A High-Level Protocol Specification Language for Industrial Security-Sensitive Protocols*

Y. Chevalier†, L. Compagna‡, J. Cuellar§,
P. Hankes Drielsma¶,
J. Mantovani‡, S. Mödersheim¶ and L. Vigneron†

Abstract. *This paper presents HLPSL, a high-level protocol specification language for the modelling of security-sensitive protocols. This language has a formal semantics based on Lamport's Temporal Logic of Actions. HLPSL is modular and allows for the specification of control flow patterns, data-structures, alternative intruder models, and complex security properties. It is sufficiently high-level to be accessible to protocol engineers (themselves not necessarily formal methods experts), yet easily translatable into a lower-level term-rewriting based language well-suited to model-checking tools. The accommodation of these contrasting features makes HLPSL able to easily specify modern, industrial-scale protocols on which existing specification languages only partially succeed.*

1. Introduction

Even assuming "perfect" cryptography, the design of security protocols is notoriously error-prone. Consumer confidence in Internet security and e-commerce infrastructures is eroding in the wake of several highly publicised security failures. Breaches in security can be very costly, particularly when they require the modification of deployed infrastructure. There is therefore a clear case to be made for the integration of formal methods into the engineering processes at standardisation committees for Internet protocols such as IETF, ITU, W3C, Oasis, IEEE, 3GPP, and OMA. The benefits of formal specification and analysis during the software engineering process are well understood: the construction of formal models serves to eliminate many ambiguities in the design process, and such models can then be rigorously analysed against formal specifications of their requirements to identify design flaws.

A variety of languages and tools (e.g. [1, 6, 11, 14, 21, 22, 23, 25, 26]) based on different formal methods and automated reasoning techniques have been devised and applied to the domain of security protocols. Unfortunately, few languages exist that are both sufficiently high-level to be accessible to engineers and protocol designers of standardisation bodies (themselves not necessarily experts in the area of formal methods) and also expressive enough to specify modern Internet protocols. For instance, constructs for modularity and flow control are often missing, making it difficult or even

†LORIA, Nancy, France; {Yannick.Chevalier,Laurent.Vigneron}@loria.fr.
‡Artificial Intelligence Lab, DIST, Università di Genova, Italy; {compa,jacopo}@dist.unige.it.
§Siemens AG, CT IC 3, 81730 Munich, Germany; jorge.cuellar@siemens.com.
¶Information Security, ETH, Zurich, Switzerland; {drielsma,moedersheim}@inf.ethz.ch.
*This work was partially funded by the FET Open Project IST-2001-39252 and the BBW Project 02.0431, "AVISPA: Automated Validation of Internet Security Protocols and Applications".

impossible to specify looping sub-protocols and other complex behaviour. Moreover, the model of the intruder is generally left implicit, thus assuming an intruder with identical capabilities over all parts of the network. This, however, may not be a faithful representation of modern network infrastructures, which often employ heterogeneous technologies vulnerable to differing intruder threats.

In this paper, we describe a language that provides all the features mentioned above: the *High-Level Protocol Specification Language (HLPSL)*. It has a formal semantics based on Lamport's Temporal Logic of Actions (TLA, [17]) that makes it easily translatable into a declarative lower-level term rewriting based language (the *Intermediate Format, IF* [3]) well-suited to automated analysis tools. HLPSL thus enjoys significant generality, as other tools can easily be made to employ HLPSL by simply adapting them to accept IF specifications as input. HLPSL is modular and allows for the specification of complex control-flow patterns, data-structures, and different intruder models. Using a formal language with a temporal logic semantics to formalise security properties gives us great generality and expressiveness. Finally, HLPSL is not restricted to logicians, but it is particularly suited for engineers and protocols designers. Indeed, HLPSL has been devised as part of the AVISPA project (`http://www.avispa-project.org`), which aims to develop push-button, industrial-strength technology — supported by expressive specification languages like HLPSL — for the analysis of large-scale Internet security-sensitive protocols and applications. In this context, we are working to introduce the use of HLPSL and public domain tools based on formal methods into the design phase at the IETF and other standardisation bodies to hopefully accelerate the standardisation of security protocols and improve their correctness.

