 /\ SND_AB(A.B.{Ns'.Na'}_K')
                /\ witness(B,A,alice_bob_na,Na')

end role
```

```
role session(A, S, B  : agent,
             Ka, Kb   : symmetric_key)
def=

  local
    SSA, RSA,
    SBA, RBA,
    SAS, RAS,
    SAB, RAB  : channel (dy)

  composition

      alice (A, S, B, Ka, SSA, RSA, SBA, RBA)
  /\  server(A, S, B, Ka, Kb, SAS, RAS)
  /\  bob   (A, S, B, Kb, SAB, RAB)

end role
```

```
role environment()
def=

  const a, b, s        : agent,
        ka, kb, ki     : symmetric_key,
        alice_bob_na, k: protocol_id

  intruder_knowledge = {a, b, s, ki}
  composition

     session(a,s,b,ka,kb)
  /\ session(a,s,i,ka,ki)
  /\ session(i,s,b,ki,kb)

end role
```

```
goal
  secrecy_of k
  authentication_on alice_bob_na
end goal
```

```
environment()
```

2.2.1 Modelling Tips and Pitfalls

Priming Placing primes correctly can be difficult but is very important for the correctness of protocol models. Consider, for instance, the first transition of role bob. One could potentially make the simple omission of forgetting one or more of the primes on variable Na, yielding the following transition:

```
1. State  = 5 /\ RCV_AB(A.B.{K'.Na.Ns'}_Kb.{Na.Ns'}_K') =|>
   State':= 7 /\ SND_AB(A.B.{Ns'.Na}_K')
              /\ witness(B,A,alice_bob_na,Na)
```

This simple error renders the transition non-executable. The reason is that an unprimed variable within a receive action serves to restrict the messages that will be accepted: in this case, for example, the role bob expects to be talking to a certain agent A, and should accept only messages where this value of A appears in the first part of the message he receives. Therefore, the variable A is written without a prime, indicating that the current value of A, which in this case is a parameter of role, is used as a pattern. Yet failing to give the prime for variable Na has the effect of erroneously restricting the accepted messages to those having at the given position the current value of Na. Even worse, this value is undefined since Na has not been initialized! Since no messages satisfy this requirement, the transition can never fire. Correcting this using a prime, i.e. Na', we get the desired effect that Na is initialized by the (unrestricted) value received at the given position in the message.

Following these simple rules for priming will hopefully help modellers avoid most problems related to the placement of primes.

1. Variables on the left hand side of a transition are often arguments of a receive action. In this situation, primed variables (e.g. X') will be assigned the value received in the message. Unprimed variables (e.g. X) will restrict the messages which are accepted.

   For example:

   ```
   RCV(A.X')
   ```

   This will appear on the left hand side of a transition, and will only enable the transition if the message received is a pair whose first component matches the *current* value of the variable A. If the transition is fired, then after the transition has completed, the value of X will be equal to whatever value was sent in the second component of the message.

2. On the right hand side of a transition, use a primed variable name and the assignment operator := when assigning a new value to a variable. For example:

   ```
   State' := 3
   ```

   This also holds when generating and using nonces, e.g.:

   ```
   Na' := new() /\ SND(Na')
   ```

3. A primed variable X' always means *the new value of* X, and it helps to read things this way.

There is one other important thing to consider in relation to priming. All state changes specified in a transition occur simultaneously. So, assume you have a transition like this in which an agent receives some value and simply forwards it on:

```
RCV(X') =|> SND(X)
```

Here, both variables must be primed. Otherwise you would be sending the current (old) value of X rather than forwarding the newly received value.

Variable Sharing Here, we refer not to the issue of modelling shared *initial knowledge*, as discussed in the previous example. Rather, we mean the sharing of *variables*. For instance, in this example, each of the three roles has local variable called `K`. However, it is inappropriate for them to share the variable because it is not some a priori knowledge and must be negotiated as part of the protocol. Therefore, each role should have its own copy of the variable to allow us to see situations where this information might not correspond properly (a situation which might represent an attack).

In fact, in HLPSL, *variables* (which have the ability to change) cannot be shared.[3] Yet, it is appropriate and possible to share (constant) *values* when you require roles to have pre-shared knowledge — for instance, a shared key. As we have seen in the previous example, this is achieved by passing the value to be shared as an argument to several roles.

Commas vs. Periods It is important to note the difference between using commas "`,`" and using periods "`.`" in HLPSL specifications.

When denoting composite messages, one should always use periods, as they indicate concatenation of (sub-)messages, e.g. when sending or receiving from a channel:

```
SND(A.B.Na')
```

Similarly, when concatenating messages to be given as an argument in a function call, periods should be used to form a single message, as in one of the previous examples where the established key was computed by taking the hash of two nonces: `Hash(Na.Nb')`.

When separating *different arguments* of predicates, roles, or goal events (discussed in the following example), commas should be used, e.g.:

```
server(A, S, B, Ka, Kb, SAS, RAS)
```

Compound Types and Modelling Forwarding In Kerberos-style protocols, situations often arise in which one agent receives a message in which there are parts she cannot actually decrypt. Rather, the agent should forward these parts on to someone else. The protocol modeller, of course, knows what the format of this message should be and can specify the forwarding agent's transition as if she were simply receiving the message. For instance, one could write `alice`'s second transition from the specification above alternatively as follows:

```
step2. State = 2 /\ RCV(A.B.{K'.Na.Ns'}_Ka.{K'.Na.Ns'}_Kb) =|>
       SND(A.B.{K'.Na.Ns'}_Kb.{Na.Ns'}_K') /\ State' := 3
```

[3] Sets are an exception to this rule because they are passed by reference, but the discussion of sets is out of scope for this tutorial. We note, for the interested reader, that variable sharing is not excluded by the HLPSL semantics but rather by our conditions for faithful translation to IF. That is, it is a feature not yet supported. For more information on variable sharing, please see the AVISPA User Manual [5].

We prefer to write such forwarding transition differently, in a way which is intended to faithfully model the aspect of the protocol that alice cannot actually decrypt the contents of the latter part of the message, as it is encrypted with Kb. Instead, we would like to write that alice receives (A.B.{K'.Na.Ns'}_Ka.X') where X' is simply some data that alice cannot understand. She expects it to be of the form {K'.Na.Ns'}_Kb, for some Kb, but even though she knows the plain text (K'.Na.Ns'), she cannot match X' to {K'.Na.Ns'}_Kb because she does not know the key Kb.

Therefore, we rather write the transition as follows:

```
step2. State = 2 /\ RCV(A.B.{K'.Na'.Ns'}_Ka.X') =|>
       SND(A.B.X'.{Na.Ns'}_K') /\ State' := 3
```

Where X is a local variable used instead of {K'.Na.Ns'}_Kb, X should be declared of the *compound type* {symmetric_key.text.text}_symmetric_key, because the value to be received for it has the following form: a symmetric key and two bit-strings, which are jointly encrypted by a symmetric key. One could also give X the most general type message, but we prefer to default to the most specific type possible when modelling, as this facilitates the analysis of the protocols when running the AVISPA Tool with the argument --typed_model=strongly. One can still perform an untyped analysis by specifying the --typed_model=no argument to those back-ends that support it, or disable compound typing by specifying --typed_model=yes.

Assigning State Numbers In Example 2, we can see that the state numbers are even for Alice, and odd for Bob and the Server. We often assign state numbers in this way, which reflects the intended order of send and receive events. This is not compulsory, however, rather it is a convenient modelling convention which keeps things clear while reading the HLPSL and the traces printed by the back-ends.

Executability For non-trivial HLPSL specifications, it is often the case that, due to some modelling mistake(s), the protocol model cannot execute to completion. This is problematic, because the back-ends might not find an attack simply because the protocol model cannot reach the state where the attack can happen. Therefore, an executability check is very important. See subsection 3.6 for how to do this. Typical modelling mistakes leading to blocked transitions are mismatches of expected and actually sent values, or, even harder to spot, mismatches of their types. See subsection 3.9 for a particularly tricky case.

Another potential source of non-executability is insufficient knowledge of the intruder. In particular, when he plays the role of an honest agent. Therefore, make sure to include all relevant values in the intruder_knowledge = {...} declaration of the environment role.

2.2.2 Discussion and Analysis Results The parameters of a role define what information it begins with, and are passed in as arguments from composed roles. For instance, the `session` role is used to describe a single execution of the protocol. The `session` role composes three roles together and defines for each role, what information it begins with by passing this in as arguments.

The `environment` role is the top-level role, and describes three concurrent sessions. The first is a typical session with the legitimate agents `a`, `b` and `s`. Note that all of the arguments are in lower-case within the environment role. This is because they are constants rather than variables.

The second and third sessions are ones in which the intruder is impersonating either Alice or Bob. You can see from the arguments to these sessions that the intruder (`i`) is playing the role of a legitimate user in order to attack the protocol. He even has a shared key with the server (`ki`) with which he can communicate in a regular manner.

When this model of the protocol is analysed using the AVISPA Tool, the following output results (the output shown here has been formatted to fit the page):

```
% avispa ex22.hlpsl

% OFMC
% Version of 2005/06/07
SUMMARY
  UNSAFE
DETAILS
  ATTACK_FOUND
PROTOCOL
  ./ex22.if
GOAL
  authentication_on_alice_bob_na
BACKEND
  OFMC
COMMENTS
STATISTICS
  parseTime: 0.00s
  searchTime: 0.16s
  visitedNodes: 21 nodes
  depth: 3 plies
ATTACK TRACE
i -> (a,3): start
(a,3) -> i: a.b.{Na(1)}_ka
i -> (s,7): a.i.{Na(1)}_ka
(s,7) -> i: a.i.{K(2).Na(1).Ns(2)}_ka.{K(2).Na(1).Ns(2)}_ki
```

```
i -> (a,3): a.b.{K(2).Na(1).Ns(2)}_ka.x61
(a,3) -> i: a.b.x61.{Na(1).Ns(2)}_K(2)
i -> (a,3): a.b.{Ns(2).Na(1)}_K(2)


% Reached State:
% state_dummy(i,authentication_on_alice_bob_na,i,0,17)
% request(a,b,alice_bob_na,Na(1),3)
% secret(K(2),k,set_99)
% contains(a,set_99)
% contains(i,set_99)
% contains(s,set_99)
% state_bob(b,i,s,kb,5,dummy_nonce,dummy_nonce,dummy_sk,12)
% state_server(s,i,b,ki,kb,1,dummy_nonce,dummy_nonce,dummy_sk,set_101,12)
% state_server(s,a,i,ka,ki,3,Ns(2),Na(1),K(2),set_99,7)
% state_alice(a,s,i,ka,0,dummy_nonce,dummy_nonce,dummy_sk,
                         {dummy_sk.dummy_nonce.dummy_nonce}_dummy_sk,7)
% state_alice(a,s,b,ka,6,Na(1),Ns(2),K(2),x61,3)
% state_bob(b,a,s,kb,5,dummy_nonce,dummy_nonce,dummy_sk,3)
% state_server(s,a,b,ka,kb,1,dummy_nonce,dummy_nonce,dummy_sk,set_92,3)
```

The tool output shows that the protocol has been found to be unsafe and that an attack has been found. Most of the output can safely be ignored, but the most interesting output is the attack trace itself. Shown under the heading `ATTACK TRACE`, it shows the exchange of messages leading to an attack: that is, is a violation of the given goal, "`authentication_on alice_bob_na`". We now examine the attack trace in more detail.

The intruder initiates the first session with agent `a` by sending the special `start` message. For reasons related to the internal workings of the `hlpsl2if` translator and the AVISPA back-ends, each role instance is assigned a session number: in this case, "3".

```
i -> (a,3): start
```

`a` replies to the intruder. Note that, because we assume that the intruder is the network, all messages pass through the intruder, even though he may not be the intended recipient. Here, we can see from the second component of the message that `a` actually wishes to talk with agent `b`. We note also that terms which are generated freshly are denoted with their variable names and a number which is used to identify them uniquely. In this case, it is the first time any agent has generated a nonce to be called `Na`, so the value is denoted by `Na(1)`.

```
(a,3) -> i: a.b.{Na(1)}_ka
```

The intruder then forwards the message to s (more precisely, the instance of s which has been assigned session number 7). However, the intruder has inserted his own name into the second component of the message, telling s that a wishes to talk to i rather than to b. Note that the intruder has not broken the encryption, but has simply copied the encrypted data into the new message.

```
i -> (s,7): a.i.{Na(1)}_ka
```

s replies to the intruder with an appropriate response, including nonces encrypted with a key ki shared between s and the intruder. The intruder now has knowledge of these nonces, as well as K(2), a session key which a will believe he can use to talk to b.

```
(s,7) -> i: a.i.{K(2).Na(1).Ns(2)}_ka.{K(2).Na(1).Ns(2)}_ki
```

The intruder sneakily switches the name of the correspondent back to b so that a is none the wiser, and forwards the message part he received from s on to a. The last part of the message, x61, is a variable related to the internal workings of the OFMC back-end, in particular to a technique known as the *lazy intruder* [6]. Intuitively, the presence of a variable means that it doesn't matter what the intruder sends for that message part, because the receiving agent will not actually check what's there.

```
i -> (a,3): a.b.{K(2).Na(1).Ns(2)}_ka.x61
```

Now, a sends the nonces encrypted with the intruder's key to b, but they are again intercepted by the intruder. Note that a still believes that the key she used for encryption, K(2), is actually shared with b. b, however, hasn't even participated yet.

```
(a,3) -> i: a.b.x61.{Na(1).Ns(2)}_K(2)
```

Now the intruder can easily decrypt these values and send them back to a encrypted with the new session key K(2). This brings a to the end of the protocol run and she issues her "request" fact (discussed in the next example), which asserts that she believes she is talking to b, while in fact she is talking with the intruder!

```
i -> (a,3): a.b.{Ns(2).Na(1)}_K(2)
```

Consequently, it is clear that the stated security goal "authentication_on alice_bob_na" was indeed violated.

We now turn to a third example, where we will discuss how to specify the security goals of a protocol.

2.3 Example 3 - security goals

This example considers the Andrew Secure RPC Protocol. Below is the A-B notation for this protocol adapted from page 45 of [10].

