AVISPA

www.avispa-project.org

IST-2001-39252

Automated Validation of Internet Security Protocols and Applications

Deliverable D2.1: The High Level Protocol Specification Language

Abstract

In this deliverable we provide the syntax and semantics of the High Level Protocol Specification Language (HLPSL) that we employ for specifying protocols and their properties with the AVISPA toolset.

Deliverable details

Deliverable version: *v1.0*
Date of delivery: *31.08.2003*
Classification: *public*

Person-months required: *20*
Due on: *31.08.2003*
Total pages: *43*

Project details

Start date: *January 1st, 2003*
Duration: *30 months*
Project Coordinator: *Alessandro Armando*
Partners: *Università di Genova, INRIA Lorraine, ETH Zürich, Siemens AG*

Information Society Technologies

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

1 Introduction 1

The High Level Protocol Specification Language (HLPSL) is an expressive language for modelling communication and security protocols. HLPSL draws its semantic roots from Lamport's Temporal Logic of Actions (TLA, [6]). TLA is an elegant and powerful language which lends itself well to specifying concurrent systems (see, e.g., [7]) precisely like the types of protocols we seek to model here. Syntactically, however, specifying protocols in a raw logic can be a daunting task. Moreover, the domain of protocol analysis calls for several syntactic constructs (such as message structure) and semantic concepts (like the notion of an intruder) that are problem-independent and arise in every model. Ideally, it would be convenient to model protocols in a language which offers such commonalities built-in. The development of HLPSL was thus undertaken with the following design objectives:

- It must provide a convenient, human readable, and easy to use language yet be powerful enough to support the specification of modern Internet protocols. To this end, HLPSL has been defined in such a way as to closely resemble a language for defining guarded transitions within a state-transition system and is equipped with constructs which allow the modular specification of protocols.
- It must enjoy a formal semantics. To this end, HLPSL has been based on Lamport's TLA and its semantics is given by a translation to a subset of TLA (see Section 7).
- It must be amenable to automated formal analysis. This is achieved by a translation of HLPSL into the Intermediate Format (IF, [2]).

1.1 Architecture

HLPSL is the language through which end users and protocol modellers make use of the AVISPA tool-set. As such, it is designed to be accessible: it should be easy for human users to both read and write HLPSL specifications. To this end, HLPSL provides a high level of abstraction and has many features that are common to most protocol specifications – such as intruder models and encryption primitives – built in. In contrast, the Intermediate Format (IF) – the language into which HLPSL specifications are translated (see Figure 1) – is a lower-level language at an accordingly lower abstraction level.

HLPSL specifications are translated into the IF by the HLPSL2IF translator. These translations, in turn, serve as input to the three analysis tools of the AVISPA tool-set (though, as shown, some additional translation to tool-specific internal formats may be required).

1.2 A Brief Overview of HLPSL Specifications

Protocol specifications in HLPSL are divided into *roles*. Some roles (the so-called *basic* roles) serve to describe the actions of one single agent in a run of a protocol or sub-protocol. Others (*composed* roles) instantiate these basic roles to model an entire protocol

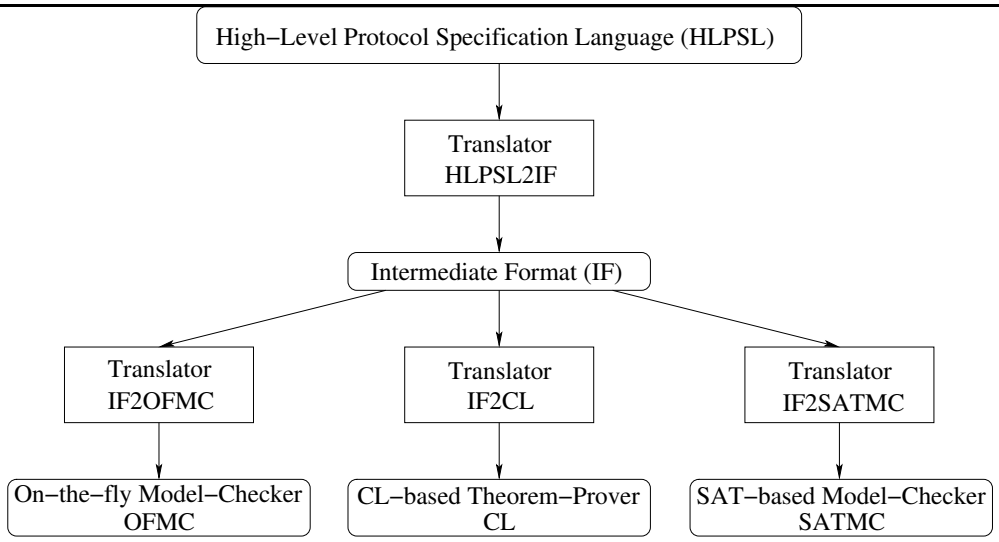

Figure 1: Architecture of the AVISPA tool

run (potentially consisting of the execution of multiple sub-protocols), a session of the protocol between multiple agents, or the protocol model itself. This latter role is often called the `Environment` role.[1]

Given a set of roles describing the protocol and an `Environment` role in which we define the concrete sessions whose execution we wish to consider, we then define our security goals. Currently, HLPSL supports only authentication and secrecy goals, but more general security objectives, specified in an LTL-derived syntax (see, e.g., [9]), are planned for future versions.

1.3 Organisation of this Document

This document is intended to provide the formal syntax and semantics of the HLPSL language while also serving as a guide for protocol modellers. Section 2 gives a brief introduction to several of the basic TLA concepts required for the HLPSL user. In Section 3, we specify how data is defined in HLPSL. In Section 4, we examine roles and describe how to define and instantiate them. Section 5 shows how to specify security goals in HLPSL. Section 6 gives evidence of the adequacy of the language by illustrating the HLPSL specification of some involved examples (whose IF translations can be found in Deliverable 2.3 [2]). The formal semantics of the HLPSL language is given in Section 7. The conclusion in Section 8 describes our experience to date with HLPSL and discusses some improvements currently in progress. Finally, the complete BNF grammar of HLPSL is included in Appendix A.

[1] As a notational convention, text intended to be read as coming directly from a HLPSL specification will appear in `typewriter` font.

2 Introductory Concepts

HLPSL is based on temporal logic. As such, we generally model protocols by describing the system *state* and then specifying the ways in which that state may change. In the case of HLPSL, the state of a protocol specification is defined by an assignment of values to all the system variables. The description of the state changes is then given by specifying so-called *transition predicates* which relate the values of variables in one state (intuitively, the current one) and another (the future, or next state). We refer to the variables in the next state as *primed* variables: for a variable X, X refers to its value in the current state and X' to its value in the next.

In HLPSL, our system specification is divided into roles. For a given role, we define its set of *state variables* as all those variables that are in scope within the role (that is, those that are visible from within the role). Note that this does not include primed variables. Intuitively, this makes sense, as the current state of a role instance should not depend on the *future* values of its variables. A role's *state* is then defined as the values of all of its state variables.

Given this notion of state, we now define a *state predicate* as a first-order formula on a role's state variables and constants. Examples of valid state predicates thus include $X = 5$ and $State = done$. A *transition predicate* is similar but may include primed variables, and so a formula like $X = 5 \wedge Y' = 7$, while not being a state predicate, is a legal transition predicate.

As we have seen, state predicates model the state of the system (or of a given role) at a single point in time. Transition predicates, conversely, are meant to model the evolution in system state that occurs when a transition is taken from one state to another. Of course, a transition might not change anything at all: we call a transition in which all state variables are left unchanged a *stuttering step.* Accordingly, a *non-stuttering step* is one in which the value of at least one state variable changes, and to assert that a transition predicate is non-stuttering is tantamount to requiring that the predicate hold only for those transitions that are non-stuttering.

Finally, we define the concepts of *actions* and *events*. If v is the set of all state variables and v' the set of all primed state variables, then we define the set of *actions* as those transition predicates $p(v, v')$ with the property that $\forall v \ : \ \exists v' \ : \ p(v, v')$. Actions may therefore include stuttering steps. *Events* are transition predicates containing at least one conjunct of the form $X' \neq X$. This definition ensures that events are non-stuttering actions so long as the domain of X contains at least two elements (otherwise, they are of course unsatisfiable). In the context of the transition system, actions are seen as the labels of transitions.

3 Variables and Constants

In HLPSL, we generally adopt the convention that variable identifiers begin with a capital letter, while constant names begin with a lower case letter. HLPSL is a typed language:

each variable and constant must have a unique type, though the types of constants are generally not specified explicitly, but rather inferred from their use. Using HLPSL's base types (described below), we may construct aggregate types such as tuples, lists, and sets. We distinguish between two kinds of function types: mappings from one type to another and one-way functions on messages. Certain types (currently only `text` and `channel`) may have attributes, enclosed within parentheses. These attributes specify additional properties that the elements of the type may take on. For instance, nonces are of type `text` but must have the additional property of uniqueness, which we call *freshness*. Therefore, variables supposed to represent nonces are of type `text (fresh)`.

Here we summarise the base types available in HLPSL

- `agent`: Values of type `agent` represent principal names. The intruder is always assumed to have the special identifier `i`.
- `public_key`: These values represent agents' public keys for asymmetric cryptography. For a given public (respectively private) key `pk`, its inverse private (respectively public) key is obtained by `inv(pk)`.
- `symmetric_key`: Variables of this type represent keys for symmetric encryption.
- `text`: As described above, `text` values are often used as nonces. When used for this purpose, we usually give the attribute `(fresh)` to indicate that subsequent values should be unique and, like any other message, unguessable by the intruder. More precisely, if, in a transition, the primed variable `Na'` appears (and `Na` is of type `text (fresh)`), then `Na'` denotes, as usual, the value of `Na` at the end of the transition, and this value will be a fresh value that the intruder cannot guess.
- `nat`: The `nat` type represents the natural numbers in non-message contexts. It is useful, for instance, for tests of numeric inequality like $X < Y$. We also use `nat` to model countably infinite sets, which we generally represent via functions from `nat` to our desired type. For example, if we wish to represent an infinite number of protocol sessions between two agents, we might declare a mapping `instances` of type `nat -> (agent, agent)`.
- `function`: The base type `function` represents functions on the space of messages. Such functions are useful for modelling, for instance, cryptographic hash functions[2] and key tables. We assume that the intruder cannot invert such functions (in essence, that they are one-way), so we enforce the restriction that, for a function `H`, the intruder can only compute the `function` application term `H(X1,...,Xn)` if he can synthesise all arguments `X1,...,Xn`.
- `bool`: Boolean values are useful for modelling, for instance, binary flags.

