HLPSL Tutorial

— A Beginner's Guide to Modeling and Checking Security Protocols —

Daniel Plasto and Siemens AVISPA team

April 28, 2005

[1] [2]

This document has been written originally by beginners to AVISPA and HLPSL, and has been revised later after clarifying issues with the designers of HLPSL and gaining valuable practical expertise using it. Beginners can follow this evolving experience and learn from mistakes instead of making their own. Our aim is to provide (hopefully) helpful guidance in an easily accessible way.

Section 1 contains a very basic introduction to what HLPSL looks like and how it is used. Section 2 contains three introductory examples that illustrate modeling with HLPSL. With these examples, both correct solutions and erroneous attempts are provided and discussed. Section 3 contains a number of tips which are 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.

In addition to this tutorial, the AVISPA Package User Manual 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.

[1] **Fix:** Remark on current version of this document: The current version of this tutorial originates directly from the work done by Daniel Plasto in 2004. The presented examples have been updated to conform to the current version of *HLPSL* and to run with the current version of *Ofmc*. Only *Ofmc* is used as model-checker since it was the most mature tool at the given time. Till then, the model-checkers *CL-Atse* and *SATMC* have improved considerably and another model-checker *TA4SP* has come up. A future version of this tutorial will have to deal with these model-checkers and further enhancements of *HLPSL*.

[2] **Fix:** ALL AVISPA MODELERS, please contribute to this document with suggestions and corrections. Especially for the tips and questions sections.

Contents

1 HLPSL Basics

AVISPA provides a language called the *High Level Protocol Specification Language (HLPSL)* for describing security protocols and specifying their intended security properties, as well as a set of tools to model-check them.

1.1 Using the AVISPA Tool

[3]

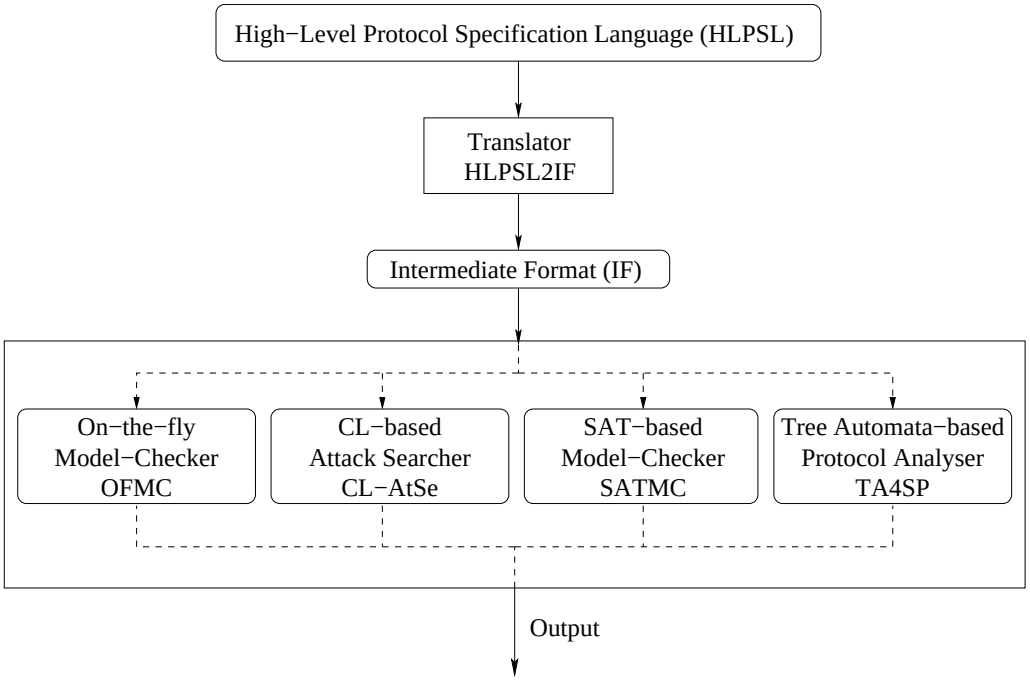

Figure 1: Architecture of the AVISPA Tool v.2

The structure of the AVISPA tool is shown in Fig. 1. A HLPSL specification is translated into the *Intermediate Format (IF)*, using a tool called *hlpsl2if*. Note that this intermediate step is transparent to the user, as the translator is called automatically. If you are interested, you can read more about the IF in the AVISPA User Manual and in the IF specification. [3, 2] The intermediate format specification is then processed by model-checkers to analyze if the security goals are violated. The four backends to the AVISPA tool (Ofmc, CL-AtSe, SATMC, and TA4SP) employ complementary model-checking techniques. Because the methods are complementary

[3] **Fix: PHD:** This and the rest of the document should be updated to reflect the usage of the AVISPA tools and not just the HLPSL2if+OFMC toolchain.

and not equivalent, situations might arise in which the backends return different results. This, however, should be the exception. If a security goal of the specification is violated these tools provide a trace which shows the sequence of events leading up to the violation and displays which goal was violated.

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 backend. The `--backend` option, however, can be used to specify an alternative. For instance, we can analyze the HLPSL file using SATMC as follows:

```
avispa example.hlpsl --backend=satmc
```

In this tutorial, we will focus on invoking the AVISPA tools with the default options. See the usage information for a more complete description of the options not discussed here.

The AVISPA tool, as well as a very helpful XEmacs mode[4] for editing HLPSL specs, are available to download and experiment with at `http://www.avispa-project.org/avispa-package/`. See the INSTALL and README files contained in the package for further information.

There is also a web interface for your first steps with HLPSL and the tools 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 backends to check the given protocol, or even use all of them and then compare their outputs. Outputs are given in a textual form (as done in the examples of this tutorial). Also, if an attack is found on a protocol, the sequence of steps leading to the attack (the trace) is displayed using Message Sequence Charts. The AVISPA web interface is available at `http://www.avispa-project.org/web-interface/`.

Now onto HLPSL ...