In more detail, the AVISPA tool takes as input a HLPSL specification that is translated into a corresponding IF specification. This is then analysed by invoking state-of-the-art back-ends (currently CL-AtSe [27], OFMC [5], and SATMC [2] are supported) which return attacks (if any) to the user in an intuitive and readable output format.

Structure of the paper. After a brief explanation of notation and conventions, we start in Section 2 by introducing the HLPSL language via a running example. In Section 3, we explain the TLA-based semantics of HLPSL. Section 4 presents modelling results. We discuss related work in Section 5. We conclude in Section 6 with some final remarks.

1.1. Basic TLA and HLPSL Conventions

The semantics of our High-Level Protocol Specification Language is based on Lamport's Temporal Logic of Actions (TLA, [17]). TLA is an elegant and powerful language which lends itself well to specifying concurrent systems like the types of protocols we seek to model here. In TLA, system behaviour is modelled by describing the *state* and then specifying the ways in which that state may change. The global state is defined by an assignment of values to all the system variables. Similarly, the state of a TLA module (or *role*, as we call them in HLPSL) is defined by an assignment of values to all the variables of the role, that is, those variables that are visible from within the role. The description of the transition relation is then given by predicates that relate the values of variables in one state (the current one) and another (the future, or next state). We refer to the variables in the next state as *primed* variables. A *state predicate* or *state formula* is a first-order formula on a role's state variables and constants. A *transition predicate* or *formula* is similar but may include primed variables.

A *basic event* is a conjunction of transition predicates, at least one of which is of the form $p(V') \neq$

1. $A \rightarrow S$: $A.B$
2. $S \rightarrow A$: $\{B.K_B\}_{inv(K_S)}$
3. $A \rightarrow B$: $\{N_A.A\}_{K_B}$
4. $B \rightarrow S$: $B.A$
5. $S \rightarrow B$: $\{A.K_A\}_{inv(K_S)}$
6. $B \rightarrow A$: $\{N_A.N_B\}_{K_A}$
7. $A \rightarrow B$: $\{N_B\}_{K_B}$

Figure 1. The NSPK protocol.

$p(V)$, where V is a tuple of variables and $p(V)$ is a state predicate. This definition ensures that events are *non-stuttering*, i.e. at least one state variable changes. The set of *events* is the closure under conjunction and disjunction of basic events.

The usual canonical form of a module in TLA (a safety property) is $Init \wedge \Box[Next]_V$, where $Init$ is a predicate describing the initial state, and $Next$ is a predicate describing the transition taken if at least one variable in V changes. Equivalently, the term $[Next]_V$ may instead be written as a conjunction of terms of the form $event \Rightarrow Next$ (without a subscript V).

2. The High-Level Protocol Specification Language

We introduce HLPSL with the help of a running example, the complete Needham-Schroeder Public Key protocol (NSPK [24]) including key-server,[1] shown in Figure 1.[2] While this is a relatively simple protocol, this example still allows us to illustrate some of the more advanced features of our protocol specification language. For instance, we add the requirement that each agent maintains a keyring of public keys (KR in the HLPSL specification given in Fig. 2): that is, a set of the agents and their respective public keys that the agent has already learned from the trusted key-server S. Moreover, an agent X should contact S *only* if X does not yet possess the public key of his communication partner in his keyring. Such "if-then-else" style flow control is often missing from existing specification languages; indeed, such control patterns cannot even be adequately described in the simple Alice & Bob notation (e.g. Fig. 1) that is standard in the literature. Also, the modelling of such a keyring requires the specification of a set of messages shared between all protocol sessions in which a given agent participates. This too is a non-trivial challenge that many specification languages cannot meet.