[2]For convenience, hash functions may be declared to be of type `hash`, which is synonymous with `function`.

3.1 Aggregate Types

There are currently three methods for type aggregation within HLPSL: tuples, lists (ordered collections), and sets (unordered collections). Tuples are declared in parentheses, while lists and sets share the overloaded square bracket constructor `[]`. Variables that represent a set or list of elements of type `t` are declared as being of type `t set` or `t list`, respectively. The type of an n-element tuple variable is given as the types of the individual elements as follows: $(type_1, type_2, ..., type_n)$. The individual elements of a list, tuple, or set are specified in a comma-separated list.

For instance, we might represent the association of agents with their respective public keys as an initially empty list of pairs as follows:[3]

```
exists KeyMap: (agent, public_key) list
init KeyMap = []
```

Lists and sets share an overloaded insertion operator `cons`. To add the association of an agent `A` and her[4] public key `Ka` to the list we have just declared, we would then write:

```
KeyMap' = cons((A,Ka), KeyMap)
```

Since the set and list operator `cons` is overloaded, if we later decided that we wanted to change `KeyMap` to a set, we would merely have to change the type in the declaration to `(agent, public_key) set`.

Lists and sets also share an overloaded operator `in` for tests of inclusion. For instance, to test if our list `KeyMap` above contains an entry for agent `B`, we could write `in((B,Kb'), KeyMap)`. If an entry exists, this statement will evaluate to true and in the subsequent state, the variable `Kb` will be bound to the value that is associated with agent `B` in `KeyMap`.

3.2 Mappings

HLPSL includes two notions of functions: functions on messages and mappings from one HLPSL type to another. The former are rigid functions: their values do not change, and thus for any rigid function f, at all points in time it holds that $f(X) = f'(X)$. The latter, in contrast, are flexible functions. Their values may change over time, and thus they are useful to model things like key tables which might be initially empty but fill up as the agent learns new public keys.

Mappings are functions relating a domain type and a range type. They are specified using the `->` operator. Syntactically, we do not distinguish between partial and total mappings. Mappings serve several important purposes in HLPSL. As mentioned above,

[3] In HLPSL, `exists` allows us to declare new local variables, and after the `init` keyword we define their initial values. These will be discussed in more detail in Section 4.1.

[4] By convention, we will assume that agents named A or Alice are female and those named B or Bob are male. We refer to the Intruder as "he".

we can use mappings with domain `nat` to represent (countably) infinite sets. We can also conveniently implement per-agent variables using mappings. Continuing our example from the last section, yet another option to model the association of agents to public keys would be with a mapping as shown here:

```
exists KeyMap: agent -> public_key
```

The value of a mapping is specified as a list of pairs in which the first component is of the domain type and the second is of the range type. If we later wanted to specify the value of `KeyMap`, we might write something like

```
init KeyMap = [(A,Ka),(B,Kb)]
```

Having done this, we can retrieve the public key of agent `A` by simple function application: `KeyMap(A)`. Note that the values to which mappings are initialised should be functions. If they are not, the first pair determines the value used: for instance, if `KeyMap` was initialised to `[(A,Ka1),(A,Ka2)]` above, the value of `KeyMap(A)` after initialisation would be `Ka1`.

After initialisation, we use the standard primed-variable notation to update mappings. If, for example, an agent learns later that agent `C` has public key `Kc`, we could add this to the key-map with the statement

```
KeyMap'(C) = Kc
```

This creates a new entry for `C` in the `KeyMap` if none previously existed, and overwrites the previous value of `KeyMap(C)` if it did.

3.3 Messages

We define the space of legal messages as the closure of the basic types described above under the operations of concatenation (via the associative "." operator) and encryption (which we write {`Msg`}`Key` for a given message `Msg` and encryption key `Key`). Notationally, we do not distinguish between symmetric and asymmetric encryption.

Thus, if we have an agent name `A` of type `agent`, a nonce `Na` of type `text (fresh)`, and a symmetric key `K`, all of the following are valid messages:

```
Na          % The nonce on its own
A.Na        % A's name concatenated with the nonce
{A.Na}K     % As above, but encrypted with K
```

Send and receive actions need not specify the type or structure of the message they are going to receive. This adds a great deal of flexibility for modelling cases in which agents may not be able to analyse the messages they receive, for example because they cannot decrypt them.

As an example, consider the case in which an agent expects to receive the latter message above, {`A.Na`}`K`, but does not possess the key `K`. Here, we can simply write that the agent

expects to receive `X`, and `X` will get assigned the entire value `{A.Na}K` without being further analysed. This is useful, as situations like this arise often, for instance in protocols in which an agent receives information she cannot decrypt and then forwards it to another agent who can.

3.4 Channels

Channels are variables over which communication takes place. A channel connects communicating parties. The `channel` type may take an attribute which specifies the intruder model to be used for communication over the channel. Currently, the only supported intruder model is `(dy)`, the Dolev-Yao model, in which the intruder, in addition to having all the capabilities of an honest agent, may divert sent messages and send new ones impersonating other agents.

In the standard Dolev-Yao model, the intruder is so powerful that we need not actually differentiate the intruder and the network as being distinct. We therefore adopt the view that the intruder *is* the network, and `(dy)` channels are always viewed as connecting an agent with the intruder (though the intended recipient may of course be another agent). Thus, all messages sent by honest agents go to the intruder, and all messages received by honest agents come from the intruder.

Further intruder models are envisioned for the future of the project: over-the-air communication, which models a channel (such as Bluetooth or 802.11) on which the intruder may hear all traffic and send messages, but not prevent messages from reaching their destination; listen-only channels, upon which the intruder may listen to messages but not divert or send them; and secure channels to which the intruder has no access.

4 Roles

A role in HLPSL can be considered as a description of behaviour. Roles may be parametrised, taking one or more arguments of arbitrary types[5], and may also declare local variables via the `exists` operator[6]. We distinguish between two different types of roles: basic and composed. *Basic roles* generally describe the actions involved in a single protocol or subprotocol run. *Composed roles*, on the other hand, consist of *instantiations* (sometimes called *compositions*) of one or more other roles. In this way, we can model situations like an agent who executes all the sub-protocols of a particular protocol in order (sequential composition), an agent who is involved in several protocol sessions at once (parallel composition), or higher-level roles like the `Environment` role, which we will see later.

In the "big picture" of HLPSL's TLA-based semantics, a basic role should be seen as the analogue to a set of TLA formulae describing an initial predicate and a next-state relation for a single system component. Composed roles then translate to formulae that

[5]Note that there is currently no notion of "higher-order" roles. That is, roles may not take other roles as arguments, though they may of course instantiate other roles in the `composition` section.

[6]For convenience, the `local` keyword is also available as a synonym for `exists`.

describe how the individual system components interact: for instance, requiring that the next-state relation of the whole system satisfy the conjunction of the next-state relations of all components.

The definition of a role generally consists of the following elements:

1. the role declaration: its name and the list of formal arguments, along with (in the case of basic roles) a player declaration;
2. declaration of local variables and ownership rules, if any;
3. initialisation of variables, if required;
4. declaration of accepting states, if any;
5. knowledge declarations, if applicable; and
6. (optionally) either a transition section (for basic roles) or a composition section (for composed roles).

4.1 Role Declaration

Roles are declared using the keyword `role` followed by an identifier (the role name) and, in parentheses, a comma separated list of arguments, along with their types. A basic role must also define its player (see Section 4.2.1). The role body then follows, starting with the `def=` keyword and ending with `end role`. As described above, roles may also include one or more of the optional "role headers": a role may declare local variables using `exists`, may assert ownership of variables with `owns`, may initialise variables in an `init` section, may define the set of its accepting states using `accepts`, and may also include one or more knowledge declarations describing the required knowledge of agents involved in the execution of the role. Initialisation and acceptance declarations may take arbitrary state predicates to describe the initial and the accepting states respectively.

Before describing these in more detail, let us take a simple example. The following role, which we will call `trivial`, takes an agent name `A` and an asymmetric public key `Ka` as arguments. The `played_by A` declaration indicates that the agent named in variable `A` will play the role. We postpone further discussion of players to Section 4.2.1. The `trivial` role also declares local variables `State` and `Na`. Finally, it initialises `State` to 0. Note that comments in HLPSL begin with the `%` symbol and continue until the end of the line.

```
% Definition of an example role
role trivial(A:agent, Ka:public_key) played_by A def=
  exists State:nat,  Na:text (fresh)
  init State = 0
end role
```

This example also demonstrates the usage of attributed types: here we declare that `Na` is fresh, indicating that it is probably to be used as a nonce.

We should note that, although we do not distinguish between them syntactically, there are in fact two notions of role parameters. Certain parameters, let us call them *rigid parameters*, will be instantiated with constants when the role is called. These are equivalent to *rigid variables* in TLA, as their values do not subsequently change. In the above example, for instance, the rigid parameter `A` might get instantiated with the constant `a`. This value does not then change.

The other type of parameters, which we call *state variable parameters*, are indeed system state variables that correspond to flexible variables in TLA. Their values may change over time. In the above example, we expect the value of `State` to change. If we were to make it a parameter rather than a local variable, it would be a state variable parameter.

4.1.1 Variable Initialisation

The `init` section specifies a state predicate which describes the set of allowed initial states. In the above example, we restrict the set of permissible initial states to only those in which the variable `State` is equal to 0. The other variables, `A`, `Ka`, and `Na`, may have arbitrary values initially.

More complicated scenarios can arise when we want to initialise, for example, a mapping. The following example shows how we can implement key-rings as a mapping from agents to a set of public keys, then initialise each keyring to the empty set. Given a set `Agents` of agents, we then write:

```
exists Kr: agent -> public_key set
init
  /\_{in(A,Agents)} Kr(A)=[]
```

Here, we initialise over a set of agents. The `/\_` operator expresses conjunction over all elements in a set.