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. [4]

[4] **Fix:** where to be found?

```
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 cryptographic key. The protocol assumes the existence of a trusted server `S`. `A` and `B` presumably do not share such a key initially; however, each shares a key with `S`. We call the key shared between `A` (respectively `B`) and `S` `Kas` (respectively `Kbs`). `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 encrypted is denoted using curly breackets). `S`, in turn, decrypts it, re-encrypts it with `Kbs`, and forwards it 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 clearly shows the sequence of events and the exchange of messages in a given protocol. Several protocol specification languages, including the first version of HLPSL, are based on A-B notation. For more complex protocols, however, A-B notation is not expressive enough to capture things like if-then-else constructs, looping and other features. That's why we use a more expressive language like HLPSL.

Once you have this flow of messages, you can begin your HLPSL specification with the basic roles. For each (type of) participant in a protocol, there will be one basic role specifying its blueprint. This blueprint can later be instantiated by one or more agents playing the given role. In the above example there will be three basic roles, often called `Alice`, `Bob`, and `Server`.

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

```
role Alice(A,B,S : agent,
           Kas : symmetric_key,
           ...)
played_by A def=
  local State: nat, Kab: symmetric_key (fresh)
  init  State = 0
  transition
    ...
end role
```

Here is a role known as `Alice`, with parameters `A`, `B` and `S` of type `agent`, and `Kas` of type `symmetric_key`. We'll assume that these values are passed to role `Alice` from somewhere else for now. The parameter `A` appears in the `played_by` section, meaning that any agent given in the argument position `A` behaves as specified by this role. 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. `Kab` is also declared to be `(fresh)`, which intuitively means that `A` will generate its value randomly.[5] The local `State` variable is initialized to 0 in the `init` section. All variables in HLPSL begin with a capital letter, and all constants begin with a lower-case letter.

For information about the different types available in HLPSL and other details, please see the HLPSL specification (Deliverable 2.1). [1]

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` (but actually, the label is unimportant). It specifies that if the value of `State` is equal to zero and a message is received 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'`, but this time encrypted with `Kbs`.

Here we see an example of *priming*. When you see `X'`, it means *the new value of the variable X*. We say *"X prime"* (the notation stems from the temporal logic TLA [6], 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, after transition `step1` fires, the right-hand-side tells us that the value of the `State` variable 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`, because it is primed.

Taking the transition as a whole, we can read it as follows: "whenever the value of `State` is equal to `0` and we receive a message on channel `RCV` whose contents are some value encrypted

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

[6] **Fix:** MR:add reference

with Kab, then we update the value of State to be equal to 2, we update the value of Kab to be equal to the contents of the encrypted message, and we send this new value of Kab, encrypted with Kbs, on channel SND." It's quite a mouthful, but fortunately HLPSL allows us to be more concise.

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

You may be wondering about these RCV and SND statements. These are not special functions, they are actually variables of type channel. They are included as role parameters like this (the type attribute (dy) is explained below):

```
role Alice(A,B,S    : agent,
           Kas      : symmetric_key,
           SND, RCV : channel (dy))
...
```

1.4 Composed Roles

Composed roles instantiate one or more basic roles, "gluing" them together so they execute together, usually in parallel. 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 you'd expect, then we might define a composed role which instantiates one of each 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 we instantiate our basic roles. The /\ operator indicates that these roles should execute in parallel.

The `Session` role usually declares all channels used by the roles as variables of type `channel (dy)`. These variables are not instantiated with concrete constants. The `dy` stands for the Doley-Yao intruder model. [5] This means that the intruder has full control over the network, such that all messages sent by agents will go to the intruder. He may suppress, analyze, and/or modify messages (as far as he knows the required keys), and send them to whoever he pleases. 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()
```

By now, you should have a basic understanding of the structure of HLPSL specifications. If you are new to HLPSL, it is strongly recommended that you continuing reading to the next section of this tutorial. For more detailed information on HLPSL specifications, this can be found in Deliverable 2.1 [1] as well as in the AVISPA Package User Manual. These documents describe 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. To explain, the phrase *share a secret* implies that K is a value known only to A and B. Consider the following protocol for producing a new shared key:

```
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 back 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.

The first two messages are for key agreement, the last one for proof that A has the new key. One property one would like to prove is 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, this property verifies that Nb has not been received by B previously (replay attacks). Of course, the new key K1 should be kept secret.

Below is a simplified view of role Alice in this protocol modeled in HLPSL:

```
role Alice(..,K,..)      % K has to be passed to each role,
                         % so that A and B match on the value of K
... def=

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

  transition

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

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

The first transition is clear: start is a signal to begin the activity. Alice creates a new value for Na (not using the old value of Na) and encrypts this value using key K before inserting the encrypted value into the channel titled SND. After this transition, Alice is in state 2.

The second transition is trickier: `RCV({Nb'}_K)`. Firstly, `Alice` receives a message. Provided 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 `Nb` value is encrypted with a new key represented as `Hash(Na.Nb')`. This key is computed from both `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, the terms `secret`, `witness` and `request` (all of which are related to describing security goals) exist in this model. As these concepts will be covered later in the tutorial, it is safe to ignore them for now.