```
A -> B : A.{Na}_Kab
B -> A : {Na+1.Nb}_Kab
A -> B : {Nb+1}_Kab
B -> A : {K1ab.N1b}_Kab
```

The protocol is used to authenticate both parties to each other and then establish a fresh shared key K1ab which can be used for further communication. The value N1b is sent for use in the future. Both K1ab and N1b are generated by B.

Operators and Functions This protocol presents more of a modelling challenge than our previous examples. For instance, the inclusion of an operator like + is something we have not seen before. HLPSL does not support arbitrary arithmetic operators; however, we can model an approximation of addition which will reflect the properties that, from a security perspective, are most important. In particular, the difference between the use of addition and, for instance, a cryptographic hash, is that the intruder could easily invert addition by subtracting 1, while we assume he cannot invert a cryptographic hash. For an operator like + it is clear that the inverse can be computed with negligible computational complexity. In this case, however, the results of additions are never sent unencrypted, and it is a reasonable assumption that the intruder cannot compute {Na+1}_K given {Na}_K without either knowing K or breaking cryptography. We therefore make the modelling assumption that the agents should simply compute *some* function of Na, not necessarily an easily invertible one. In this case, we can use a function symbol in HLPSL — that is, a value of type hash_func. For this example, we will define a successor function called Succ. This function will be known to the intruder, so he will be able to calculate successors of values he knows but not invert values of the form Succ(Na) (unless he already knows Na).

The HLPSL specification of this protocol is given below.

```
role alice (A, B     : agent,
            Kab      : symmetric_key,
            Succ     : hash_func,
            SND, RCV : channel(dy))
played_by A
def=

  local State  : nat,
        Na,Nb  : text,
        K1ab   : symmetric_key,
        N1b    : text

  const alice_bob_k1ab, alice_bob_na, bob_alice_nb: protocol_id

  init State := 0

  transition

  0. State  = 0 /\ RCV(start) =|>
     State':= 2 /\ Na' := new()
                /\ SND(A.{Na'}_Kab)

  2. State  = 2 /\ RCV({Succ(Na).Nb'}_Kab) =|>
     State':= 4 /\ SND({Succ(Nb')}_Kab)
                /\ witness(A,B,bob_alice_nb,Nb')

  4. State  = 4 /\ RCV({K1ab'.N1b'}_Kab) =|>
     State':= 6
                /\ request(A,B,alice_bob_k1ab,K1ab')
                /\ request(A,B,alice_bob_na,Na)

end role
```

```
role bob (A, B     : agent,
          Kab      : symmetric_key,
          Succ     : hash_func,
          SND, RCV : channel(dy))
played_by B
def=

  local State     : nat,
        Nb,Na,N1b : text,
        K1ab      : symmetric_key

  init State := 1

  transition
  1. State  = 1 /\ RCV(A.{Na'}_Kab) =|>
     State':= 3 /\ Nb' := new()
                /\ SND({Succ(Na').Nb'}_Kab)
                /\ witness(B,A,alice_bob_na,Na')

  3. State  = 3 /\ RCV({Succ(Nb)}_Kab) =|>
     State':= 5 /\ N1b' := new()
                /\ K1ab' := new()
                /\ SND({K1ab'.N1b'}_Kab)
                /\ witness(B,A,alice_bob_k1ab,K1ab')
                /\ request(B,A,bob_alice_nb,Nb)
                /\ secret(K1ab',k1ab,{A,B})
                /\ secret(N1b',n1b,{A,B})

end role
```

```
role session(A, B  : agent,
             Kab   : symmetric_key,
             Succ  : hash_func)
def=

  local SAB, RAB,
        SBA, RBA  : channel (dy)

  composition

      alice(A, B, Kab, Succ, SAB, RAB)
  /\  bob  (A, B, Kab, Succ, SBA, RBA)

end role
```

```
role environment()
def=

  const alice_bob_k1ab, alice_bob_na, bob_alice_nb,
        n1b, k1ab     : protocol_id,
        a, b          : agent,
        kab, kai, kib : symmetric_key,
        succ          : hash_func

  intruder_knowledge = {a, b, kai, kib, succ}

  composition

     session(a,b,kab,succ)
  /\ session(a,b,kab,succ)
  /\ session(a,i,kai,succ)
  /\ session(i,b,kib,succ)

end role
```

```
goal
  secrecy_of n1b, k1ab
  authentication_on bob_alice_nb
  authentication_on alice_bob_na
  authentication_on alice_bob_k1ab
end goal
```

```
environment()
```

2.3.1 Discussion and Analysis Results

Specifying Security Goals Security goals are specified in HLPSL by augmenting the transitions of basic roles with so-called *goal facts* and by then assigning them a meaning by describing, in the HLPSL `goal` section, what conditions — that is, what combination of such facts — indicate an attack. As we will see, a simple example of this is secrecy, where the goal facts assert which values should be secret between whom, and the goal declaration in the `goal` section describes that anytime the intruder learns a secreet value, and it is not explicitly a secret between him and someone else, then it should be considered an attack. Internally, the attack conditions are specified in terms of temporal logic, but useful and concise macros are provided for the two most frequently used security goals, authentication and secrecy. We illustrate with a discussion of the goals of the example protocol above.

The Andrew Secure RPC protocol should ensure both secrecy and mutual authentication. We begin with the former. We would like to ensure that the exchanged key `K1ab` and the generated nonce `N1b` are kept secret among `A` and `B`. We achieve this by first adding goal facts to our specification: in this case, goal facts that assert which values should be kept secret, and which agents are allowed to know such secrets. As a modelling rule of thumb, it is generally advisable to place such `secret` facts in the role that creates the value that should be secret (though for composed values like Diffie-Hellman keys, the creating role may not be unique). In this example, role `bob` creates both values, and so we augment his last transition (labelled `3.`), with the following secret facts (where the primes are required there to refer to the new values of `K1ab` and `N1b`):

```
/\ secret(K1ab',k1ab,{A,B})
/\ secret(N1b' ,n1b, {A,B})
```

These indicate that `B` allows that the two values are shared between (only) `A` and `B`. Note that if `bob` is participating in a session with the intruder (such as the third session specified in the `environment` role above), then we will have `A=i`. In such cases, the intruder is allowed to know the value that has been declared secret. The constant second argument to the secret facts are called *protocol ids* and must be declared of type `protocol_id` in the `const` section of the `environment` role. We adopt the modelling convention of using for the protocol id the name of the variable the secrecy fact refers to, in lower case. In the case of secrecy facts, protocol ids serve merely to distinguish different secrecy goals. This can facilitate analysis later, for instance if one wants to check only the secrecy of `N1b`, ignoring `K1ab` for a particular experiment, one can comment out the corresponding statement `secrecy_of k1ab` in the goal section.

Goal facts as such are merely events in the trace of a protocol model. In addition to the facts themselves, we must make clear how these facts should be interpreted: that is which combinations of goal facts are legal and desirable, and which should be considered attacks? This is done in the HLPSL specification's `goal` section. For secrecy, we need simply write

```
secrecy_of k1ab, n1b
```

as shown in the example. This, as mentioned above, is actually a macro for a temporal logic formula. Intuitively, it states that any values appearing in the first position of secret facts which contain either `k1ab` or `n1b` in the second position are secret. If the intruder learns such a value and he is not in the set given in the third position of the associated secret fact, then this represents an attack.

Authentication The `witness` and `request` events are goal facts related to authentication. They are used to check that a principal is right in believing that its intended peer is present in the current session, has reached a certain state, and agrees on a certain value, which typically is fresh. They always appear in pairs with identical third parameter.

In this example, for instance, the two participants should certainly agree on the value of the exchanged key `K1ab`. In particular, `alice` wishes to be sure that this value was indeed created by `bob`, that it was created for her for the purpose of being used as a shared key, and that it was not replayed from a previous session. To achieve this, we write the line

```
/\ request(A,B,alice_bob_k1ab,K1ab')
```

in the last transition of `alice`. It reads as follows: "agent `A` accepts the value `K1ab'` and now relies on the guarantee that agent `B` exists and agrees with her on this value." Moreover, the third argument `alice_bob_k1ab` is used for distinguishing different authentication pairs: that is, for asserting with what purpose the value is being interpreted. As a modelling convention, it is usually the names of the authenticating role, the role to be authenticated, and the name of the variable being checked, all in lower case, concatenated together. It should be declared as a constant of type `protocol_id` in the top-level role. The interpretation of `request` is thus even stronger, as `A` requires not only that `B` exists and agrees on the value, but also that `B` intended it to be used for the protocol id `alice_bob_k1ab`.

There is also `wrequest` which corresponds to weak authentication. No replay protection is imposed if one uses `wrequest`. Taking the example `request` fact above, were it a `wrequest`, then the requirement that `B` exists would be loosened. It would then suffice that `B` had existed sometime in the past and had, at that time, agreed on value `K1ab'`, having interpreted it as protocol id `alice_bob_k1ab`.

We also include the matching `witness` predicate in role `bob`, as part of the transition in which the value `K1ab` is sent to `alice`.