4.1.2 Variable Ownership

In an `owns` declaration, a role can optionally list those variables of which it is the owner. Ownership asserts that a (potentially shared) variable may only be written in the ways that the owning role specifies. To illustrate, assume that variable `X` is owned by role `A`. If `X` changes value in a particular step, then if `A` is a basic role, it must be one of the transitions of `A` that modified the value of `X`. If, however, `A` is a composed role, then one of the roles instantiated by `A` must have modified `X`'s value. Note that read access to owned variables is not restricted: any role to which the variable is visible may read its value as usual.

Although many roles may potentially be able to write to an owned variable, only one role may actually own it. Consider, for example, a composed role `Many_As` which instantiates the role `A` twice:

```
role A(Id:nat, X:text) played_by Id def= ... end role
role Many_As(X:text) def=
  owns X
  composition
    A(1,X) /\ A(2,X)
end role
```

In this case, the role `A` may *write* `X`, but may not *own* it. From a TLA standpoint, we have two formulae – namely, `A(1,X)` and `A(2,X)` – specifying ways in which `X` may change. And the fact that the role `Many_As` *owns* the variable `X` asserts that `X` may change in *no other* ways than those that satisfy the conjunction of these two formulae.

4.1.3 Acceptance

Roles may describe, via a state predicate, the set of so-called *accepting* states. The notion of acceptance is generally used to indicate "successful" completion. This is useful for sequential composition: if we have two sequentially composed instantiations `A();B()`, then how do we know when `A` is finished and `B` should begin? A set of transitions in a role carries no information about the order in which they will be executed, so we cannot say something like "`B` begins after the last transition in role `A` has fired."

Instead, we define accepting states and say that `B` starts after `A` has accepted.

For a more concrete example, consider the Internet Key Exchange Protocol (IKE, [5]). In IKE phase 1, the parties establish an authenticated channel over which they will communicate. Then in phase 2, they proceed to establish Security Associations. One might model IKE in HLPSL as two roles, `Phase1` and `Phase2`. Accordingly, one agent participating in a run of the whole protocol would most likely be modelled as the sequential composition (see Section 4.3) `Phase1 ; Phase2`.[7] However, in order to determine *when* role `Phase2` begins execution, we need a well-defined notion of when role `Phase1` is finished. More specifically, we want a notion of *successful* completion: that is, we only want to proceed with phase 2 if phase 1 successfully established an authenticated channel.

For this, we use the notion of acceptance. We describe, via a state predicate, the role state once the authenticated channel has been established. Assume for the sake of example that the authentication has taken place when role `Phase1`'s state variables `State` and `Auth` take on the values of `5` and `1`, respectively. We then define our acceptance predicate via the `accept` role header as follows:

```
accept State = 5 /\ Auth = 1
```

The sequential composition `Phase1 ; Phase2` can then be explicitly understood as expressing that, whenever the acceptance predicate of role `Phase1` holds, it may stop and role `Phase2` may begin execution.

[7] For simplicity, we ignore role arguments for the moment.

4.1.4 Knowledge Declarations

The optional *knowledge* declarations follow the role body and allow us to specify what initial knowledge is required for participation in a protocol execution. For each agent, we may give a set of terms that the agent must know. These can be arbitrary compound message terms built up of variables and constants that are in scope within the role.

For instance, we might require that each agent in a certain protocol possess his or her own private key and certificate. Assuming we are modelling role `Alice` to be played by agent `A` with public key `Ka`, and further that the certificates are to be signed by a trusted server `S`, we can write this as follows:

```
role Alice(A, S: agent,
           Ka, Ks: public_key) played_by A def=
    ...    % Other role headers
    knowledge(A) = { inv(Ka), {A,Ka}inv(Ks) }
    ...    % Role body
end role
```

A single role may have multiple knowledge declarations, which is why they are parametrised (in the example above, by `A`). This can be useful to model situations in which multiple parties are seen to "play" the role collectively, or, by parametrising a knowledge declaration by the intruder's name `i`, to experiment with scenarios in which certain information is compromised. By adding the following declaration to role `Alice` above, we could test what happens if agent `A` compromises her private key.

```
knowledge(i) = { inv(Ka) }
```

We also use a knowledge declaration (generally in the top-level `Environment` role) to specify the intruder's initial knowledge.

4.2 Basic Roles

We often view the whole system as a transition system using the notions of state, transition, and respective predicates defined above. We view the states as being labelled by an assignment of state variables and the transitions as being labelled by actions.

For a given basic role, the values of its variables make up its state, and the role definition gives the legal transitions. Of course, not all variables are visible to all roles: each role can only access those variables which are passed in to it as arguments and, of course, its own local variables.

A basic role is characterised by the inclusion of a `transition` section in which, for the subset of the state variables that are visible from that particular role, legal transitions are defined. We distinguish between two different types of transitions, respectively called spontaneous actions (denoted by the `--|>` arrow) and immediate reactions (specified using the `=|>` arrow).

4.2.1 Players

Each basic role has an identifier specifying the name of the agent playing the role, for instance agent `A` plays role `Alice`. We call this notion the role's *player* and use the `played_by` declaration to specify the name of the variable which holds the player name. We always assume that the name of the player is passed in via an argument and is not a local variable.

For example, we specify that a role `Bob` (with several arguments) is played by the agent whose name is stored in variable `B`, as follows:

```
role Bob(A, B: agent,
         Ka, Kb: public_key) played_by B def=
         ...
end role
```

The notion of players is reserved for basic roles.

4.2.2 Transitions

Spontaneous actions relate a state predicate on the LHS with an action on the RHS. Intuitively, a spontaneous action transition `A --|> B` expresses that, whenever we are in a state that fulfills state predicate `A`, we *may* make a transition labelled by the action `B` into a new state. Note, however, that we do not require that this transition fire right away. When and if it does fire, we obtain the new state by taking the old state and applying any side effects that `B` might produce; that is, any state variables not affected by `B` remain the same. We call such transitions "spontaneous" because, although they are enabled whenever state predicate `A` is true, they convey nothing about when they must fire.

Immediate reaction transitions, on the other hand, have the form `X =|> Y` and relate an event `X` and an action `Y`. This expresses that, whenever we take a transition that is labelled in such a way as to make the non-stutter event predicate `X` true, then we *must* immediately (more precisely, simultaneously) execute action `Y`.

In addition to communication actions (described in the next section), we allow the following tests in state predicates:

- Tests of equality using the notation `A = B` or inequality using the notation `not(A = B)` or the equivalent shorthand `A /= B`.
- Less than tests using the operator `<` (currently defined only on values of type `nat`).
- Tests of set and list inclusion or exclusion using the notation `in(Element,List)` or `not(in(Element,List))`, respectively.

Both kinds of transitions may be preceded by a label, which may be an alphanumeric string starting with a lower case letter and ending with a period. Examples are shown below:

```
anAction.  X --|> Y
aReaction. Q  =|> R
```

However, these labels carry no information about the order in which the transitions fire. They are merely names.

4.2.3 Communication

Communication in HLPSL is synchronous. We model synchronicity via the immediate reaction transition: given a receive action on the LHS of a `=|>` transition, we are ensured that when the corresponding send action takes place, the receive action will happen simultaneously.

Communication takes place over channels, which are themselves merely variables with values like any other. By convention, we generally assign channels convenient names like `SND` and `RCV` and then write `SND(Msg)` and `RCV(Msg)`. This is, however, merely a shorthand. The former action, assuming that it appears on the RHS of a transition, meaning it is a write action, is a short form for `SND'=Msg`. The latter, assuming that it appears on the LHS of a transition and is therefore a read action, is short for `(RCV-flag'/=RCV-flag)/\(RCV'=Msg)`, where `RCV-flag` is a binary flag which is toggled each time a new message arrives on channel `RCV`. Recalling our restriction that events contain at least one predicate of the form `X'/=X`, we can see that the former is an action and the latter an event.

In Dolev-Yao models, we try to group the receipt of a message and the sending of the corresponding response into a single transition. Such transitions generally take the following form:

```
State = 0 /\ RCV(Request) =|> State' = 2 /\ SND(Reply)
```

This is fine for the Dolev-Yao intruder model. However, we cannot use this in over-the-air (OtA) intruder models. The reason is that, in the DY model, we have synchronous communication between honest agents and the intruder. The intruder himself, however, need not act synchronously. He lies between two communicating agents and introduces a delay between receiving a message and sending the response, because he himself will not react immediately as the agents do in the example transition above.

In the OtA model, however, the intruder may not prevent messages from arriving at their intended destinations. The agents may communicate directly with one another, and we will therefore not have this delay introduced by the intruder. Without it, however, the whole protocol run between honest agents will chain together a series of immediate reactions and will appear to be executed in a single step. The intruder will have no chance to perform any actions to influence the protocol run, he will only see the state before the run begins and then the state after the run is finished. Clearly, we cannot allow this in OtA.

Instead, we must enforce that the agents themselves introduce the required delay. For this, we need the spontaneous action arrow (`--|>`). In an OtA model, we would split our example from above into two transitions as follows:

```
state = 0 /\ RCV(Request) =|>  state' = 1
state = 1                 --|>  state' = 2 /\ SND(Reply)
```

This more general model works for both DY and OtA channels. However, since we will focus on DY models for now, we can safely use the former model until we start using OtA channels later in the project.

Binding and matching of messages: Variables within receive actions can serve two purposes: on the one hand, we want to restrict which messages are accepted (for instance, we accept only those messages encrypted with a certain key), on the other hand, we may want to introduce new bindings from variables to parts of the incoming message. We accomplish the former using unprimed variables and the latter using primed variables. Take, for instance, the receipt of the first message of the Needham-Schroeder Public Key protocol (NSPK, [3, 10]). Within the responder role, we might have a receive action such as the following one:

```
state = 0 /\ RCV({Na'.A}Kb) =|> ...
```

Here, we assume that the responder knows the identity of his communication partner in advance. The presence of `A` and `Kb` restricts the messages that the responder will accept: in particular, we assert here that the message must be a pair, encrypted with `Kb`, and that the second component of the pair must be the current value of variable `A`. The first component, however, may be anything, and in the state resulting from this transition, `Na` will be bound to its value. We specify this by writing the primed variable `Na'` in the receive action.