Example 1:

Alice-Bob Notation:

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

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

  local
    State   : nat,
    Na      : text(fresh),
    Nb      : text,
    K1      : message

  const
    nb      : protocol_id

  init
    State = 0

  transition

  1. State  = 0 /\ RCV(start) =|>
     State' = 2 /\ SND({Na'}_K)
```

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

end role
```

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

  local
    State   : nat,
    Nb      : text(fresh),
    Na      : text,
    K1      : message

  const
    nb      : protocol_id

  init
    State = 1

  transition

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

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

end role
```

```
role Session(
    A,B  : agent,
    K    : symmetric_key,
    Hash : function)
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
    nb          : protocol_id,
    kab,kai,kib : symmetric_key,
    a,b         : agent,
    h           : function

  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
```

```
    Bob authenticates Alice on nb
  end goal
```

```
  Environment()
```

Upon converting this to intermediate format and running the *Ofmc* model-checker, the following output is given:

```
% OFMC
% Version of 2005/04/14
SUMMARY
  SAFE
DETAILS
  BOUNDED_NUMBER_OF_SESSIONS
PROTOCOL
  ex1.if
GOAL
  as_specified
BACKEND
  OFMC
COMMENTS
STATISTICS
  parseTime: 0.00s
  searchTime: 11.79s
  visitedNodes: 5430 nodes
  depth: 12 plies
```

It can be seen that no attacks were found. Or, in other words, the stated security goals were satisfied.

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

[7]

[7] **Fix:** MR: maybe emphasize in the example the faulty parts by boldface or underlining

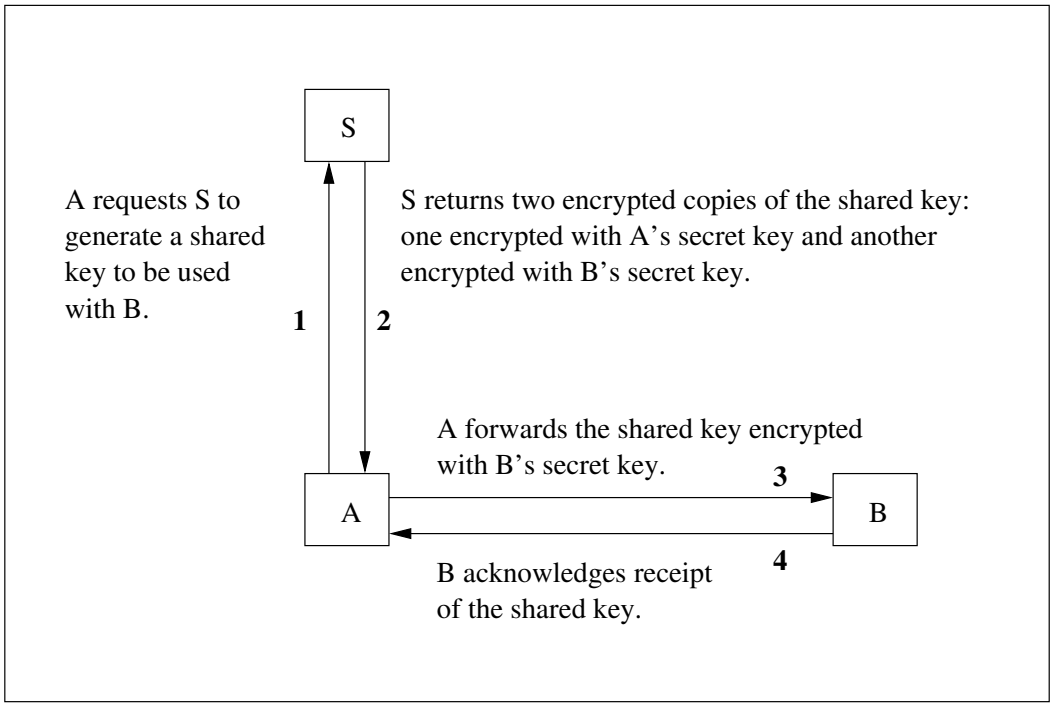

Figure 2: Representation of Kerberos style protocol for this example

This example looks at a Kerberos style protocol with 3 principals: `A`, `B` and `S`. `A` wants 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 process.

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)
```

To provide an insight into the experiences faced as a HLPSL modeler, this example is structured to reflect some of the authors' personal experiences whilst attempting to model this protocol. Firstly, initial attempts at solving this example are included and discussed before the correct solution is finally given. This way, it is anticipated that this will provide a clearer understanding into not only the modeling language itself, but also common errors faced when modeling in HLPSL.

The first attempt at this example is given below. Please note that there are a lot of errors which you may already recognize.

```
role Alice (A,B:agent,...)
played_by A def=

  local State: nat,
        Na   : text(fresh),
        Nb,Ns: text

  init State = 0

  transition

  % Receives START and send the 1st message to Server
  step1. State = 0 /\ RCV(start) =|>
         SND(A.B.{Na'}_Ka) /\ State' = 2

  % Receives keys from server and send message to B
  step2. State = 2 /\ RCV(A.B.{K.Na'.Ns'}_Ka.{K.Na.Ns}_Kb) =|>
         SND(A.B.{K.Na.Ns}_Kb.{Na.Nb'}_K) /\ State' = 3

  step3. ...

end role
```

```
role Bob (A,B:agent,...)
played_by B def=

  local State: nat,
        Na   : text,
        Nb   : text(fresh),
        Ns   : text

  init State = 0

  transition
```

```
  step1. State = 0 /\ RCV(A.B.{K.Na'.Ns'}_Kb.{Na.Nb'}_K) =|>
         SND(A.B.{Nb'.Na}_K) /\ State' = 1

end role
```

```
role Server(....)
played_by S def=

  local A, B : agent,
        Na   : text,
        Ns   : text(fresh),
        State: nat

  init State = 0

  transition

  step1. State = 0 /\ RCV(A.B.{Na'}_Ka) =|>
         SND(A.B.{K.Na.Ns'}_Ka.{K.Na.Ns}_Kb}) /\ State' = 1

end role
```

Whilst attempting to correct this version, much confusion centered around when to prime variables. In the above example, there are several examples of incorrect priming. Priming is actually quite simple if you follow these simple rules:

If your variable is on the left hand side of a transition, it is typically an argument 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 contains in its first component, a value that 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.

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

```
State' = 3
```

Also use primes for freshly generated nonces, e.g.:

```
SND(Na')
```

A primed variable 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, if you have a transition like this:

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

you must prime both variables. Otherwise you would be sending the current value of `X` rather than forwarding the recently received value.

Another issue arising from this first attempt centers around the concern of when variables should be shared. In this example, all three roles know a variable named `K`. However, it is inappropriate for them to share the same 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 (perhaps an attack?).