```
/\ witness(B,A,alice_bob_k1ab,K1ab')
```

The goal fact `witness(B,A,alice_bob_k1ab,K1ab')` should be read "agent `B` asserts that we wants to be the peer of agent `A`, agreeing on the value `K1ab'` in an authentication effort identified by the protocol id `alice_bob_k1ab`."

In the `goal` section of the protocol, we write

```
authentication_on bob_alice_nb
authentication_on alice_bob_na
authentication_on alice_bob_k1ab
```

to indicate that `witness` and `request` goal facts containing those three protocol ids should be taken into account. One can also write `weak_authentication_on` to specify an authentication goal with no replay protection.

Again, the goals above are actually macros for temporal logic formulae. Intuitively, the authentication goal asserts that it is always true that a `request` event has been preceded by an accompanying `witness` event. Accompanying, in this case, means that the two facts agree on the protocol id and the value, and that the two agents names are reversed. Moreover, for strong authentication, no agent should accept the same value twice from the same communication partner: that is, as of one time point before a `request` event, the same value had never been previously requested. This definition corresponds to Lowe's notion of agreement in [14].

In our example we have used `witness` and `request` for three purposes:

- `alice` authenticates `bob` on the value of `Na` (which holds because only `bob` can decrypt `Na` and send back `Succ(Na) to alice`),
- `bob` authenticates `alice` on the value of `Na` (which holds because only `alice` can decrypt `Nb` and send back `Succ(Nb) to bob`),
- `alice` authenticates `bob` on the value of `K1ab`. We abuse strong authentication on `K1ab` here to express that `K1ab` should be generated freshly (and not replayed).

Detecting Replay Attacks One must provide an appropriate analysis scenario within the `environment` role. Often, the structure of the protocol suggests such a scenario. In this case, there is a well known attack on the Andrew Secure RPC Protocol in which an intruder replays the fourth message from a previous protocol run in the place of a legitimate fourth message from `B`. This makes `A` use an old session key, which may have been compromised over time.

In general, replay attacks can be found by specifying multiple parallel sessions between the same agents, as is the case with the first two sessions declared in the `environment` role above. Unfortunately, this can result in a significant slowdown in the analysis.

Therefore, to help find replay attacks, OFMC provides the `-sessco` (*session compilation*) option as follows:

```
% avispa ex3.hlpsl --ofmc -sessco
```

With session compilation, OFMC finds the replay attack even without the second parallel session between `a` and `b`. This is because it first simulates a run of the whole system and in a second run, it lets the intruder take advantage of the knowledge learnt in the first run.[4]

The AVISPA Tool finds the expected replay attack and yields the following output:

```
% avispa ex31.hlpsl
```

[4] We note that the `-sessco` option is also handy for a quick check of executability. In the current version, however, it only works if the `State` variables of each role strictly increase from one transition to the next.

```
% OFMC
% Version of 2005/06/07
SUMMARY
  UNSAFE
DETAILS
  ATTACK_FOUND
PROTOCOL
  ./ex31.if
GOAL
  replay_protection_on_k1ab
BACKEND
  OFMC
COMMENTS
STATISTICS
  parseTime: 0.00s
  searchTime: 6.59s
  visitedNodes: 809 nodes
  depth: 8 plies
ATTACK TRACE
i -> (a,3): start
(a,3) -> i: a.{Na(1)}_kab
i -> (a,6): start
(a,6) -> i: a.{Na(2)}_kab
i -> (b,3): a.{Na(2)}_kab
(b,3) -> i: {succ(Na(2)).Nb(3)}_kab
i -> (b,6): a.{Na(1)}_kab
(b,6) -> i: {succ(Na(1)).Nb(4)}_kab
i -> (a,3): {succ(Na(1)).Nb(4)}_kab
(a,3) -> i: {succ(Nb(4))}_kab
i -> (a,6): {succ(Na(2)).Nb(3)}_kab
(a,6) -> i: {succ(Nb(3))}_kab
i -> (b,3): {succ(Nb(3))}_kab
(b,3) -> i: {K1ab(7).N1b(7)}_kab
i -> (a,3): {K1ab(7).N1b(7)}_kab
i -> (a,6): {K1ab(7).N1b(7)}_kab


% Reached State:
% state_dummy(i,replay_protection_on_k1ab,i,0,17)
% request(a,b,alice_bob_k1ab,K1ab(7),6)
% request(a,b,alice_bob_k1ab,K1ab(7),3)
% request(a,b,alice_bob_na,Na(2),6)
% request(a,b,alice_bob_na,Na(1),3)
% witness(b,a,alice_bob_k1ab,K1ab(7),i)
% request(b,a,bob_alice_nb,Nb(3),3)
% secret(K1ab(7),k1ab,set_77)
% secret(N1b(7),n1b,set_78)
% contains(a,set_77)
% contains(b,set_77)
% contains(a,set_78)
% contains(b,set_78)
% witness(a,b,bob_alice_nb,Nb(3),i)
% witness(a,b,bob_alice_nb,Nb(4),i)
% witness(b,a,alice_bob_na,Na(1),i)
% witness(b,a,alice_bob_na,Na(2),i)
% state_bob(b,i,kib,succ,1,dummy_nonce,dummy_nonce,dummy_nonce,
                          dummy_sk,set_90,set_91,13)
% state_alice(a,i,kai,succ,0,dummy_nonce,dummy_nonce,dummy_sk,
                          dummy_nonce,9)
% state_alice(a,b,kab,succ,6,Na(2),Nb(3),K1ab(7),N1b(7),6)
% state_bob(b,a,kab,succ,3,Nb(4),Na(1),dummy_nonce,dummy_sk,set_84,set_85,6)
% state_alice(a,b,kab,succ,6,Na(1),Nb(4),K1ab(7),N1b(7),3)
% state_bob(b,a,kab,succ,5,Nb(3),Na(2),N1b(7),K1ab(7),set_77,set_78,3)
```

We can see that the goal "authentication_on alice_bob_k1ab" has been violated. We briefly explain the attack given in the ATTACK TRACE section:

The first legitimate session begins and a sends the first message, which is intercepted by the intruder.

```
i -> (a,3): start
(a,3) -> i: a.{Na(1)}_kab
```

The second legitimate session likewise begins, with the first message intercepted by the intruder. We indent the messages of the second session in order to clearly distinguish them from those of the first.

```
                                    i -> (a,6): start
                                    (a,6) -> i: a.{Na(2)}_kab
```

The intruder forwards the first message of the first session to b, who returns the next message of the protocol

```
i -> (b,3): a.{Na(2)}_kab
(b,3) -> i: {succ(Na(2)).Nb(3)}_kab
```

And likewise for the second session.