Roles. We model the protocol in a modular fashion, focusing not on the exchange of messages that takes place as in several existing approaches such as [11], but rather dividing the specification into several *roles*. Roles may be parametrised and may also declare local variables. We distinguish between two different types of roles: *basic roles* describe the actions of one agent involved in a single protocol or sub-protocol execution, whereas *composed roles* instantiate and conjoin one or more other roles. The specification of the example roles `Alice` and `Server` is shown in Figure 2.[3] A basic role may be seen as the analogue to a module in TLA, describing an initial predicate and a next-state relation. Roles are defined over a set of parameters (which are variables that can be shared between roles) and a set of local variables, not accessible from outside the role.

Though not illustrated in this example, roles may declare *ownership* of parameters, asserting that the

[1] A detailed description of the protocol goes beyond the scope of this paper. We refer the interested reader to [8].

[2] We use . to denote the concatenation of messages and $inv(K)$ to indicate the inverse of public key K.

[3] Comments in HLPSL begin with the '%' symbol and continue to the end of the line. By HLPSL convention, all protocol sessions begin with the occurrence of the START signal (see Section 3.).

```
role Alice(A, B, S: agent, Ka, Ks: public_key;
     KR: (agent,public_key) set,  SND, RCV: channel (dy)) played_by A def=
 local 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 /\ START() /\ in((B,Kb'), KR)
      =|> State'=2 /\ SND({Na'.A}Kb') /\ witness(A,B,na,Na')
  % Start of the protocol otherwise.
  step1b. State =0 /\ START() /\ not(in((B,Kb'), KR))
      =|> State'=1 /\ SND(A.B)
  % Receipt of response from server
  step2.  State =1 /\ RCV({B.Kb'}inv(Ks))
      =|> State'=2 /\ KR' = cons((B,Kb'), KR)
                   /\ SND({Na'.A}Kb') /\ witness(A,B,na,Na')
  % Receiving the second message and sending the third.
  step3.  State =2 /\ RCV({Na.Nb'}Ka)
      =|> State'=3 /\ SND({Nb'}Kb) /\ request(A,B,nb,Nb')
end role

role Server(S: agent, Keys: function;
            SND, RCV: channel (dy)) played_by S def=
  local A, B: agent
  transition
      step0. RCV(A'.B') =|> SND({B'.Keys(B')}inv(Keys(S)))
end role
```

Figure 2. Specification of the NSPK protocol (the Bob role is defined analogously to the Alice role).

owned variables may change in *only* the way described by the owning role despite the fact that they are visible from outside. A role may also define an *acceptance* predicate to indicate the conditions for "successful" completion. This is needed for sequential composition and looping: if we have two sequentially composed role instantiations, then the second role begins execution after the first one has accepted.

Types. In HLPSL, both variables and constants are typed.[4] Types are specified using the '`:`' symbol; for instance, `agent` represents the type of agent names, and `text(fresh)` the type of freshly generated nonces. Typing is used to exclude implementation-dependent type-flaw attacks. The AVISPA tool can, however, employ an untyped model by ignoring all type information.

HLPSL also allows us to declare new function symbols. Such functions assume the properties of perfect cryptographic hash functions: they are injective and not invertible by the intruder. Moreover, function symbols are themselves messages like any other. They can therefore be either known or unknown to the intruder (modelling a publicly known or secret function, respectively) and can be transmitted within messages (modelling the negotiation of cryptographic algorithms). They can also be employed to model key-tables. For instance, in the NSPK example, we have used the function `Keys` to model the association of agents to their public keys.

Operators. Since many protocols use identical algebraic properties of operators on messages, we find it convenient to separate their definitions into a so-called *prelude file* (as in CAPSL [11]). The

[4] Note that, in HLPSL, variable names begin with a capital letter and constant names with a lower-case letter. We refer to the intruder by the constant `i`.

standard prelude includes such operators as exponentiation and XOR and should be sufficient for many scenarios, but the ability to introduce new operators and specify their algebraic properties gives the user great flexibility: for instance, to model the properties of a particular cryptosystem used in a given protocol.