In fact, in HLPSL, *variables* (which have the ability to change) cannot[8] be shared (except for sets and lists which are passed by reference). Yet, it is appropriate and possible to share constant *values* when you require roles to have pre-existing knowledge - for instance, a shared key. This is accomplished by passing them as an argument to several roles.

It is important to note the difference between using commas "`,`" and using decimal-points "`.`" in HLPSL specifications.

When sending or receiving data from a channel, one should always use decimal points as they indicate concatenation, e.g.:

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

When passing arguments to a function and for role arguments, commas should be used to separate them, e.g.:

[8] **Fix:** Change this when necessary

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

There is another problem with the example which is very important for properly modeling the protocol.

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

Apart from some problems with the priming of variables, this should be written differently: `A` receives `(A.B.{K.Na'.Ns'}_Ka.X)` where `X` is a bunch of data that `A` cannot understand. He expects it to be of the form `{K.Na.Ns}_Kb`, for some `Kb`, but even if he knows the plaintext `(K.Na.Ns)`, he cannot match `X` to `{K.Na.Ns}_Kb` because he does not know the key `Kb`.

The transition should be written as:[9]

```
step2. State = 2 /\ RCV(A.B.{K.Na'.Ns'}_Ka.X') =|>
       SND(A.B.X'.{Na.Nb'}_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. This format is something the recepient might be able to check even without knowing the symmetric key used for encryption. One could also give `X` the most general type `message`, but this will slow down the model-checkers.

A corrected version of Example 2 given below:

```
role Alice (A, S, B: agent,
            Ka     : symmetric_key,
            SND_SA, RCV_SA, SND_BA, RCV_BA: channel(dy))
played_by A
def=

  local State : nat,
        Na    : text (fresh),
        Ns    : text,
        K     : symmetric_key,
```

[9] **Fix:** **PHD:** Of course, this isn't really *incorrect* because we don't explicitly reason about honest agent knowledge. But perhaps we should leave this out of the tutorial and explain it in more detail in the user manual. **DvO:** The above is of course not technically incorrect, but I think we have made clear enough why that version is not the desired one.

```
        X     : {symmetric_key.text.text}_symmetric_key

  const na    : protocol_id

  init   State = 0

  transition

  1. State  = 0 /\ RCV_BA(start) =|>
     State' = 2 /\ 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,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    : text(fresh),
        K     : symmetric_key(fresh),
        Na    : text

  init  State = 1

  transition

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

  const na        : protocol_id

  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,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

  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
  Alice authenticates Bob on na
end goal
```

```
Environment()
```

Here you can see the problems mentioned previously have been fixed, the full parameters have been filled in, and the *composed* roles `Session` and `Environment` have been given.

Notice how the state numbers are even for Alice, and odd for Bob and the Server, which reflects the intended order of send and receive events. This is not compulsory, however, it is a convenient convention which keeps things clear while reading the HLPSL and the traces printed

by the model-checkers.

Note that for non-trivial HLPSL specifications, it is often the case that, due to some modeling mistake(s), the model [10] cannot execute until its intended end. This is very problematic because the model-checkers might not find an attack just 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 modeling mistakes leading to blocked transitions are mismatches of expected and actually sent values – or even harder to spot, mismatches of their types! See subsection 3.10 for a particularly nasty case.

A less frequent but very tricky 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.

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 somehow fool the protocol. He even has a shared key with the server (`ki`) with which he can communicate in a regular manner.

When this revised model of the protocol is tested using *Ofmc*, the following output is provided:

```
% OFMC
% Version of 2005/04/14
SUMMARY
  UNSAFE
DETAILS
  ATTACK_FOUND
PROTOCOL
  ex22.if
GOAL
  a authenticates b on na
%request(a,b,na,Na(1))
BACKEND
```

[10] **Fix:** MR:protocol?

```
  OFMC
COMMENTS
STATISTICS
  parseTime: 0.00s
  searchTime: 0.17s
  visitedNodes: 26 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.x60
(a,3) -> i: a.b.x60.{Na(1).Ns(2)}_K(2)
i -> (a,3): a.b.{Ns(2).Na(1)}_K(2)


% Reached State:
% wrequest(a,b,na,Na(1))
% state_Alice(a,s,b,ka,6,Na(1),Ns(2),K(2),x60,3)
% state_Server(s,a,b,ka,kb,1,dummy_nonce,dummy_sk,dummy_nonce,3)
% state_Bob(b,a,s,kb,5,dummy_nonce,dummy_nonce,dummy_sk,3)
% state_Alice(a,s,i,ka,0,dummy_nonce,dummy_nonce,dummy_sk,
                          {dummy_sk.dummy_nonce.dummy_nonce}_dummy_sk,7)
% 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_sk,dummy_nonce,12)
% secret(K(2),s)
% secret(K(2),i)
% secret(K(2),a)
% state_Server(s,a,i,ka,ki,3,Ns(2),K(2),Na(1),7)
```

The output following the keyword `Reached State` describes the events already seen (in this case, `wrequest(a,b,na,Na(1))` and a number of `secret` statements for the nonce value `K(2)`, plus the states of the various active agents).

The important parts of this are "`UNSAFE`", the violated goal, which was: "`a authenticates b on na`", and the attack trace, which is explained below:

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

The intruder initiates the first session (which for some strange reason is labeled with "3"[11])

[11] **Fix: PHD:** We are the tool developers, so none of this is allowed to be "strange" to us. We can say that

by sending the special `start` message to `A`.

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

`A` sends a message to the intruder, or rather, the intruder intercepts the message sent by `A`.

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

The intruder then forwards message to `S`. However, two small details have been changed: the receiving instance of `S` belongs to the second session, labeled with "7", rather than the original session. Moreover, the message now tells `S` that `A` wishes to talk to `i`, rather than `B`. Note that the intruder has not broken the encryption, but has simply copied the encrypted data into the new message.

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

`S` sends back to the intruder an appropriate response including nonces encrypted with a key shared between the `S` and the intruder. The intruder now has knowledge of these nonces, as well as `K`, a session key which `A` will believe he can use to talk to `B`.