4.3 Composed Roles

In composed roles, we have no `transition` section. Rather, we have a section entitled `composition` in which we instantiate other roles, both basic and composed. Composition can be sequential (using the `;` operator) or parallel (using the `/\` operator – as in TLA, parallel composition is expressed via conjunction). We may also compose in parallel over a set, using the same notation we have already seen in Section 4.1 for variable initialisation over a set.

An important compositional role is of course the top-level role. The current HLPSL syntax does not allow compositional instantiation outside of a role definition. That is, any composition we require – be it parallel or sequential – must be inside a role. We can then instantiate that role outside of any role definitions, calling the main role. This top-level role is usually called `Environment`.

5 Security Goals

In the future, HLPSL will support very general security goals specified in a restricted temporal logic syntax. At the time of writing, however, only general secrecy and authentication goals are supported. This is, however, sufficient to specify a large number of problems.

Goals are given in their own section, which generally comes at the end of a HLPSL specification. It begins with the keyword `goal` and ends with `end goal`. Between the two, multiple security goals may be listed.

That a certain variable `V` should remain permanently secret is expressed by the goal `secrecy_of V`. Should `V` ever be obtained or derived by the intruder, a security violation will result. We assume that all roles using a variable named `V` intend for their respective copies to be kept secret. In the place of a single variable, we may also give a comma separated list of variables which are permanent secrets.

The default HLPSL authentication goals take two forms, corresponding respectively to Lowe's definitions of strong and weak authentication [8]. We give the names of variables on whose value the two parties authenticate in a comma separated list. In both the authenticating and the authenticated roles, these variables must have the same names.

To express that role `Alice` should authenticate `Bob` on nonce `Nb` but that `Bob` should weakly authenticate `Alice` on nonce `Na`, we would write the following goal section:

```
goal
    Alice authenticates Bob on Nb
    Bob weakly authenticates Alice on Na
end goal
```

6 Some Examples

We now examine several concrete examples. We will look at full HLPSL specifications for a number of variants of the Needham-Schroeder Public Key Protocol (NSPK).

6.1 "Standard" NSPK

We first give the HLPSL specification of the usual three-message variant of NSPK with no key server. Here we see the specification of the initiator role, which we call `Alice`.

```
role Alice (A, B: agent,
            Ka, Kb: public_key,
            SND, RCV: channel (dy)) played_by A def=
  exists State : nat, Na : text (fresh), Nb: text
  init State=0
  knowledge(A) = { inv(Ka) }
  transition
    step1. State=0 /\  RCV(start)
       =|> State'=1 /\ SND({Na'.A}Kb)
    step2. State=1 /\ RCV({Na.Nb'}Ka)
       =|> State'=2 /\ SND({Nb'}Kb)
end role
```

Alice's arguments include her own name and the name of the (intended) responder (`B`), their respective public keys, and two Dolev-Yao channels (`SND` for sending messages and `RCV` for receiving them). Via the `knowledge`, we assert that any agent playing role `Alice` must know the private key corresponding to public key `Ka`.

We can see that the protocol begins upon reception of a `start` message. This is a dummy message sent by the intruder to signal to an initiator that he or she should start a new run of the protocol. Strictly speaking, this is not necessary: we could omit the `RCV(start)` and use the spontaneous action arrow instead. However, including the dummy message allows us to write all of our transitions in the convenient form of paired receives and sends (see Section 4.2.3).

It is important to note that although our transitions are named `step1` and `step2`, these are merely labels. They do nothing to impose an ordering on the two transitions: protocol modellers must do this themselves with the guards that they choose for their transitions.

The role of the responder (called `Bob`) is defined analogously.

```
role Bob(A, B: agent,
         Ka, Kb: public_key,
         SND, RCV: channel (dy)) played_by B def=
  exists State : nat, Na: text, Nb: text (fresh)
  init State=0
  knowledge(B) = { inv(Kb) }
  transition
    step1. State=0 /\ RCV({Na'.A}Kb)
       =|> State'=1 /\ SND({Na'.Nb'}Ka)
    step2. State=1 /\ RCV({Nb}Ka)
       =|> State'=2
end role
```

We now define a composed role called `NSPK`, which takes as an argument a specification of the valid sessions and starts them all in parallel. An individual session has the form `(A,B,Ka,Kb)`, indicating that agent `A` will serve as initiator with public key `Ka` and agent `B` will serve as responder with key `Kb`. For each session, it in turn instantiates one `Alice` role and one `Bob` role in parallel.

```
role NSPK(S, R: agent -> channel (dy),
          Instances: (agent,agent, public_key,public_key) set) def=
  exists A, B: agent, Ka, Kb: public_key
  composition
     /\_{in((A,B,Ka,Kb),Instances)}
         Alice(A,B,Ka,Kb,S(A),R(A))
      /\ Bob(A,B,Ka,Kb,S(B),R(B))
end role
```

We now define and our top-level role, which we call `Environment`. The top-level role may be given any name – it need not necessarily be called `Environment` – but it is important to note that it may not take any arguments.

```
role Environment() def=
    composition
    NSPK([(a,s_a),(b,s_b)],                    % S
         [(a,r_a),(b,r_b)],                    % R
         [(a,b,ka,kb),(a,i,ka,ki)])            % Instances
end role
```

In the third argument to the `NSPK` role, we specify two session instances: one in which agent `a` communicates with agent `b`, and one in which agent `a` communicates with the intruder `i`.

Here we see constants, such as `a` and `s_a`, that we have not declared anywhere. When the compiler encounters such constants, they are considered to be global constants of the appropriate types.

We also specify three security goals: authentication of both parties and secrecy of the two nonces `Na` and `Nb`.

```
goal     Alice weakly authenticates Bob on Nb
         Bob weakly authenticates Alice on Na
         secrecy_of Na, Nb
end goal
```

Finally, we instantiate our main `Environment` role. All HLPSL specifications end with the instantiation of this main role. This is written simply on its own line and does not require a `composition` section as follows:

```
Environment()
```

6.2 NSPK with replay-protection

We now model a modified version of the Needham-Schroeder Public Key protocol (still with no key server). Our modification will be to require that each agent maintain a set of the nonces he or she has seen from previous protocol runs. This will require declaration and initialisation of the nonce sets as well as changes to the rules so that agents remember the nonces they have seen.

Each agent will have an additional variable `L` which stores the set of nonces. At instantiation time, we will use a mapping from agents to text sets called `noncestore` to handle the sets in a flexible way.

```
role Alice (A, B: agent,
            Ka, Kb: public_key,
            L: text set,
            SND, RCV: channel (dy)) played_by A def=
  exists State : nat, Na : text(fresh), Nb: text
  init State=0
  transition
    step1. State=0 /\  RCV(start)
       =|> State'=1 /\ L'=cons(Na',L) /\ SND({Na'.A}Kb)
    step2. State=1 /\ RCV({Na.Nb'}Ka) /\ not(in(Nb',L))
       =|> State'=2 /\ SND({Nb'}Kb) /\ L'=cons(Nb',L)
end role

role Bob(A, B: agent,
         Ka, Kb: public_key,
         L: text set,
         SND,RCV: channel (dy)) played_by B def=
  exists State : nat, Na: text, Nb: text (fresh)
  init State=0
  transition
    step1. State=0 /\ RCV({Na'.A}Kb) /\ not(in(Na',L))
        => State'=1 /\ L'=cons(Nb',cons(Na',L)) /\ SND({Na'.Nb'}Ka)
    step2. State=1 /\ RCV({Nb}Ka)
        => State'=2
end role

role NSPK(S, R: agent -> channel (dy),
          Instances: (agent,agent, public_key,public_key) set,
          Agents: agent set)
def=
  exists A, B: agent, Ka, Kb: public_key,
         L: agent -> text set
  init         /\_{in(A,Agents)} L(A)=[]
  composition
     /\_{in((A,B,Ka,Kb),Instances)}
         Alice(A,B,Ka,Kb,L(A),S(A),R(A))
      /\ Bob(A,B,Ka,Kb,L(B),S(B),R(B))
end role

role Environment() def=
    composition
    NSPK([(a,s_a),(b,s_b)],                    % S
         [(a,r_a),(b,r_b)],                    % R
```

```
          [(a,b,ka,kb),(a,i,ka,ki)],          % Instances
          [a,b,i])                            % Agents
end role
goal      Alice weakly authenticates Bob on Nb
          Bob weakly authenticates Alice on Na
          secrecy_of Na, Nb
end goal
Environment()
```