```
                                  i -> (b,6): a.{Na(1)}_kab
                                  (b,6) -> i: {succ(Na(1)).Nb(4)}_kab
```

The intruder simply forwards this message to a. He is not playing an active role yet, but simply listening to the messages and silently forwarding them on.

```
i -> (a,3): {succ(Na(1)).Nb(4)}_kab
(a,3) -> i: {succ(Nb(4))}_kab
```

Yet again, the message is simply forwarded to a.

```
                                  i -> (a,6): {succ(Na(2)).Nb(3)}_kab
                                  (a,6) -> i: {succ(Nb(3))}_kab
```

Here we can see the end of the first session.

```
i -> (b,3): {succ(Nb(3))}_kab
(b,3) -> i: {K1ab(7).N1b(7)}_kab
i -> (a,3): {K1ab(7).N1b(7)}_kab
```

And finally the intruder takes some action. Instead of sending the third message of the second session to b and obtaining some response, the intruder simply sends the fourth message of the first session again, which will result in a using this old value of K1ab again.

```
                                  i -> (a,6): {K1ab(7).N1b(7)}_kab
```

2.4 Example 4 - Algebraic Operators

In practise, attacks on security protocols are sometimes based on the algebraic properties of the operators used for encryption. Two operators that are used often are XOR and modular exponentiation, both of which have algebraic properties which may introduce attacks: for instance, both are associative and commutative. XOR also has the cancellation property that X XOR X = 0, while exponentiation has the identity property $X^1 = X$.

Analysis which incorporates properties of algebraic operators is very challenging. HLPSL supports specifying protocol models that use exponentiation and XOR (written, respectively, `exp(g,a)` and `xor(a,b)`), and certain[5] back-ends in the AVISPA Tool can exploit some of the properties of these algebraic operators. We refer the interested reader to the AVISPA User Manual for more information on the technical details of support for properties of algebraic operators.

In this subsection, we consider an involved example which includes the XOR operator, a variant of the well-known Needham-Schroeder public key protocol [15]. The A-B notation of this protocol is shown below:

```
A -> B: {Na.A}_Kb
B -> A: {Nb.xor(Na,B)}_Ka
A -> B: {Nb}_Kb
```

In this authentication protocol, the two participants `A` and `B` share their public keys (`Ka` and `Kb`, respectively) in advance. `A` generates a nonce `Na` and sends it, concatenated together with his own name and encrypted with `Kb`, to `B`. `B` replies with his own nonce `Nb` and the nonce he received from `A`, XOR'ed with `B`, all encrypted with `Ka`. `A`, in turn, can check that this is indeed the nonce she sent by decrypting the message and verifying that `xor(xor(Na,B),B)=Na`. In the last step, `A` replies to `B` with `Nb` encrypted with `Kb`.

The HLPSL specification of this protocol is shown below.

```
role alice (A,B     : agent,
            Ka,Kb   : public_key,
            Snd,Rcv : channel (dy)) played_by A def=

  local
     State  : nat,
     Na     : message,
     Nb     : text

  init State := 0

  transition
    1. State  = 0 /\ Rcv(start) =|>
       State':= 1 /\ Na' := new() /\ Snd({Na'.A}_Kb)
                  /\ witness(A,B,bob_alice_na,Na')
                  /\ secret(Na',na,{A,B})
```

[5] At the time of writing, CL-AtSe supports both and OFMC supports exponentiation.

```
    2. State  = 1 /\ Rcv({Nb'.xor(Na,B)}_Ka) =|>
       State':= 2 /\ Snd({Nb'}_Kb)
                  /\ wrequest (A,B,alice_bob_nb,Nb')

end role
```

```
role bob (B,A    : agent,
          Kb,Ka   : public_key,
          Snd,Rcv : channel (dy)) played_by B def=

  local
     State : nat,
     Na    : message,
     Nb    : text

  init State := 0

  transition
    1. State  = 0 /\ Rcv({Na'.A}_Kb) =|>
       State':= 1 /\ Nb' := new() /\ Snd({Nb'.xor(Na',B)}_Ka)
                  /\ witness(B,A,alice_bob_nb,Nb')
                  /\ secret(Nb',nb,{A,B})

    2. State  = 1 /\ Rcv({Nb}_Kb) =|>
       State':= 2 /\ wrequest(B,A,bob_alice_na,Na)

end role
```

```
role session (A,B: agent,
              Ka, Kb : public_key) def=

  local SA, RA, SB, RB: channel (dy)

  composition
    alice(A,B,Ka,Kb,SA,RA)
    /\ bob(B,A,Kb,Ka,SB,RB)
```

```
end role
```

```
role environment() def=

  const
     a, b, i    : agent,
     ka, kb, ki : public_key,
     bob_alice_na,
     alice_bob_nb,
     na, nb     : protocol_id

  intruder_knowledge = {a,b,i,ka,kb,ki,inv(ki)}

  composition
    session(a,b,ka,kb)
    /\ session(a,i,ka,ki)

end role
```

```
goal
     weak_authentication_on alice_bob_nb
     weak_authentication_on bob_alice_na
     secrecy_of na, nb
end goal
```

```
environment()
```

Analysis with XOR The analysis results of the above protocol using CL-AtSe are shown below. First, the attack found by default is the following one:

```
% avispa NSPKxor.hlpsl --cl-atse

SUMMARY
  UNSAFE
```

```
DETAILS
  ATTACK_FOUND
  TYPED_MODEL

PROTOCOL
  ./NSPKxor.if

GOAL
  Secrecy attack on (n5(Nb))

BACKEND
  CL-AtSe

STATISTICS

  Analysed   : 6 states
  Reachable  : 5 states
  Translation: 0.02 seconds
  Computation: 0.00 seconds


ATTACK TRACE

  i -> (a,6):  start
  (a,6) -> i:  {n9(Na).a}_ki
               & Secret(n9(Na),set_70);  Add a to set_70;  Add i to set_70;

  i -> (a,3):  start
  (a,3) -> i:  {n1(Na).a}_kb
               & Secret(n1(Na),set_57);
                 Witness(a,b,bob_alice_na,n1(Na));
               & Add a to set_57;  Add b to set_57;

  i -> (b,4):  {xor(i,xor(b,n9(Na))).a}_kb
  (b,4) -> i:  {n5(Nb).xor(i,n9(Na))}_ka
               & Secret(n5(Nb),set_66);
                 Witness(b,a,alice_bob_nb,n5(Nb));
               & Add a to set_66;  Add b to set_66;

  i -> (a,6):  {n5(Nb).xor(i,n9(Na))}_ka
  (a,6) -> i:  {n5(Nb)}_ki
```

The attack trace printed by CL-AtSe by default may not be minimal. This is in fact the first attack found by CL-Atse. However, it is possible to ask CL-Atse to output one of the smallest attacks (in number of steps) with the -short option:

```
% avispa NSPKxor.hlpsl --cl-atse -short

SUMMARY
  UNSAFE

DETAILS
  ATTACK_FOUND
  TYPED_MODEL

PROTOCOL
  ./NSPKxor.if

GOAL
  Secrecy attack on (n5(Nb))

BACKEND
  CL-AtSe

STATISTICS

  Analysed   : 6 states
  Reachable  : 5 states
  Translation: 0.01 seconds
  Computation: 0.01 seconds


ATTACK TRACE

  i -> (a,6):  start
  (a,6) -> i:  {n9(Na).a}_ki
               & Secret(n9(Na),set_70);  Add a to set_70;  Add i to set_70;

  i -> (b,4):  {xor(i,xor(b,n9(Na))).a}_kb
  (b,4) -> i:  {n5(Nb).xor(i,n9(Na))}_ka
```