```
i -> (a,3): a.b.{K(2).Na(1).Ns(2)}_ka.x60
```

The intruder sneakily switches the name of the correspondent back to `B` so that `A` is none the wiser.

```
(a,3) -> i: a.b.x60.{Na(1).Ns(2)}_K(2)
```

Now `A` sends the nonces encrypted with the intruder's key to `B`, but they are again intercepted by the intruder. (`A` still believes that they are encrypted with the key of `B`.)

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

Now the intruder can easily decrypt these values and send them back encrypted with the new session key `K`, so that `A` will believe she is indeed talking to `B`, but is really talking with the intruder!

the user need not concern himself with the reason because it is related to tool internals, but we can't say "if you don't understand why it's a 3, don't worry, we don't either." **DvO:** It *is* strange form the user's perspective. I'd be happy to remove this remark if the tools get changed to yield some more intuitive numbering.

Consequently, it is clear that the stated security goal "`a authenticates b on n`" was indeed violated. The problem with the protocol is that the identities of the parties were not protected in any way allowing the intruder to tamper with them. An improved version of the protocol was devised which solved this problem. It is included below.

Alice-Bob Notation:

```
A -> S: A.B.{Na}_Ka
S -> A: {K.Na.Ns.B}_Ka. {K.Na.Ns.A}_Kb
A -> B: {K.Na.Ns.A}_Kb. {Na.Ns.A}_K
B -> A: {Ns.Na.B}_K
```

The updated HLPSL model [12] reflecting the above A-B notation to eliminate this attack is:

```
role Alice (A, S, B: agent,
            Ka     : symmetric_key,
            SND_SA, RCV_SA, SND_BA, RCV_BA: channel(dy))
played_by A
def=

  local State : nat,
        Na    : text (fresh),
        Ns    : text,
        K     : symmetric_key,
        Tx    : {symmetric_key.text.text}_symmetric_key

  const na    : protocol_id

  init   State = 0

  transition

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

  2. State  = 2 /\ RCV_SA({K'.Na.Ns'.B}_Ka.Tx') =|>
     State' = 4 /\ SND_BA(Tx'.{Na.Ns'.A}_K')

  3. State  = 4 /\ RCV_BA({Ns.Na.B}_K) =|>
```

[12] **Fix:** MR:specification?

```
      State' = 6 /\ request(A,B,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    : text(fresh),
        K     : symmetric_key(fresh),
        Na    : text

  init  State = 1

  transition

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

  const na    : protocol_id
```

```
  init  State = 5

  transition

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

  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
  Alice authenticates Bob on na
end goal
```

```
Environment()
```

By reading the specification (or by quickly looking at the A-B notation), it is evident that the intruder cannot modify the values of `A` or `B` without detection. This is because they are encrypted from Message 2 onwards. Running this updated protocol through the model checker produces no attacks.

For information on how to specifiy the security goals of a protocol, this is discussed in the following section.

2.3 Example 3 - security goals

This example looks at the Andrew Secure RPC Protocol. Below is the A-B notation for this protocol adapted from page 45 of the Clark and Jacob library [**?**] to reflect the notation style of HLPSL.

```
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 shared key `K1ab` which can be used for further communication. `K1ab` should be *fresh*, i.e. never used

before. The value N1b is sent for use in the future. Both K1ab and N1b are generated by B.

Our first attempt at the protocol is below, keep in mind that the initial definition was misunderstood.[13]

First, buggy version of Example 3:

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

  local  State  : nat,
         Na, Nb : text(fresh)

  init State = 0

  transition

  step1. State  = 0 /\ RCV(start) =|>
         State' = 1 /\ SND({Kab}_Na')

  step2. State  = 1 /\ RCV({Na+1.Nb'}_Kab) =|>
         State' = 3 /\ SND({Nb'+1}_Kab)

  step3. State  = 3 /\ RCV({K'ab.N'b}_Kab)

end role
```

```
role Bob (A, B    : agent,
          Kab     : symmetric_key,
          SND,RCV : channel(dy))
```

[13] **Fix: PHD:** I think this example is really problematic. The K'ab and the Na+1 is, I think, very dangerous and will have readers thinking "Well why can't I write Na+1 if that's what I want to express?" I think this example needs to be restructured alot. Do we really want to show the incorrect models that result if you misinterpret what a protocol means? **DvO:** I think it's helpful to show also erroneous attempts.

```
played_by B def=

  local State  : nat,
        Na, Nb : text(fresh)

  init State = 0

  transition

  step1. State  = 0 /\ RCV({Kab}_Na') =|>
         State' = 2 /\ SND({Na'+1.Nb'}_Kab)

  step2. State  = 2 /\ RCV({Nb+1}Kab) =|>
         State' = 4 /\ SND({K'ab.N'b}_Kab)

end role
```

On a glance we can spot something wrong: `SND({Kab}_Na')` should read `SND({Na'}_Kab)` in `step1` of `A`. [14]

We can also see the variables `K'ab` and `N'b`. These will soon be changed to `K1ab` and `N1b` and declared in the local statements of both roles.

The state numbers are somewhat confusing and should be rewritten.

Also, in role `A`, `Na` should be declared as `text(fresh)`, while `Nb` should simply be `text` and vice-versa in role `B`. This is because in role `A`, `Na` is freshly generated while `Nb` is just some value received in a message. In role `B`, `Na` is just a value while `Nb` is freshly generated.

Another problem here is that we do not have the "`+`" operator in HLPSL so we cannot write "`Na + 1`". The solution to this is to use a function called `succ` (for successor), however we are unsure how suitable this approach is. The problem is that the intruder is not capable of reversing the `succ` function, whereas in real life he would be quite capable of obtaining the value of `Na` given `Na+1`.[15]

Second, correct version of Example 3:

```
role Alice (A, B     : agent,
```

[14] **Fix:** MR:is this mistake interesting to introduce ? i do not think so

[15] **Fix:** **PHD:** I think it's fine to explain the pros and the cons of a particular modelling decision. But this sounds a bit unprofessional, and I do think it can be a problem to include this + stuff when it's not even valid syntax. **DvO:** Well, this is a typical example of the problems a modeler has in real life.