6.3 NSPK with key-server

We model the key-server as an agent called `server` who replies to every request with a signed certificate. The individual agents are augmented with the additional variable `KeyRing` which stores the public keys the agent already knows. Each agent's transitions must also account for the new possibility that he does not know the public key of his communication partner, in which case he must request it from the server. We assume, however, that each agent knows his own public key and that of the key server.

```
role Alice (A, B: agent,
            Ka, Ks: public_key,
            KeyRing: (agent, public_key) set,
            SND, RCV: channel (dy)) played_by A def=
  exists State : nat, Na : text(fresh),
         Nb: text, Kb: public_key
  init State=0
  transition
    % Start of the protocol, provided Alice already knows
    % Bob's public key
    step1a. State=0 /\  RCV(start) /\ in((B,Kb'), KeyRing)
        =|> State'=2 /\ SND({Na'.A}Kb')

    % Start of the protocol if Alice must request Bob's
    % public key from the server.
    step1b. State=0 /\  RCV(start) /\ not(in((B,Kb'), KeyRing))
        =|> State'=1 /\ SND(A.B)

    % Receipt of response from key-server
    step2. State = 1 /\ RCV({B.Kb'}inv(Ks))
       =|> KeyRing' = cons((B,Kb'), KeyRing) /\ SND({Na'.A}Kb')
        /\ State' = 2

    % Receiving the second message of the protocol and sending
    % the third
```

```
      step3. State=2 /\ RCV({Na.Nb'}Ka)
         =|> State'=3 /\ SND({Nb'}Kb)
  end role

  role Bob(A, B: agent,
           Ka, Ks: public_key,
           KeyRing: (agent, public_key) set,
           SND, RCV: channel (dy)) played_by B def=
    exists State : nat, Na: text, Nb: text (fresh),
           Ka: public_key
    init State=0
    transition
      % Normal start of the protocol, if Bob knows Alice's
      % public key
      step1a. State=0 /\ RCV({Na'.A}Kb) /\ in((A,Ka'), KeyRing)
          =|> State'=2 /\ SND({Na'.Nb'}Ka')

      % Start of the protocol if Bob must request Alice's public
      % key from the key-server.
      step1b. State=0 /\ RCV({Na'.A}Kb) /\ not(in((A,Ka'), KeyRing))
         =|> State'=1 /\ SND(B.A)

      % Receipt of response from key-server
      step2. State = 1 /\ RCV({A.Ka'}inv(Ks))
         =|> KeyRing' = cons((A,Ka'), KeyRing) /\ SND({Na.Nb'}Ka')
          /\ State' = 2

      % Last step of the protocol
      step2. State=2 /\ RCV({Nb}Ka)
          => State'=3
  end role

  % The key server
  role server(Ks: public_key,
              KeyMap: (agent,public_key) set,
              SND,RCV: channel (dy)) def=
    exists A:agent, B:agent, Kb: public_key
    transition
    step0. RCV(A'.B') /\ in((B,Kb'), KeyMap)
       =|> SND({B'.Kb'}inv(Ks))
  end role

  role NSPK(S,R: agent -> channel (dy),
```

```
           Ks: public_key,
           Instances: (agent,agent, public_key,public_key) set,
           Agents: agent set)
def=
  exists A, B: agent, Ka, Kb: public_key,
         KeySet: agent -> (agent, public_key) set
  init          /\_{in(A,Agents)} KeySet(A)=[]
  composition
     /\_{in((A,B,Ka,Kb),Instances)}
         Alice(A,B,Ka,Ks,KeySet(A),S(A),R(A))
      /\ Bob(A,B,Ka,Ks,KeySet(B),S(B),R(B))
end role

role Environment() def=
    composition
    NSPK([(a,s_a),(b,s_b)],                    % S
         [(a,r_a),(b,r_b)],                    % R
         ks,
         [(a,b,ka,kb),(a,i,ka,ki)],            % Instances
         [a,b,i])                              % Agents
 /\ server(ks, [(a,ka),(b,kb),(i,ki)], snd_srv, rcv_srv)
end role
goal     Alice weakly authenticates Bob on Nb
         Bob weakly authenticates Alice on Na
         secrecy_of Na, Nb
end goal
Environment()
```

7 Semantics

7.1 HLPSL and TLA

In this section we describe how HLPSL specifications can be mapped into TLA by means of a sample translation from the former to the latter and illustrate how this translation works via a running example. Since TLA enjoys a formal semantics, this translation provides (albeit indirectly) a formal semantics to HLPSL.

In general, we can translate any HLPSL specification to an equivalent one in TLA as we will do here with a simple example. Notions such as acceptance and sequential composition are not explicitly addressed by this example but are still easy to translate. For simplicity, this example also ignores the notion of session repetition. In HLPSL, the semantics of a role instantiation includes the notion that once a protocol run is finished, the instance can start a new protocol run at any time. This, however, also has a straightforward translation

into TLA.

7.2 "Under the Hood" of HLPSL

HLPSL provides convenient built-in notions of messages and different intruder models. To translate a HLPSL specification into TLA, however, we must make explicit the semantics of all features of the language, including those that are generally hidden from the end user when modelling a protocol.

7.2.1 Messages

We begin by specifying operations on messages, such as concatenation and encryption, and their properties. In HLPSL, we do not distinguish between symmetric and asymmetric encryption. More precisely, the distinction between the two is not made syntactically, but rather on the basis of the key type. In our translation to TLA, however, we will explicitly differentiate between the two. Given the sets *Agent* of agent names, *Nonces* of valid nonces, and *Msg* of all messages, we therefore need three functions, call them *Pair*, *ACrypt*, and *SCrypt* for pairing, asymmetric encryption, and symmetric encryption, respectively.

We can now specify the properties of our operations. Pairing (expressed in HLPSL via the "." operator) is associative. We enforce this via the following formula:

$$\begin{aligned} PairAssoc \;\triangleq\; & \forall\, m1, m2, m3 \in Msg \;: \\ & Pair(Pair(m1, m2), m3) = Pair(m1, Pair(m2, m3)) \end{aligned}$$

We must also explicitly formalise our assumptions regarding cryptography. More specifically, we assume that the intruder cannot generate a ciphertext $\{M\}_K$ unless he already knows that term or knows both M and K. Said another way, we assume that no two ciphertexts $\{M_1\}_{K_1}$ and $\{M_2\}_{K_2}$ are equal unless $M_1 = M_2$ and $K_1 = K_2$. We assert this via the formula:

$$\begin{aligned} UniqueCrypto \;\triangleq\; & \wedge \forall\, k1, m1, k2, m2 \in Msg \;: \\ & \quad ACrypt(k1, m1) = ACrypt(k2, m2) \Rightarrow \wedge\; k1 = k2 \\ & \qquad\qquad\qquad\qquad\qquad\qquad \wedge\; m1 = m2 \\ & \wedge \forall\, k1, m1, k2, m2 \in Msg \;: \\ & \quad SCrypt(k1, m1) = SCrypt(k2, m2) \Rightarrow \wedge\; k1 = k2 \\ & \qquad\qquad\qquad\qquad\qquad\qquad \wedge\; m1 = m2 \end{aligned}$$

This alone, of course, is not sufficient to enforce the restriction that the intruder may only decrypt those cryptograms for which he possesses the appropriate decryption key. At this stage, however, we concern ourselves only with the structure of messages and the operations on them. We address the intruder in the next subsection.

For public keys, we require that the inverse function (written `inv` in HLPSL) is bijective:

$$\begin{aligned} InvBij \;\triangleq\; & \wedge\, \forall\, pk2 \;:\; \exists\, pk1 \;:\; inv(pk1) = pk2 \\ & \wedge\, \forall\, pk1, pk2 \;:\; inv(pk1) = inv(pk2) \Rightarrow pk1 = pk2 \end{aligned}$$

Bijectivity of inverse then implies the desirable property that $\forall\, pk \,:\, inv(inv(pk)) = pk$.

Finally, we need a means of enforcing freshness of nonces. For this, we simply maintain a set of those nonces that have already been used and say that $Fresh(n)$ holds if n is not a used nonce.

$$Fresh(n) \;\triangleq\; n \notin UsedNonces$$

7.3 The Intruder

The simple declaration of a `channel` with the attribute `(dy)` succinctly specifies that this channel can be assumed to be under control of the well-known Dolev-Yao intruder. We must now formalise the capabilities of this intruder.

The intruder has three analysis rules: he can decompose a pair into its components, or he can decrypt encrypted terms if he possesses the appropriate key. This latter analysis possibility is divided into two separate rules for symmetric and asymmetric encryption. If we assume that the current knowledge of the intruder is stored in a variable IK, these three analysis rules can be specified as follows:

Split a pair into its components:

$$\begin{aligned} ASplit \;\triangleq\; \exists\, m1, m2 \in Msg \,:\, & \wedge\, Pair(m1, m2) \in IK \\ & \wedge\, IK' = IK \cup \{m1, m2\} \end{aligned}$$

Decrypt messages encrypted with the intruder's public key:

$$\begin{aligned} AAdec \;\triangleq\; \exists\, k, m \in Msg \,:\, & \wedge\, ACrypt(ki, m) \in IK \\ & \wedge\, inv(ki) \in IK \\ & \wedge\, IK' = IK \cup \{m\} \end{aligned}$$

Decrypt symmetric messages if the intruder has the appropriate key:

$$\begin{aligned} ASdec \;\triangleq\; \exists\, k, m \in Msg \,:\, & \wedge\, SCrypt(k, m) \in IK \\ & \wedge\, k \in IK \\ & \wedge\, IK' = IK \cup \{m\} \end{aligned}$$

In addition, the intruder may of course generate new composed terms based on those messages he already possesses. Here, we have four rules: the intruder may generate a new pair, may encrypt a message using either a symmetric or an asymmetric key, or may apply a function to a term he already knows. Note that this latter generation rule has no counterpart among the analysis rules, because we assume that the intruder cannot invert functions.

Generation of a pair:

$$GPair \triangleq \exists\, m1, m2 \in Msg.\ :\ \wedge\ m1 \in IK \\ \wedge\ m2 \in IK \\ \wedge\ IK' = IK \cup \{Pair(m1, m2)\}$$

Asymmetric encryption:

$$GAcrypt \triangleq \exists\, k, m \in Msg\ :\ \wedge\ k \in IK \\ \wedge\ m \in IK \\ \wedge\ IK' = IK \cup \{ACrypt(k, m)\}$$

Symmetric encryption:

$$GScrypt \triangleq \exists\, k, m \in Msg\ :\ \wedge\ k \in IK \\ \wedge\ m \in IK \\ \wedge\ IK' = IK \cup \{SCrypt(k, m)\}$$

Application of a function:

$$GApply \triangleq \exists\, f, m \in Msg\ :\ \wedge\ f \in IK \\ \wedge\ m \in IK \\ \wedge\ IK' = IK \cup \{f(m)\}$$

Finally, the intruder may also generate fresh data not based on any previous knowledge. We model this via the *GFresh* transition predicate below:

$$GFresh \triangleq \exists\, x\ :\ \wedge\ x' \in Msg \\ \wedge\ Fresh(x') \\ \wedge\ IK' = IK \cup \{x'\}$$

Like Gray and McLean in [4], we group these actions together into a single step called *Manipulate*, which represents the intruder performing a single analysis or generation step in order to augment his knowledge.

$$Manipulate \triangleq \vee ASplit \vee AAdec \vee ASdec \\ \vee GPair \vee GAcrypt \vee GScrypt \vee GApply \vee GFresh$$

The intruder may, of course, also introduce messages into the network and divert messages, preventing them from reaching their intended destinations. We model these two intruder actions with parametrised next-state predicates, each of which takes a channel as an argument as well as a flag which is toggled to indicate that a new message has been sent on that channel. As per HLPSL convention, we call the operation by which the intruder sends messages *Impersonate*, though he also sends messages under his own name via this action. We also define a helper transition predicate $toggle(x)$ that flips a one-bit variable.