```
                & Secret(n5(Nb),set_66);
                  Witness (b,a,alice_bob_nb,n5(Nb));
                & Add a to set_66;  Add b to set_66;

  i -> (a,6):   {n5(Nb).xor(i,n9(Na))}_ka
  (a,6) -> i:   {n5(Nb)}_ki
```

Clearly, an attack has been found. Note that the CL-AtSe back-end uses a slightly different format for some aspects of the attack traces than OFMC does. For example, it writes an interpretation of the IF facts as tests or actions in the attack trace. We now examine this attack.

The first message, used to initiate the protocol, is the start message sent by the intruder.

Agent a replies with the first message of the protocol. This reply is shown in the second line of the trace:

```
(a,6) -> i:   {n9(Na).a}_ki  & Secret(n9(Na),set_70)
```

The nonce she generates is given value n9(Na): an instance of nonce Na. a also emits a secret fact declaring n9(Na) to be secret. For reasons internal to the AVISPA Tool, the set of agents sharing the secret n9(Na) is stored in an automatically generated variable called set_70, though this can generally be safely ignored. The two events Add a to set_70 and Add i to set_70 in this protocol step indicate that both a and the intruder (i) are members of the set: that is, the intruder is allowed to know this "secret".

The intruder now XORs his own name together with that of b and the newly learned nonce n9(Na), and sends the result to b as though it was a fresh nonce coming from a.

```
i -> (b,4):  {xor(i,xor(b,n9(Na))).a}_kb
```

b replies in turn, generating his own nonce n5(Nb) and XOR-ing the nonce he believes came from a (i.e. xor(i,xor(b,n9(Na)))) together with his own name, yielding the following message:

```
(b,4) -> i:  {n5(Nb).xor(i,n9(Na))}_ka
              & Secret(n5(Nb),set_66);
                Witness(b,a,alice_bob_nb,n5(Nb));
```

Note that the second component of this message can be reduced using the properties of XOR: xor(i,n9(Na)) = xor(b,xor(i,xor(b,n9(Na)))).

This message is thus precisely what a in session 6 is expecting as the second message of the protocol. The intruder simply forwards it to her:

```
i -> (a,6):  {n5(Nb).xor(i,n9(Na))}_ka
```

Finally, a interprets n5(Nb) as the intruder's nonce and sends it to him, encrypted with ki.

```
        (a,6) -> i:  {n5(Nb)}_ki
```

This last message represents a secrecy attack, as instance 4 of b has declared the value n5(Nb) to be secret, and obviously the intruder possess it. Examining the HLPSL specification, we can see that, specifically, he will declare it to be a secret shared between a and b, so the intruder should not be able to learn it. Interestingly, it is a secrecy attack that could not be found without taking into account the algebraic properties of XOR. We note that it is also symptomatic of an authentication attack, as instance 4 of agent b also believes that the value xor(i,xor(b,n9(Na))) originates from agent a. This is of course not true, and were b to assert his belief with his wrequest fact in his final step of the protocol, an attack would result, but the secrecy attack is, in this case, found first. To search for the authentication attack, one can comment out the secrecy_of goal in the HLPSL specification, leaving only the two authentication goals.

3 HLPSL Tips

3.1 Priming Variables

Always remember that if a variable is being assigned a new value, then the variable name on the left-hand side of the := must be primed. If you would like to refer to the value of a variable that is assigned a new value in the current transition, then using prime will refer to the new value, and not using prime will refer to the old value of the variable.

Here are some guidelines:

- In the RCV channel, if you are receiving a new value then the variable used to store this value should be primed.
- In the SND channel, if you are sending an old value, don't prime the variable.
- If sending a value just received or computed in the same step, then prime the variable.
- A local variable should be assigned a value before first reading or sending it: either in the init section (without primes) or by assigning a value to its primed instance.

3.2 Witness and Request

When using witness and (w)request, the third argument is an identifier of type protocol_id declared in the top-level role. This is used to associate the witness and (w)request predicates with each other and to refer to them in the goal section.

3.3 Secrecy

If you want to express that a certain value (represented by a term `T`) produced or selected by a role played by `A` is a shared secret between A and a set of agents (say, `B` and `C`), then write the following `secret` facts in role played by `A` in the transition in which `T` is determined:

```
secret(T,t,{A,B,C})
```

The label `t` (of type `protocol_id`) is used to identify the goal. In the HLPSL goal section, give the statement `secrecy_of t` to refer to it.

Give the secrecy events as early as possible, i.e. right when the secret term has been created in the respective role(s), because the secrecy check takes effect only after the events have been issued and it will stay in effect till the end of the protocol run.

If a value `T` that should be kept secret is determined by a single role (in particular, if it is an atomic value like a nonce produced by `new()`), then the secrecy statement should be given in — and only in — the role introducing the value.

If the secret is a combination of ingredients from several roles, then secrecy predicates should be given in all roles contributing to the non-atomic secret value. Unfortunately, if the intruder plays one of these roles in one session and legitimately learns the "secret", then he can re-use this value in some other session (where he does not play the role of an honest agent) to masquerade as one of the honest agents, while the other agents believe that the value is a shared secret between honest agent only, and this attack cannot be detected. Still, this should not be a serious problem, since it is indicative of an authentication attack, which should be found nevertheless.

If a role played by `A` shares a secret `T` with some player `U` of another role, and the identity of `U` is not accessible for `A` (e.g. because of anonymity), the predicate `secret(T,t,{U})` cannot be given in the role of `A`. In this case, it should be given in the role of `U` instead, right after the transition that sends `T` to `U` has been authenticated.

3.4 Constants and Variables

Do not forget to declare all constants used in your model and to give them a suitable type. Otherwise, the compiler will warn and the back-ends might yield unexpected results. Also, do not use the same variable (or constant) name in different roles with different types.

Furthermore, in each role all local variables that are not of type `channel` and

- do not occur primed within a receive action on left-hand side of a transition, and
- are not assigned to using `:=` on the right–hand side of a transition

before being read, should be assigned an initial value in the `init` section of the role.

3.5 Concatenation (.) and Commas (,)

Full-stops (e.g. ".") should be used when composing messages. For example:

```
SND(A.B.Na')
```

Commas (e.g. ",") should be used when passing multiple arguments to functions and events, e.g.:

```
secret(Kab,kab,{B})
```

Note that "." is associative while "," is not. Thus, `(A.B).Na' = A.(B.Na')`, which allows to check for (a limited sort of) type-flaw attacks.

3.6 Exploring executability of your model

The "`-p`" (ply) option for OFMC allows you to easily step through the search tree for a given protocol. An important symptom of errors in specifications is that they can cause a protocol model to be unexecutable. OFMC's `-p` flag is thus a useful debugging tool to check manually that your protocol specification allows agents to execute all the steps required for an honest run of the protocol.

For instance, the following command yields the "root node" of the search space and a list of possible transitions available from this point numbered from 0 upwards:

```
% avispa protocol.hlpsl --ofmc -p
```

To explore a particular branch of the search tree, select a transition number and call the AVISPA Tool again with the ply option and the transition number. For example, the following command will take the first transition from the root node, and display both a history of what has happened in the protocol and a list of transitions available from this new point:

```
% avispa protocol.hlpsl --ofmc -p 0
```

In general, one can pass a path in the search tree, represented by transition numbers separated by spaces, to OFMC's ply option. The command below will take the first transition from the root node, then the third available transition from there. Once again, you will see a history and a list of possible transitions:

```
% avispa protocol.hlpsl --ofmc -p 0 2
```

This technique is very useful for debugging protocol specifications and can be used to test that your protocol is indeed behaving correctly. We advise that you try to make your way through a normal "run" of the protocol.

There is another useful option for OFMC: the `-sessco` option. In addition to searching for replay attacks as previously discussed, this automatically explores the search tree created to check that all the states/transitions are reachable. If the outcome is negative, it usually indicates an executability problem with your model.

Other back-ends to the AVISPA Tool also offer features that can be very useful when debugging HLPSL specifications. For brevity, we merely list a selection of them here and leave it to the reader to try them out.

CL-AtSe with `-noexec` The `-noexec` option to CL-AtSe does not actually search for attacks, but rather prints out the initial internal system state of the analysis. This includes a listing the initial intruder knowledge, terms the intruder cannot forge like private keys, and the elements initially contained in sets. In addition, the possible transitions for each role instance are listed. If a given role instance is at a choice point (for instance, an if-then-else branching point), then the different branches of this choice point, as well as their trigger conditions, are listed as well. This gives a nice view of what each role instance can do, and under what conditions.

SATMC's Executability Check SATMC also includes functionality to check the executability of a HLPSL specification. Running SATMC and passing the module option

```
--check_only_executability=true
```

will produce an explicit listing of the rules of the translated IF specification and an assessment of the executability of each one. SATMC is particularly strict about the proper use of types in HLPSL specifications; this feature can thus be very useful for finding errors relating to typing that may lead to non-executability of a protocol specification.

3.7 Detecting Replay Attacks

The current version of the AVISPA Tool does not fully support repeating declared sessions (although OFMC's `-sessco` option gives a good approximation). This may lead to a situation where a replay attack is not detected. A work-around (which unfortunately slows down the verification considerably) for this is to declare two valid sessions in the top-level composed role: for instance, two parallel sessions between A and B (see example 3 from Section 2).

3.8 Instantiating Sessions

Session instantiation sometimes appears simpler than it actually is. Usually, the situation is as follows: there is a top-level role usually called `environment`. In the `environment` role, a number of sessions are instantiated corresponding to the composed role `session`. The `session` role usually instantiates one instance of each basic role. For instance, alice and bob.

The code might look something like this:

```
role environment()
```

```
def=

  const a,b: agent

  composition

     session(a,b,...)
  /\ session(a,i,...)

end role

role session(A,B: agent,...)
def=

  composition
    alice(A,...) /\ bob(B...)

end role

role alice(A: agent,...)
played_by A def=
  ...
end role

role bob(B: agent,...)
played_by B def=
  ...
end role
```

The above means that there are three agents (or principals) taking part in this scenario: `a`, `b` and `i`. In two of the sessions, `a` plays role `alice`: call these two instances alice 1 and alice 2 (see Figure 3). In the first session, the role of `bob` is played by `b` (call this instance bob 1), while in the second session, it is played by the intruder (bob 2).

Currently, HLPSL passes variables by value (except for sets, which are passed by reference). This means that alice 1 and alice 2 have separate copies of all local variables and are effectively separate state machines. They share, however the common identity `a` passed as parameter `A`, and constants like `na`, which are used in the `witness` and `(w)request` predicates. For example, `request(A,B,alice_bob_na,Na)`.

An interesting example of when this is important is shown in example 3, where the `environment` role contains the following code:

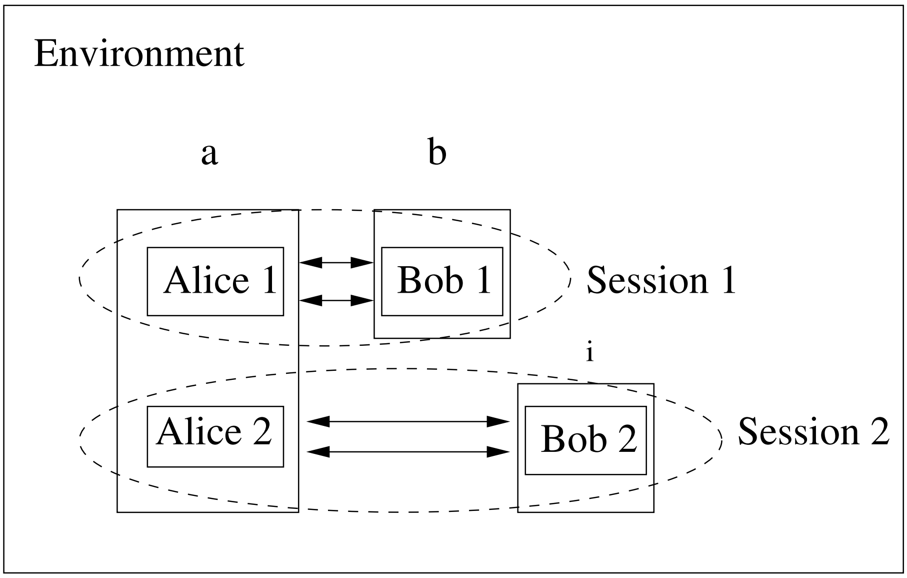

Figure 3: A valid representation of role instantiation

```
composition

   session(a,b,kab)
/\ session(a,b,kab)
   ...
```

Essentially, this code sample is stating that there are two identical sessions between the same client and the same server (`a` and `b`). This is a requirement of the attack on the Andrew secure RPC protocol. The final message (below) will not be accepted by any other than the original client because it is encrypted with their shared key.

```
B -> A : {K1ab, N1b}_Kab
```

Consider the following change to example 3:

```
composition

    session(a1,b,kab1)
/\  session(a2,b,kab2)
    ...
```

As a result of this change, the attack is not found. Thus, role instantiation is quite subtle, and one should specify analysis scenarios carefully.

3.9 Function Results

In the current version of the AVISPA Tool, the return types of functions are not fully supported. In particular, all function results (like, e.g., hashed values and keys looked up via a function) implicitly have the most general type `message`. This can cause subtle executability problems when receiving in a variable a value that has been computed by the peer using a function. If you do not give the variable the type `message`, reception will fail.

3.10 Declaring Channels

For reasons related to the semantics of HLPSL, each channel used in a specification should be different from the others. This can be easily achieved by declaring a different HLPSL variable for each DY channel.

The fine details of this are related to the formal semantics of HLPSL and may be interesting to advanced users. For most users, however, the following modelling rule of thumb should suffice: each agent must have access to a channel for sending and a different channel for receiving. Channels should be variables and cannot be declared as constants. We encourage modellers to declare channels as local variables of the `session` role or in the `environment` role. One should avoid declaring channels in the basic roles. Whether or not agents agree on channel names does not really matter. Look at the traces produced by the back-ends and you will see that everything is sent to and received from the intruder anyway.

For completeness, we also note that channels cannot be owned. We have not explicitly discussed variable ownership in this tutorial, so this tip can safely be ignored by users who are not modelling protocol specifications which include ownership.

4 Questions and answers about HLPSL

- Q: Should `(w)request` appear in the transition where the data is received, or in the last transition? Does it matter?
  A: Yes, it does indeed matter. The `(w)request` term should be emitted in the transition after which the authentication should be considered successful. It's not always as simple as the transition in which certain data is received. For instance, two agents might be authenticating one another based on a shared key. Invariably, in an asynchronous protocol, one agent will have all the key material before the other. But she shouldn't emit her `(w)request` term before she can assume that her communication partner has all the keying material as well, because otherwise he won't have emitted his `witness` term and a false attack will result.

- Q: How is the `message` data-type different to the `text` type?
  A: `message` is the supertype of all types including e.g. `nat` and `text`, while the latter stands for uninterpreted bit-strings.

- Q: What does `secret(T1,t1,{A})` actually mean? That the value of the term is known only to `A`, or that the value is shared between the current role and `A`? It seems to be ambiguous in some situations.
  A: It means that the value is known to `A` (and any other role sets `RS` for which a predicate `secret(T2,t2,RS)` is given where `T1=T2`).

- Q: When should the intruder be allowed to assume a role?
  A: Protocols which include a trusted server will sometimes assume implicitly that this server is honest: that is, there may be trivial attacks if this server is compromised. For this reason, one generally does not declare analysis scenarios in which the intruder assumes any such trusted roles. Roles of non-trusted end participants can generally be played by the intruder.

- Q: Do the tools support the spontaneous transitions (e.g. ”`--|>`”) described in D2.1?
  A: At the time of writing, not yet.

- Q: Are temporal logic style goals supported by HLPSL?
  A: Not yet. Incorporating temporal logic security goals is planned for the next release of the AVISPA Tool.

- Q: Is `exp` a special function like `inv`? What exactly does it mean?
  A: Yes, `exp` is special. It is an operator in the message algebra defined in the `prelude.if` file to have the following properties:

  `exp(exp(X,Y),Z) = exp(exp(X,Z),Y)` and `exp(exp(X,Y),inv(Y) = X`

- Q: Where can the most up-to-date documentation on HLPSL and IF be found?
  A: The AVISPA User Manual [5] is the best resource.

A Symbols and Keywords

Symbol	Description	Example
`.`	associative concatenation (of messages)	`SND(ABC.XY.Z)`
`,`	separates elements of a set, or arguments of a predicate or role	
`'`	prime, used for referring to the next (new) value of variable in a transition	`X'`

`;`	sequential composition of roles	`Phase1(...);`
		`Phase2(...)`

`%`	comment (until end of line)	
`:=`	initialisation (of local variable) in init-section	`init X := 1`
`:=`	assignment to (primed!) local variable	`X' := 1`
`=`	equality test of assigned variables or other expressions	`X = 1`
`<`	less than	`X < 2`
`/\`	conjunction (logical AND)	`X = 2 /\ Y = 3`
`/\`	parallel composition of roles	`alice(...) /\ bob(...)`

`/\_`	conjunction over elements in a set	`/\_{in(A,Agents)}`
		`Kr(A)=[]`

`->`	mapping from one data-type to another	`KeyMap: agent -> public_key`
`=|>`	immediate transition	`RCV(X) =|> SND(Y)`
`{ }`	set delimiter e.g. in knowledge declaration	`{a,b}`
`{ }_`	encryption or signature	`SND({X}_K)`
`( )`	indicates arguments of function, send or receive statement, or role.	

`accept`	used in sequential composition to indicate when a role is finished and the new role can begin	`accept`
		`State=5 /\ Auth=1`

`agent`	data-type for agents	
`bool`	data-type for boolean values	
`channel(dy)`	data-type for channels. Currently only Dolev-Yao channels implemented.	
`composition`	marks beginning of composition section of a composed role	
`cons`	add element to set	`L' = cons(X,L)`

Symbol	Description	Example
`def=`	indicates beginning of body of a role	
`delete`	delete element from set	`L' = delete(X,L)`
`end role`	indicates end of role	
`exp`	exponentiation operator (prefix)	`exp(g,x)` represents g^x
`hash_func`	data-type for one-way functions	
`i`	intruder's identity	
`in`	check if element is in list or set	`in(X,L)`
`init`	indicates initialisation of local variables	`init State := 0`
`inv`	inverse of a key: given a public key returns private key	`inv(Ka)`
`intruder_knowledge`	defines knowledge of the intruder	`intruder_knowledge={a,kai}`
`local`	indicates local variable section	`local State : nat`
`message`	general type of message contents	
`nat`	data-type for natural numbers	
`not`	logical negation	`not(in(X,L))`
`owns`	ownership of a variable: if a role owns a variable, only this role may change the value of the variable	`owns X`
`played_by`	for basic roles: specifies which agent is playing this role	`played_by A`
`public_key`	data-type for public keys	
`request`	used to check strong authentication (together with `witness`)	`request(A,B, alice_bob_na,Na)`
`secret`	used to check secrecy	`secret(K,k,{A,B})`

`set`	data-type for unordered collection of typed values	`local S : text set`
		`init S := {}`

`symmetric_key`	data-type for symmetric keys	
`text`	data-type for uninterpreted bit-strings (like nonces)	
`transition`	marks beginning of transitions section of basic role	
`witness`	used to check authentication (together with `(w)request`)	`witness(B,A, bob_alice_na,Na)`
`wrequest`	used to check weak authentication (together with `witness`)	`wrequest(A,B, alice_bob_na,Na)`
`xor`	prefix xor operator	`xor(a,b)`

References

[1] A. Armando and L. Compagna. SATMC: a SAT-based model checker for security protocols. In *Proceedings of the 9th European Conference on Logics in Artificial Intelligence (JELIA'04)*, volume 3229 of *LNAI*, pages 730–733, Lisbon, Portugal, 2004. Springer-Verlag.

[2] The AVISPA Tool v1.1. Available at `http://www.avispa-project.org/`, 2006.

[3] AVISPA. Deliverable 2.1: The High-Level Protocol Specification Language. Available at `http://www.avispa-project.org/publications.html`, 2003.

[4] AVISPA. Deliverable 2.3: The Intermediate Format. Available at `http://www.avispa-project.org/publications.html`, 2003.

[5] AVISPA. The AVISPA User Manual. Available at `http://www.avispa-project.org/publications.html`, 2005.

[6] D. Basin, S. Mödersheim, and L. Viganò. OFMC: A Symbolic Model-Checker for Security Protocols. *International Journal of Information Security*, 2004.

[7] Y. Boichut, P.-C. Héam, O. Kouchnarenko, and F. Oehl. Improvements on the Genet and Klay Technique to Automatically Verify Security Protocols. In *Proceedings of Automated Verification of Infinite States Systems (AVIS'04)*, ENTCS, 2004. To appear.

[8] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, 1990.

[9] Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, J. Mantovani, S. Mödersheim, and L. Vigneron. A High Level Protocol Specification Language for Industrial Security-Sensitive Protocols. In *Proc. SAPS'04*. Austrian Computer Society, 2004.

[10] J. Clark and J. Jacob. A Survey of Authentication Protocol Literature: Version 1.0, 17. Nov. 1997. URL: `www.cs.york.ac.uk/~jac/papers/drareview.ps.gz`.

[11] D. Dolev and A. Yao. On the Security of Public-Key Protocols. *IEEE Transactions on Information Theory*, 2(29), 1983.

[12] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[13] L. Lamport. *Specifying Systems*. Addison-Wesley, 2002.

[14] G. Lowe. A hierarchy of authentication specifications. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop (CSFW'97)*, pages 31–43. IEEE Computer Society Press, 1997.

[15] R. M. Needham and M. D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. Technical Report CSL-78-4, Xerox Palo Alto Research Center, Palo Alto, CA, USA, 1978. Reprinted June 1982.

[16] M. Turuani. The CL-Atse Protocol Analyser. In F. Pfenning, editor, *Proceedings of 17th International Conference on Rewriting Techniques and Applications, RTA*, Lecture Notes in Computer Science, Seattle (WA), Aug. 2006. Springer.