```
            Kab      : symmetric_key,
            Succ     : function,
            SND, RCV : channel(dy))
played_by A
def=

  local State  : nat,
        Na     : text (fresh),
        Nb     : text,
        K1ab   : symmetric_key,
        N1b    : text

  const k1ab, na, nb : protocol_id

  init State = 0

  transition

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

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

  4. State  = 4 /\ RCV({K1ab'.N1b'}_Kab) =|>
     State' = 6
                /\ request(A,B,k1ab,K1ab')
                /\ request(A,B,na,Na)

end role
```

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

```
  local State  : nat,
        Nb     : text (fresh),
        Na     : text,
        K1ab   : symmetric_key (fresh),
        N1b    : text (fresh)

  const k1ab, na, nb : protocol_id

  init State = 1

  transition

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

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

end role
```

```
role Session(A, B  : agent,
             Kab   : symmetric_key,
             Succ  : function)
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 k1ab, na, nb  : protocol_id,
        a, b          : agent,
        kab, kai, kib : symmetric_key,
        succ          : function

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

  composition

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

end role
```

```
goal
  secrecy_of K1ab, N1b
  Bob authenticates Alice on nb
  Alice authenticates Bob on na
  Alice authenticates Bob on k1ab % freshness of k1ab
end goal
```

```
Environment()
```

This example correctly corresponds to the A-B notation definition above.

A short remark concerning the `goal` section. The current[16] implementation of *Ofmc* does not use the `goal` section but assumes a more low-level and precise means to describe security goals, using `secret`, `witness` and `request`.[17]

In this example we would like to ensure that the new key `K1ab`, and the generated nonce `N1b` are kept secret among `A` and `B`. So in the creating role, we place these lines (where the primes are required there to refer to the new values of `K1ab` and `N1b`):

```
/\ secret(K1ab', A) /\ secret(K1ab', B)
/\ secret(N1b' , A) /\ secret(N1b' , B)
```

indicating that `B` allows that the two values are shared between (only) `A` and `B`.

The `witness` and `request` events are used to check that a principal is right to assume 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.

Suppose `A` has received some data and would like to ensure that it came from `B` within the same session. We place the line

```
/\ request(A,B,na,Na)
```

in the transitions of `A`. It reads as follows: "I am `A`, and I know `Na`, and expect that `B` already asserted to know the same value `Na`, and that I have not checked this value before."[18] The third argument `na`, used for distinguishing different authentication pairs, is usually the lower case of the name of the variable being checked, and should be declared as a constant of type `protocol_id` in the top-level role.

There is also `wrequest` which corresponds to weak authentication. No such replay protection is imposed if one uses `wrequest`.

Now the matching `witness` predicate will appear in `B`, as part of the transition in which the value `Na` is sent to `A`.

```
/\ witness(B,A,na,Na)
```

[16] **Fix:** Update when this changes.

[17] **Fix: PHD:** This is not strictly accurate. We do not use `witness/request` because Ofmc does not read the goals section. That Ofmc does not read the goal section simply means that Ofmc assumes that the temporal formula for which "A authenticates B..." is an abbreviation is always the same. We would still have the witness and request terms even if Ofmc read the goals section, so it is not a case of "instead". **DvO:** I changed the above. Fine now?

[18] **Fix: PHD:** I think this should be a local statement. That is, Alice is not saying to somebody "please make sure" because there is no somebody. Rather, Alice is asserting that she has accepted what seems to come from B. We have some text on this in the SAPS'04 HLPSL paper ... perhaps it could be reused? **DvO:** Slightly changed; fine now?

This reads: "I am B, please remember that I have presented Na to A so that we can check this later."

The semantics of the witness and (w)request predicates when used to describe authentication goals are as follows: if a (w)request predicate is fired, then there must have been a corresponding witness event sometime in the past. By "corresponding" we mean that the agent arguments must be switched, the protocol_id argument must be identical, and the value of the fourth argument must also be identical. request is the same except additionally the value of the fourth argument must not have been "requested" before.[19] [20]

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).

Upon running this through our model-checker, we find that there are no attacks, and so we might conclude that our security goals are valid. However we know that this is not the 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.

Why is this attack not detected? We have a security goal that states: "I am A, and I have received K1ab from B, please check to make sure that B indeed generated the value of K1ab, and that I have not received this value before." This security goal is suitable, but by default the current[21] implementation of *Ofmc* does not consider repeated sessions, therefore replay attacks will not be detected.

To help finding replay attacks, *Ofmc* provides the sessco (session compilation) option.

```
ofmc ex3.if -sessco
```

[19] **Fix: PHD:** Well, the "exact semantics" of witness and request terms are actually just boolean variables. This is the meaning we interpret, and we express this meaning via the declaration in the goals section (currently unfortunately just macros for what are actually temporal formulae). So this is a bit imprecise...here again (and for the description of witness), we have text in the HLPSL paper ... can it perhaps be reused? **DvO:** Slightly changed; fine now?

[20] **Fix:** MR: maybe explain difference with and without (w)?

[21] **Fix:** Update when this changes.

actually finds the replay attack. 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. By the way, the `sessco` option is also handy for a quick check of executability, yet unfortunately it fails if the `State` variables of each roles do not strictly increase from one transition to the next.

A suitable work-around, which also applies to the other model-checkers, is to add another legitimate session between `A` and `B` in our environment role. This session will occur in parallel, in any arbitrary interleaving of state-changes. This will quite naturally allow all model-checkers to find replay attacks.

Third modified version of Example 3[22]:

```
role Environment()
def=

  const k1ab, na, nb  : protocol_id,
        a, b          : agent,
        kab, kai, kib : symmetric_key,
        succ          : function

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

  composition

     Session(a,b,kab)
  /\ Session(a,b,kab)
  /\ Session(a,i,kai)
  /\ Session(i,b,kib)

end role
```

Once this modification was made we found that the attack was detected. Here is the output:

```
% OFMC
% Version of 2005/04/15
SUMMARY
  UNSAFE
```

[22] **Fix:** J: Shouldn't Session be provided with a fourth argument, namely succ?