Toggle flips a single bit:

$$\begin{array}{lll} \mathit{toggle}(\mathit{flag}) & \triangleq & \vee \wedge \mathit{flag} = 0 \\ & & \quad \wedge \mathit{flag}' = 1 \\ & & \vee \wedge \mathit{flag} = 1 \\ & & \quad \wedge \mathit{flag}' = 0 \end{array}$$

The intruder sending a messages on channel *chan*:

$$\begin{array}{lll} \mathit{Impersonate}(\mathit{chan}, \mathit{SND\_flag}) & \triangleq & \wedge \exists\, m \in \mathit{IK} \,:\, \mathit{chan}' = m \\ & & \wedge\ \mathit{toggle}(\mathit{SND\_flag}) \end{array}$$

The intruder intercepting a message on channel *chan*:

$$\begin{array}{lll} \mathit{Divert}(\mathit{chan}, \mathit{RCV\_flag}) & \triangleq & \wedge\ \mathit{RCV\_flag}' \neq \mathit{RCV\_flag} \\ & & \wedge\ \mathit{IK}' = \mathit{IK} \cup \mathit{chan}' \end{array}$$

7.4 An Example Translation

For our example, we will translate a HLPSL specification of the SHARE protocol (see [1]), a two-message key establishment protocol. In Alice and Bob notation, SHARE looks like this:

$$\begin{array}{ll} A \rightarrow B : & \{J_A\}_{K_B} \\ B \rightarrow A : & \{J_B\}_{K_A} \end{array}$$

Each agent sends a nonce, encrypted with the another's public key, to be used as a half-key for the establishment of a session key.

7.4.1 Translating Basic Roles

The HLPSL specification of the basic roles `Alice` and `Bob` are as we might expect:

```
% Alice, the initiator
role Alice(A, B:agent,
           Ka, Kb: public_key,
           SND, RCV: channel (dy)) played_by A def=
  exists State:nat, Ja:text (fresh), Jb:text
  init State=0
  knowledge(A) = { inv(Ka) }
  transition
    step1. State=0 /\ RCV(start) =|> State'=1 /\ SND({Ja'}Kb)
    step2. State=1 /\ RCV({Jb'}Ka) =|> State'=2
end role

% Bob, the responder
role Bob(A, B:agent,
```

```
        Ka, Kb: public_key,
        SND, RCV: channel (dy)) played_by B def=
  exists State:nat, Ja:text, Jb:text (fresh)
  init State=0
  knowledge(B) = { inv(Kb) }
  transition
    step1. State=0 /\ RCV({Ja'}Kb) =|> State'=1 /\ SND({Jb'}Ka)
end role
```

Translation of basic roles is quite straightforward. For each one, we require an initial predicate and a next-state relation. Such transition relations in TLA are generally formalised as disjunctions of possible single steps the system might take (like our intruder *Manipulate* action in the previous section). In HLPSL, we have no explicit disjunction operator, but the semantics of the `transition` section of a basic role is exactly the disjunction of the individual transitions. More precisely, the `transition` section of our role `Alice` above expresses that, at any time, `Alice` may execute either transition `step1` *or* transition `step2` (provided, of course, that the respective guards on the left-hand sides hold). Recall that, in HLPSL, the order in which the individual steps are specified in the `transition` section is unimportant, so we have simple disjunction.

The TLA translation of role `Alice` follows. Here, we parametrise all predicates, because there may of course be several instances of `Alice` executing in parallel. Note, however, that each predicate is parametrised only by those state variables it requires, not with Alice's entire state. This is done for brevity, though it would of course be correct to include all of Alice's state variables in each next-state predicate. Also worthy of note is that we assume that the dummy `start` message is simply the string "start".

Alice's initial predicate:

$$Init\_Alice(State) \triangleq State = 0$$

Step1 – The reception of the start message and reply with the first message of the protocol:

$$\begin{aligned} &Alice\_Step1(State, SND, SND\_flag, RCV, RCV\_flag, Ja, Kb) \triangleq \\ &\quad \wedge\ State = 0 \\ &\quad \wedge\ RCV\_flag' \neq RCV\_flag \\ &\quad \wedge\ RCV' = \mathsf{start} \\ &\quad \wedge\ Ja' \in Msg \\ &\quad \wedge\ Fresh(Ja') \\ &\quad \wedge\ UsedNonces' = UsedNonces \cup \{Ja'\} \\ &\quad \wedge\ SND' = ACrypt(Kb, Ja') \\ &\quad \wedge\ toggle(SND\_flag) \\ &\quad \wedge\ State' = 1 \end{aligned}$$

Step2 – Reception of the second message of the protocol:

$$\begin{aligned} &Alice\_Step2(State, RCV, RCV - flag, Ka, Jb) \triangleq \\ &\quad \wedge\ State = 1 \\ &\quad \wedge\ RCV\_flag' \neq RCV\_flag \\ &\quad \wedge\ RCV' = Crypt(Kb, Jb') \\ &\quad \wedge\ State' = 2 \end{aligned}$$

Here, we assume a typed model: in *Alice_Step*2, only messages that are indeed encrypted terms will be accepted. In the untyped model, however, the translation to TLA is just as simple: we merely loosen the restriction and have Alice accept any valid message.

Given these definitions, we can define Alice's next-step relation. Firstly, the intruder could perform a *Manipulate* step in order to augment his knowledge. Alternatively, Alice could execute either of the two transitions defined above. However, recall that communication in HLPSL is synchronous. More specifically, for Dolev-Yao intruder models, communication is synchronous between agents and the intruder. For this reason, the next-state relation is not as simple as the disjunction of these two action predicates. Rather, in order to preserve semantic equivalence with HLPSL's synchronous communication, we must require that each send action (respectively receive action) is executed simultaneously with a *Divert* action (respectively *Impersonate* action) on the same channel. Bearing in mind that the semantics of our immediate reaction transition requires that both sides of the transition occur simultaneously, the complete next-state relation for the Alice role is defined as follows:

$$
\begin{array}{l}
Next\_Alice(State, SND, SND\_flag, RCV, RCV\_flag, Ja, Jb, Ka, Kb) \triangleq \\
\quad \vee\ Manipulate \\
\quad \vee\ \wedge\ Impersonate(RCV, RCV\_flag) \\
\qquad \wedge\ Alice\_Step1(State, SND, SND\_flag, RCV, RCV\_flag, Ja, Kb) \\
\qquad \wedge\ Divert(SND, SND\_flag) \\
\quad \vee\ \wedge\ Impersonate(RCV, RCV\_flag) \\
\qquad \wedge\ Alice\_Step2(State, RCV, RCV\_flag, Ka, Jb)
\end{array}
$$

The semantics of the `transition` section for basic roles is equivalent to Lamport's "square-box" notation ($\Box[trans]_v$) which asserts that, for any subset of state variables v and next state predicate *trans* any transition which affects the variables in v must satisfy *trans*. Given our translation of the `Alice` role, therefore, we should expect that the next-state relation for the whole system will include one conjunct of the form $\Box[Next\_Alice]_{\langle State, SND, RCV, Ja, Jb, Ka, Kb\rangle}$ for each instance of `Alice`.

The TLA translation of the `Bob` role is analogous and is shown below:

Bob's initial predicate:

$$Init\_Bob(State) \triangleq State = 0$$

Step1 – The reception of the first message of the protocol and Bob's reply:

$$
\begin{array}{l}
Bob\_Step1(State, SND, SND\_flag, RCV, RCV\_flag, Ja, Jb, Ka, Kb) \triangleq \\
\quad \wedge\ State = 0 \\
\quad \wedge\ RCV\_flag' \neq RCV\_flag \\
\quad \wedge\ RCV' = ACrypt(Kb, Ja') \\
\quad \wedge\ Jb' \in Msg \\
\quad \wedge\ Fresh(Jb') \\
\quad \wedge\ UsedNonces' = UsedNonces \cup \{Jb'\} \\
\quad \wedge\ SND' = Crypt(Ka, Jb') \\
\quad \wedge\ toggle(SND\_flag) \\
\quad \wedge\ State' = 1
\end{array}
$$

$$
\begin{array}{l}
Next\_Bob(State, SND, SND\_flag, RCV, RCV\_flag, Ja, Jb, Ka, Kb) \triangleq \\
\quad \vee\ Manipulate \\
\quad \vee\ \wedge\ Impersonate(RCV, RCV\_flag) \\
\qquad \wedge\ Bob\_Step1(State, SND, SND\_flag, RCV, RCV\_flag, Ja, Jb, Ka, Kb) \\
\qquad \wedge\ Divert(SND, SND\_flag)
\end{array}
$$

7.4.2 Translating Composed Roles

Completing our HLPSL specification, we now define a `SHARE` role in which, for each desired protocol session, we instantiate one `Alice` role and one `Bob` role. Finally, we define and

instantiate the Environment role, which calls the SHARE role with three sessions: one between agents a and b, one between a and i, and one between i and b (where, in each case, the former agent plays the Alice role and the latter the Bob role).