```
DETAILS
  ATTACK_FOUND
PROTOCOL
  ex31.if
GOAL
  a authenticates b on k1ab
%wrequest(a,b,k1ab,K1ab(7))
BACKEND
  OFMC
COMMENTS
STATISTICS
  parseTime: 0.00s
  searchTime: 1.19s
  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:
% wrequest(a,b,k1ab,K1ab(7))
% state_Alice(6,a,b,kab.succ.6.Na(2).Nb(3).K1ab(7).N1b(7).dummy,6)
% state_Alice(0,a,i,kai.succ.0.dummy_nonce.dummy_nonce.dummy_sk.dummy_nonce.dummy,9)
% state_Bob(1,b,i,kib.succ.1.dummy_nonce.dummy_nonce.dummy_sk.dummy_nonce.dummy,13)
% secret(K1ab(7),a)
% secret(K1ab(7),b)
% secret(N1b(7),a)
```

```
% secret(N1b(7),b)
% witness(a,b,nb,Nb(4)
% state_Bob(3,b,a,kab.succ.3.Nb(4).Na(1).dummy_sk.dummy_nonce.dummy,6)
% state_Alice(6,a,b,kab.succ.6.Na(1).Nb(4).K1ab(7).N1b(7).dummy,3)
% state_Bob(5,b,a,kab.succ.5.Nb(3).Na(2).K1ab(7).N1b(7).dummy,3)
```

We can see that the goal "`a authenticates b on k1ab`" has been violated. Here is an explanation of what has happened taken from the `ATTACK TRACE` section:

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

The first legitimate session begins and `A` sends the first message which is intercepted by the intruder.

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

The second legitimate session likewise begins, with the first message intercepted by the intruder.

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

The intruder forwards the first message of the first session to `B`, who returns the next message of the protocol.

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

And likewise for the second session.

```
i -> (a,3): {succ(Na(1)).Nb(4)}_kab
(a,3) -> i: {succ(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,6): {succ(Na(2)).Nb(3)}_kab
(a,6) -> i: {succ(Nb(3))}_kab
```

Yet again, the message is simply forwarded to `A`.

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

Here we can see the end of the first session.

```
i -> (a,6): {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.

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 in the same step, prime!
- 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.

3.3 Secrecy

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

```
secret(T,A) /\ secret(T,B) /\ secret(T,C)
```

3.4 Constants

Do not forget to declare all constants used in your model and to give them a suitable type. Otherwise, the model-checkers might yield strange results.

3.5 Concatenation (.) and Commas (,)

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

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

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

```
secret(Kab, B)
```

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

3.6 Exploring executability of your model with *Ofmc*

Did you know about the "`-p`" (ply) option for *Ofmc*? It allows you to easily step through the search tree for a given protocol. One 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 two agents to execute all the steps required for an honest run of the protocol.

After you run the AVISPA tool for the first time on a given HLPSL file, say `protocol.hlpsl`, it will have produced the corresponding IF file, `protocol.if`. *Ofmc* itself can be found in the directory `bin/backends` of the AVISPA package directory and can be called directly with the IF file as input as follows:

```
Ofmc protocol.if -p
```

You will be shown the "root node" of the search space and a list of possible transitions available from this point numbered from 0 upwards.

```
Ofmc protocol.if -p 0
```

This 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.

```
Ofmc protocol.if -p 0 2
```

This 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.

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

There is another useful option for *Ofmc*: the `-sessco` option. This automatically explores the search tree created to check that all the states/transitions are reachable. If the outcome is negative, it usually indicates a problem with your model.

3.7 The Goal Section

The goals section is not used by *Ofmc*. However the other model-checkers *do* use it. You should in general include a `goal` section so that you can make use of *SATMC* and *CL-Atse*.[23]

[23] **Fix:** Update when this changes

3.8 Detecting Replay Attacks

Currently,[24] the model-checkers do not fully support repeated sessions. This may lead to a situation where a replay attack is not detected. A work-around (which unfortunately slows down the verification a lot) for this is to declare two valid sessions in the top-level composed role. For instance, two sessions between A and B. (See example 3 from Section 2)

3.9 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=
  ...
```

[24] **Fix:** Update when this changes.

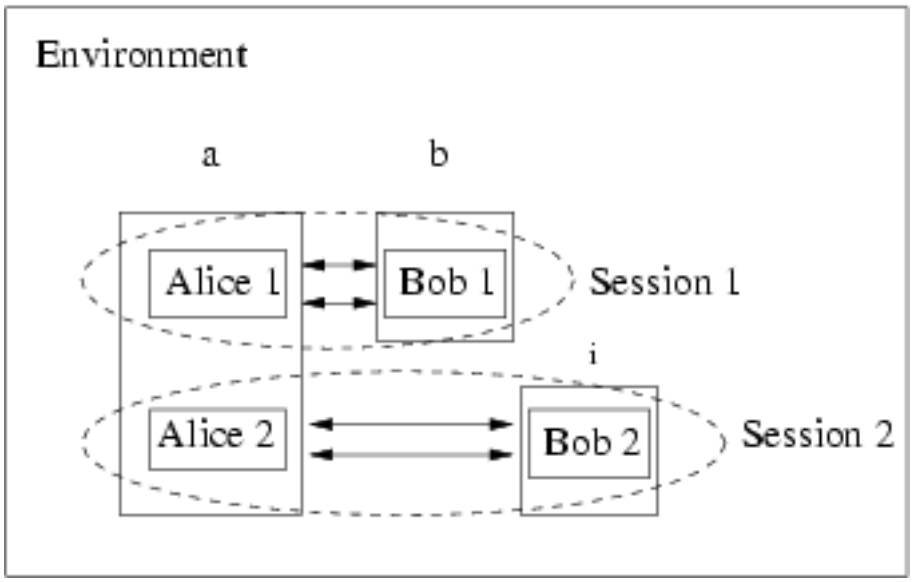

Figure 3: A valid representation of role instantiation