In the instantiation of the SHARE role, we assume that each pair of communicating agents shares two pairs of channels: one send channel from Alice to Bob (and one vice-versa), and one receive channel for Bob to receive messages from Alice (and one vice-versa). Here, we adopt the naming convention for channel variables that s_xy stands for "channel on which agent x sends messages intended for agent y" and r_xy should be read "channel on which agent x receives messages believed to be coming from agent y." Recall that in HLPSL, regardless of *intended* sender and receiver, all DY channels are assumed to connect directly to the intruder. Thus, when the intruder plays a protocol role under his own name (as an honest agent, that is), the respective channels then connect the intruder with himself.

```
% In the SHARE role, we instantiate one initiator and
% one responder for each protocol session.
role SHARE( S,R: (agent,agent) -> channel (dy),
                  Sessions: (agent,agent,public_key,public_key) list) def=
  composition
  /\_{in((A,B,Ka,Kb),Sessions)}
        Alice(A,B,Ka,Kb,S(A,B),R(A,B))
     /\ Bob(A,B,Ka,Kb,S(B,A),R(B,A))
end role

% Within the Environment role, we instantiate the SHARE
% role with the desired protocol sessions.
role Environment() def=
  knowledge(i) = { a, b, i, ka, kb, ki }
  composition
  SHARE(  [((a,b),s_ab), ((b,a),s_ba),
           ((a,i),s_ai), ((i,a),s_ia),
           ((i,b),s_ib), ((b,i),s_bi) ], % S -- send channels
          [((a,b),r_ab), ((b,a),r_ba),
           ((a,i),r_ai), ((i,a),r_ia),
           ((b,i),r_bi), ((i,b),r_ib) ], % R -- receive channels
          [(a,b,ka,kb),(a,i,ka,ki),
           (i,b,ki,kb)])                % Sessions
end role

Environment()
```

To translate composed roles, we "flatten" our instantiations as far as possible. In this case, we are instantiating a finite number of sessions involving a finite number of agents. Expanding the instantiation gives the following:

```
   % First session
   Alice(a,b,ka,kb,s_ab,r_ab)      % Alice1
/\ Bob(a,b,ka,kb,s_ba,r_ba)        % Bob1
   % Second session
/\ Alice(a,i,ka,ki,s_ai,r_ai)      % Alice2
/\ Bob(a,i,ka,ki,s_ia,r_ia)        % Bob2
   % Third session
/\ Alice(i,b,ki,kb,s_ib,r_ib)      % Alice3
/\ Bob(i,b,ki,kb,s_bi,r_bi)        % Bob3
```

As indicated in the comments above, we will refer to the individual instances as Alice1, 2, and 3 and Bob1, 2, and 3. Each, of course, requires its own local variables. These we introduce via a simple existential quantifier. As a last step, we must also specify the initial knowledge of the intruder (which must include the dummy start message). As in HLPSL, free variables are implicitly assumed to be global constants. For brevity, we will write $vars(X)$ to mean all state variables of role instance X. Our final system specification is then:

$$
\begin{aligned}
\mathit{InitIntruder} \triangleq\ & \wedge\ \mathsf{start} \in \mathit{IK} \\
& \wedge\ a, b, i, ka, kb, ki \in \mathit{IK}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Session1Spec} \triangleq\ & \exists\, a1\_state, a1\_ja, a1\_jb, \\
& b1\_state, b1\_ja, b1\_jb\ : \\
& \wedge\ \mathit{Init\_Alice}(a1\_state) \\
& \wedge\ \mathit{Init\_Bob}(b1\_state) \\
& \wedge\ \Box[\mathit{Next\_Alice}(a1\_state, s\_ab, sab\_flag, r\_ab, rab\_flag \\
& \qquad a1\_ja, a1\_jb, ka, kb)]_{vars(Alice1)} \\
& \wedge\ \Box[\mathit{Next\_Bob}(b1\_state, s\_ba, sba\_flag, r\_ba, rba\_flag \\
& \qquad b1\_ja, b1\_jb, ka, kb)]_{vars(Bob1)}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Session2Spec} \triangleq\ & \exists\, a2\_state, a2\_ja, a2\_jb, \\
& b2\_state, b2\_ja, b2\_jb\ : \\
& \wedge\ \mathit{Init\_Alice}(a2\_state) \\
& \wedge\ \mathit{Init\_Bob}(b2\_state) \\
& \wedge\ \Box[\mathit{Next\_Alice}(a2\_state, s\_ai, sai\_flag, r\_ai, rai\_flag \\
& \qquad a2\_ja, a2\_jb, ka, ki)]_{vars(Alice2)} \\
& \wedge\ \Box[\mathit{Next\_Bob}(b2\_state, s\_ia, sia\_flag, r\_ia, ria\_flag \\
& \qquad b2\_ja, b2\_jb, ka, ki)]_{vars(Bob2)}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Session3Spec} \triangleq\ & \exists\, a3\_state, a3\_ja, a3\_jb, \\
& b3\_state, b3\_ja, b3\_jb\ : \\
& \wedge\ \mathit{Init\_Alice}(a3\_state) \\
& \wedge\ \mathit{Init\_Bob}(b3\_state) \\
& \wedge\ \Box[\mathit{Next\_Alice}(a3\_state, s\_ib, sib\_flag, r\_ib, rib\_flag \\
& \qquad a3\_ja, a3\_jb, ki, kb)]_{vars(Alice3)} \\
& \wedge\ \Box[\mathit{Next\_Bob}(b3\_state, s\_bi, sbi\_flag, r\_bi, rbi\_flag \\
& \qquad b3\_ja, b3\_jb, ki, kb)]_{vars(Bob3)}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{SHARE\_Spec} \triangleq\ & \wedge\ \mathit{InitIntruder} \\
& \wedge\ \mathit{Session1Spec} \\
& \wedge\ \mathit{Session2Spec} \\
& \wedge\ \mathit{Session3Spec}
\end{aligned}
$$

A note about types: One difference between HLPSL and TLA is that the former is typed, while the latter is not. For this reason, to ensure semantic equivalence, we must explicitly state invariants on the types of variables and constants we use. Given a type invariant *TypeInv* for a system specification *Spec*, we must then prove that $\mathit{Spec} \Rightarrow \Box \mathit{TypeInv}$. The typing system of the HLPSL2IF compiler, however, ensures that this should hold trivially for any legal HLPSL specification.

8 Conclusions and Current Work

The choice of basing HLPSL on TLA afforded us a "best of both worlds" situation in which we can take advantage of an existing language with a rich semantics while also augmenting it with constructs specific to protocol modelling that make it a convenient language in practice.

The HLPSL language presented in this document has already proved itself to be an expressive and versatile language for modelling security protocols. Within the project we found it convenient to set up a group of people (containing representatives from all the project's partners led by Siemens), called the *modelling task-force*, devoted to the specification of protocols in HLPSL. To date, the modelling task-force has modelled over a dozen protocols in HLPSL, ranging from very simple ones such as SHARE to moderately complex protocols like IKE and TLS [11]. In practice, the framework of temporal logic, upon which HLPSL based, yields a language that is powerful yet readable and very intuitive to work with.

The benefit of experience, however, also brings to light new and updated requirements that may not have been evident during the initial design phase of the language. As we attempt to model more and more complex protocols, our modelling requirements point the way to possible improvements to the HLPSL language. Furthermore, additional feedback will come from work that will be reported in forthcoming deliverables (D3.1 "Security Properties", D2.2 "Assumptions on Environment", and D3.3 "Session Instances"). Here, we have attempted to document the current version of the HLPSL language: that is, the language that our modelling task-force has been working with and the language that has been the basis for the current version of the HLPSL2IF translator. We devote the rest of this section, however, to current work towards an even better High Level Protocol Specification Language.

8.1 Typing

We want a language in which we can effectively specify both typed and untyped simulations. In the former, agents accept only type-correct messages, while in the latter agents do not check the types of the messages they receive. We expect that, in general, protocol modellers will write typed HLPSL specifications and then decide at compile time (via an option to the HLPSL2IF translator) whether or not to include typing information in their simulations.

To this end, certain improvements are currently being integrated into the HLPSL grammar, documentation, and the HLPSL2IF translator. Firstly, appropriate syntax is required to specify the exact type of compound messages like pairs, function applications, and encrypted terms. The current proposal calls for the following addition to the grammar (see Appendix A for the complete grammar):

```
Compound_arg_type ::=
     Simple_type
   | Compound_arg_type "." Compound_arg_type
```

```
| "{" Compound_arg_type "}" Compound_arg_type
| "function" "(" Compound_arg_type ")"
| "inv" "(" Compound_arg_type ")"
```

A concrete example of message terms and their new types given this addition to the grammar is shown below:

```
A       : agent
Ka      : public_key
F       : function
A.Ka    : agent.public_key
{A.Ka}  : {agent.public_key}public_key
F(A)    : function(agent)
inv(Ka) : inv(public_key)
```

This improvement will allow for flexible description of typed simulations.

In addition, for the sake of clarity, we intend to introduce a new `nonce` type to replace `text (fresh)`. The `text` type is meant to represent any valid message data, and `text (fresh)`, accordingly, is message data that is freshly generated. However, we feel it is advantageous to disambiguate the two and create an explicit `nonce` type.

Finally, we are currently discussing the possibility of differentiating, syntactically, symmetric from asymmetric encryption. At the moment, which encryption scheme is meant is inferred from the key type: asymmetric encryption for public keys and inverses, and symmetric encryption for everything else. However, this precludes the modeller from specifying scenarios in which, for instance, a cryptographic hash value is used as a public key or a public key is used for symmetric encryption. Distinguishing between the two would allow the user to model such general situations.

8.2 Goals

The current version of HLPSL supports the standard authentication and secrecy goals described in Section 5. These are convenient and reduce the risk of missing attacks due to ill-specified goals. As the modelling task-force has found, they are also sufficient for describing a wide range of problems.

We intend, however, to introduce more general goals based on temporal logic. The basic idea will be to augment a HLPSL specification with additional events, then specify goals as restricted temporal formulae over these events.

These additional events will take the form of messages sent on special channels. These channels must be persistent, so that we have a history of the events that have taken place, and they should be global so that they are writable to all roles and in scope within the `goal` section of the specification. We illustrate by example. We use the following ASCII representations of LTL operators: `[]` for "henceforth" and `<->` for "sometime in the past."

We take as our first example the following authentication goal specified in the NSPK example of Section 6.1.

```
Alice weakly authenticates Bob on Nb
```

Assume we want to express this goal using the proposed goal framework rather than the standard authenticate goal. For the names of our special global channels, we will borrow the terms "witness" and "request" from the IF: on the `witness` channel, we will record events whenever one agent generates fresh data intended for another. On the `request` channel, we will register an event whenever one agents accepts fresh data as having come from another. Both channels will thus require the names of both agents and the message data on which they authenticate. However, there might be more than one authenticate goal between two agents. We therefore need some identifier specifying to *which* set of message data this witness (respectively request) event refers. The current proposal is to introduce a new type for this called `protocol_expression_id`. The messages on the two special channels are thus of type `(agent,agent,protocol_expression_id,text)`.

On the RHS of role `Bob`'s transition labelled `step1`, we then add the event

```
witness(B,A,cNb,Nb')
```

This expresses that the agent named in variable `B` has freshly generated the value `Nb'` for the agent named in variable `A`. The `cNb` term is a new constant that identifies the message term upon which the goal should authenticate, as described above.

Similarly, on the RHS of the transition `step2` within role `Alice`, we add the event

```
request(B,A,cNb,Nb')
```

This formalises Alice's acceptance of the value `Nb'` as having been generated for her by the agent named in `B`.

Having augmented our specification with these additional events, we can then specify the weak authentication goal explicitly as follows:

```
[]( request(B,A,cNb,Nb) -> <-> witness(B,A,cNb,Nb) )
```

Where `->` is normal logical implication.

Intuitively, this reads "whenever a request is made by agent `A` accepting the value `Nb` as coming from agent `B`, then sometime in the past there must have been a witness event in which `B` freshly generated the same value of `Nb` for `A`". This is precisely the definition of our weak authentication goal.

Our work in progress regarding the introduction of additional events and the specification of goals via temporal formulae over those events indicates that this is a general enough strategy to easily specify almost all safety properties. This will be a significant improvement to HLPSL as soon as it is in place.

References

[1] M. Abadi. Two facets of authentication. In *Proceedings of the 11th IEEE Computer Security Foundations Workshop (CSFW'98)*, pages 27–33. IEEE Computer Society Press, 1998.

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

[3] 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`.

[4] J. Gray and J. McLean. Using temporal logic to specify and verify cryptographic protocols (progress report). pages 108–116.

[5] D. Harkins and D. Carrel. RFC 2409: The Internet Key Exchange (IKE). 1998.

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

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

[8] 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.

[9] Z. Manna and A. Pnueli. The temporal framework for concurrent programs. In R. Boyer and J. Moore, editors, *The Correctness Problem in Computer Science*, pages 215–274. Academic Press, 1981.

[10] 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.

[11] L. C. Paulson. Inductive analysis of the internet protocol TLS. *ACM Transactions on Computer and System Security*, 2(3):332–351, 1999.

A HLPSL Grammar

```
%-------------------------------------------------
%  Grammar of the HLPSL
%-------------------------------------------------
% General conventions:
% * variables (var_ident) start with a capital letter
% * constants (const_ident) are integers or start with lowercase
% * comments (%...) will be intercepted by the lexical analysis
% * Grammatical elements whose names start with "Maybe_" are optional
SpecHLPSL ::= Role_definitions_list
              Maybe_goal_declaration
              % Call of the main role (ex: env)
              var_ident "(" ")"

Role_definitions_list ::= Role_definition
                        | Role_definition Role_definitions_list

% Roles may be either basic or compositional.
Role_definition ::= Basic_role
                  | Composition_role

% Basic roles must include a player definition and generally
% contain a transition declaration section.
Basic_role ::= Role_declaration Player_def Role_headers
               Maybe_transition_declaration
               "end role"

% Composition roles have no transition section, but rather
% a composition section in which they instantiate other roles.
Composition_role ::= Role_declaration Role_headers
                     Maybe_composition_declaration
                     "end role"

% A role declaration consists of the keywork "role", the role name
% and a list of formal argument
Role_declaration ::= "role" var_ident "(" Formal_arguments ")"

% A role's headers include:
% * the "def=" keyword
% * (optionally) declaration of local variables via "exists"
% * (optionally) declaration of any owned variables
% * (optionally) initalisation of variables in an "init" section
```

```
% * (optionally) definition of accepting states
% * (optionally) knowledge declaration(s)
Role_headers ::= "def="
                 Maybe_exists_declaration
                 Maybe_owns_declaration
                 Maybe_init_declaration
                 Maybe_accept_declaration
                 Maybe_knowledge_declaration

Formal_arguments ::= Variables_declaration_list
                   |

% Used to bind the role and the identifier of the agent playing the role
Player_def ::= "played_by" var_ident

% Exists is used to hide local variables.
% Local is synonymous.
Maybe_exists_declaration ::= "exists" Exists_declaration
                           | "local" Exists_declaration
                           |

Exists_declaration ::= Variables_declaration_list

% owns -- That a variable is "owned" by a role means that the value of
% this variable can only be changed as specified by this role.
% Subroles instantiated by the owning role may change the variable but
% may not own it themselves.
Maybe_owns_declaration ::= "owns" Owns_declaration
                         |

Owns_declaration ::= Variables_list

Maybe_init_declaration ::= "init" Init_declarations_list
                         |

Init_declarations_list ::= Init_declaration
                         | Init_declaration "/\" Init_declarations_list

Init_declaration ::=
     Expression "=" Expression
     % Initialisation of all the elements of a set/ list/ partial function
   | "/\" "_" "{" Parameters_instance "}" Init_declaration
```

```
% Acceptance is used for sequential composition to mark the stop states
% after which the following instantiation may begin.
Maybe_accept_declaration ::= "accept" Accept_declaration
                           |

Accept_declaration ::= Predicate

% Definition of the transition section (for basic roles)
Maybe_transition_declaration ::= "transition" Transitions_list
                               |

% Definition of the composition section (for composed roles)
Maybe_composition_declaration ::= "composition" Compositions_list
                                |

Maybe_knowledge_declaration ::= Knowledge_declaration
                                Maybe_knowledge_declaration
                              |

% Declaration of required knowledge
Knowledge_declaration :==
   "knowledge" "(" Variable_or_constant ")" "=" "{" Expressions_list "}"

Maybe_goal_declaration ::= Goal_declaration
                         |

% Goal_declaration defines the goals section
Goal_declaration ::= "goal" Goal_formulas_list "end goal"

Goal_formulas_list ::= Goal_formula
                     | Goal_formula Goal_formulas_list

Goal_formula ::=

     % var_ident here are only basic roles identifiers
     % "authenticates" is a shortcut for the LTL formula:
     % request (A,B,C,D) -> <-> witness(A,B,C,D)
     var_ident "authenticates" var_ident "on" Variables_list

     % Weak authentication
   | var_ident "weakly" "authenticates" var_ident "on" Variables_list

     % Secrecy goals
```

```
   | "secrecy_of" Variables_list

     % General LTL formulae
   | LTL_formula

Compositions_list ::=
     Composition
   | "/\" "_"  "{" Parameters_instance "}" Composition

Composition ::=
     Role_instance "/\" Composition
   | Role_instance

Role_instance ::=
     "(" Role_instance ")"
   | "LOOP" Role_instantiation
   | Role_instantiation ";" Role_instance
   | Role_instantiation

Role_instantiation ::=
     var_ident "(" Effective_arguments ")"

Variables_declaration_list ::=
     Variable_declaration
   | Variable_declaration "," Variables_declaration_list

Variable_declaration ::=
     Variables_list ":" Type_of

% attributes qualify certain types such as channel and text variables
Type_attribute ::=
     % The corresponding text variable has a new fresh value each time it
     % is primed
     "fresh"

     % Dolev-Yao channels
   | "dy"

     % Over-The-Air channels
   | "ota"

Type_of ::=
     Subtype_of
```

```
   | Subtype_of "->" Subtype_of

Subtype_of ::=
     Simple_type
   | Subtype_of "list"
   | Subtype_of "set"
   | Subtype_of "(" Type_attribute ")"
   | "(" Types_list ")"

Types_list ::=
     Subtype_of
   | Subtype_of "," Types_list

Simple_type ::=
     "agent"
   | "channel"
   | "public_key"
   | "text"
   | "nat"
   | "bool"
   | "symmetric_key"
   | "function"
   % hash is synonymous for function
   | "hash"
   | "{" Constants_list "}"

Constants_list ::=
     const_ident
   | const_ident "," Constants_list

Formula ::=
      Expression "=" Expression
   |  Expression "<" Expression
   | "in" "(" Expression "," Expression ")"
   | "not" "(" Expression "=" Expression ")"
   | "not" "(" "in" "(" Expression "," Expression ")" ")"
     % Syntactic sugar for inequality
   | Expression "/=" Expression
     % The two boolean values
   | "true"
   | "false"

Transitions_list ::=
```

```
     Transition
   | Transition Transitions_list

Transition ::=
     % Spontaneous actions are enabled when the state predicate on the
     % LHS is true.
     label "." Predicate "--|>" Actions_list

     % Immediate reactions fire immediately whenever the non stutter
     % predicate on the LHS is true;
   | label "." Predicate "=|>" Actions_list

Predicate ::=
     Formula
   | Formula "/\" Predicate
   | Variable_or_constant "(" Expressions_list ")"
   | Variable_or_constant "(" Expressions_list ")" "/\" Predicate
     % Dummy start message for Dolev-Yao models.
   | var_ident "(" "start" ")"

Expressions_list ::=
     Expression
   | Expression "," Expressions_list

Actions_list ::=
     Action
   | Action "/\" Actions_list

Action ::=
     var_ident "'" "=" Expression
     % For updating flexible functions
   | var_ident "'" "(" Effective_arguments ")" "=" Expression
   | Variable_or_constant "(" Expressions_list ")"

Parameters_instance ::=
     Bracketed_variables_list
   | "in" "(" Bracketed_variables_list "," Expression ")"

Bracketed_variables_list ::=
     var_ident
   | "(" Variables_list ")"

Variables_list ::=
```

```
     var_ident Var_param
   | var_ident Var_param "," Variables_list

Var_param:
     "(" Variables_list ")"
   |

% We allow composed messages as effective arguments of type text
Effective_arguments ::=
     Expressions_list
   |

Variable_or_constant ::=
     var_ident
   | const_ident
   | "i"


Expression ::=
     var_ident "'"
   | "inv" "(" Expression ")"

     % Concatenation, left-associative
   | Expression "." Expression

     % Encryption
   | "{" Expression "}" Expression

     % Grouping
   | "(" Expression_list ")"

     % Function application
   | Variable_or_constant "(" Expression_list ")"

     % Operator for insertion into a set or concatenation at the head of
     % a list
   | "cons" "(" Expression "," Expression ")"

   | Variable_or_constant
   | "[" "]"
   | "[" Expressions_list "]"
     % Domain of a function
   | "dom" Variable_or_constant
```

```
label ::= const_ident

% The syntax for general LTL goals is not yet fully agreed upon
LTL_formula ::=
[a-zA-Z0-9=><-,.{}’’]+\n

%end
```