```
end role

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

So what does this mean? Well, there are three agents (or principals) taking part in this scenario: `a`, `b` and `i`. In two of the sessions, `a` plays role Alice: Alice 1 and Alice 2 (see Figure 3). In the first session, the role or Bob is played by `b`, while in the second session, it is played by the intruder.

Currently,[25] HLPSL passes variables by value (except for lists and 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. The only things tying these two roles together are the common identity `a` given as for the formal role parameter `A`, and constants like `na`, which are used in the `witness` and `(w)request` predicates. For example, `request(A,B,na,Na)`.

An interesting example of when this is important is shown in example 3. In the `Environment` role, the following code exists:

```
composition
```

[25] **Fix:** Update when this changes.

```
   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. Now you can see how important it is to understand role instantiation!

3.10 Function Results

Currently[26], the types of functions are not properly supported. In particular, all function results (like e.g. hashed values and key looked up via a function) implicitly have the most general type `message`. This can cause very 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.

3.11 Declaring Channels

You may have noticed that there are many different styles used to define the channels used in a protocol. However, the recommended style is to declare, for each connection used by each rule, a separate channel.

[26] **Fix:** Update when this changes.

Do not worry too much about this. The most important thing is that each agent must have access to a channel for sending and a different channel for receiving. Whether or not agents agree on channel names does not really matter. Look at the traces produced by the model-checkers and you will see that everything is sent to and received from the intruder anyway![27]

[28]

3.12 Global constants and variables

Global constants (confusingly called "rigid variables" by some logicians) are immutable values that are known to and shared between several agents. These are often used; typical examples include names and public keys of agents, and shared symmetric keys. They should be declared in the `Environment` role.

Global variables (also called "flexible variables") are shared between several agents, too, but can change their value over time. They are rarely used, and currently[29] only implemented for sets, which are passed by reference. A global variable produces a direct implicit synchronous communication between the agents sharing it, i.e. the communication is not via sending messages on a channel, and it cannot be seen by the intruder. It is like magic.

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 knows that her communication partner has all the keying material as well, because otherwise he won't have emitted his `witness` term and a trivial 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

[27] **Fix: PHD:** Well, it doesn't really matter from an implementation standpoint, but it's very important from the semantics side. In TLA shared channels could of course lead to contradictions. **DvO:** Slightly changed - fine now?

[28] **Fix:** We may later have a new style of channels in which we will explicitly link channels with the sending and receiving agents, as well as defining other useful features of the channel!

[29] **Fix:** Update when this changes

for uninterpreted bit-strings.

- Q: What does `secret(term, 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 roles `R` for which a predicate `secret(term-with-same-valuse, R)` is given.

- Q: When should the intruder be allowed to assume a role? For example the Needham-Schroeder Public Key Protocol (NSPK) example does not include an intruder impersonating the server. Why is this and when in this appropriate?
  A: Allowing an intruder to assume the role of a trusted server usually violates the security goals of a protocol. Trivially. It is interesting to experiment with intruders impersonating trusted servers but usually the protocol will be dependant on the reliability of this server and so we do not allow an intruder to impersonate it. It is also interesting to experiment with who you can trust by allowing the intruder to assume different roles, however if the protocol specifies that a role is trusted, then it is not really flawed if there is an attack when that role is played by an intruder, is it?

- Q: Do the tools support the spontaneous transitions (e.g. "`--|>`") described in D2.1?
  A: No[30].

- Q: Are temporal logic style requirements supported by HLPSL yet?
  A: No[31].

- 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 certain 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 HLPSL2IF and IF be found?
  A: The AVISPA User Manual [3] is the best resource. Also, in the AVISPA repository under the directories `shared/docs/HLPSL` and `shared/docs/IF` you will find up-to-date versions of D2.1 and D2.3, the HLPSL and IF deliverables. [1, 2]

[30] **Fix:** Update when this changes.
[31] **Fix:** Update when this changes.

A Symbols and Keywords

Symbol	Description	Example
.	associative concatenation (of messages)	SND(ABC.XY.Z)
,	separates elements of a tuple, set, or list, or arguments of a function or role	
’	prime, used for referring to the next (new) value of variable in a transition	X’
;	sequential composition if roles	Phase1(...); Phase2(...)
%	comment (until end of line)	
=	initializsation (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.	
[ ]	delimits list values	init L = []
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 elements to set or list	L’ = cons(X,L)

Symbol	Description	Example
`def=`	indicates beginning of body of a role	
`end role`	indicates end of role	
`function`	data-type for one-way functions	
`hash`	a synonym to `function`	
`i`	intruder's identity	
`in`	check if element is in list or set	`in(X,L)`
`init`	indicates initialization 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}`
`list`	data-type for ordered collection of typed values	
`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[32] 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,na,Na)`
`secret`	used to check secrecy	`secret(Key,A) /\ secret(Key,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	
`text(fresh)`	data-type for nonces	
`transition`	marks beginning of transitions section of basic role	
`witness`	used to check authentication (together with `(w)request`)	`witness(B,A,na,Na)`
`wrequest`	used to check weak authentication (together with `witness`)	`wrequest(A,B,na,Na)`

References

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

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

[3] AVISPA. The AVISPA User Manual. Available at http://www.avispa-project.org, 2006.

[4] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. Technical Report 39, Digital Systems Research Center, february 1989.

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