Transitions. Transitions relate a trigger event on the LHS with an associated action on the RHS, separated by `=|>`. We use `/\` to denote normal conjunction. Each transition is triggered whenever its guard event predicate is satisfied and fires immediately; we therefore refer to transitions also as *immediate reactions*. For instance, Alice's last transition (`step3` in Fig. 2) states that when she is in `State` equal to 2 and she receives a message encrypted with her public key and containing the already known value of nonce `Na` (the one she has previously sent) and a new value[5] of nonce `Nb`, then `State` is updated, a message encrypted with Bob's public key is sent and a goal fact is asserted (see below).

Communication. Communication in HLPSL is synchronous and takes place over channels, 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)`.

In particular, in the well-known Dolev-Yao (DY, [13]) intruder model, all communication is synchronous with the intruder, i.e. every message received by an honest agent comes from the intruder, and every message sent by an honest agent goes directly to the intruder and is added to his knowledge. This is not a restriction, as the DY intruder may intercept any message and replay it to any other agent. Thus, even though the model is synchronous, a message is not necessarily received by the intended recipient, but rather by the intruder. We may thus say that we identify the intruder with the network. Also, in the DY model, we can model every transition of honest agents as an immediate reaction to an incoming message: this reflects [12]. Together, the identification of intruder and network combined with this immediate reaction comprise a technique known as *step compression*: we need not consider all interleavings of intermediate transitions of the honest agents. In [5], it is also discussed why this is equivalent to the DY intruder.

Intruder Models. Security protocols execute on heterogeneous communication channels characterised by different security properties and thus requiring different intruder models. To faithfully model such settings, each channel is associated with a particular intruder model, and our logic-based semantics allows us to describe alternative intruder models in a simple axiomatic manner. Intruder capabilities are described in the prelude file, while, in HLPSL, the `channel` type takes an attribute (`(dy)` in our example) which serves as a macro specifying which intruder model should be used. Each such macro corresponds to a set of intruder axioms in the prelude file, which includes, for instance, axioms describing channels controlled by the DY intruder; axioms describing channels modelling "location-limited" communication [10] upon which the intruder cannot, in general, prevent messages from reaching their destinations; axioms describing secure channels to which the intruder has no access whatsoever; and channels which provide non-repudiation properties. The user can also define new channel types. In HLPSL, The explicit specification of the intruder's capabilities allows us to model modern heterogeneous networks (by equipping roles with multiple channels of different types) and enables us to easily analyse protocols under alternative intruder models (by simply experimenting with different channel types). This is particularly important for formalising some of the protocols currently under discussion at the IETF [4] as well as protocols executing in settings like

[5] A primed variable in a field of a receiving channel indicates the receipt of a new value for this variable.

```
role Environment() def=
 local Kr_A, Kr_B, Kr_I : (agent,public_key) set
 init Kr_A = nil /\ Kr_B = nil /\ Kr_I = nil
 const keys : function, na, nb: protocol_id
 knowledge(i) = {a,b,i,keys,inv(keys(i))}
 composition
  Server(s,keys;snd_srv,rcv_srv) /\
  % Session 1, between agents a and b
  Alice(a,b,s,keys(a),keys(s);Kr_A,s_a,r_a) /\
  Bob(a,b,s,keys(b),keys(s);Kr_B,s_b,r_b) /\
  % Session 2: agent a talking to the intruder, posing as Bob
  Alice(a,i,s,keys(a),keys(s);Kr_A,s_a,r_a)
end role
```

Figure 3. Instantiation via the `Environment` role.

those of [10].

Instantiation. Given the basic roles that make up our protocol specification, we can now instantiate them via composed roles. Composition can be sequential (using the `;` operator) or parallel (using the `/\` operator). Using these composition operators, we can model situations such as an agent who executes several sub-protocols of a particular protocol suite in order, or an agent who is involved in several protocol sessions at once.

One possible instantiation of our NSPK example protocol is shown in Figure 3. Here, we instantiate a single trusted server and two sessions of the protocol in parallel: one between agents `a` and `b` and one in which agent `a` initiates a protocol run with the intruder `i`. Note that in this latter session, the intruder need not be explicitly modelled by a call to role `Bob`, as his DY capabilities subsume this. Although we give one particular ground instantiation in the example, we can also specify an instantiation that contains variables for the agent names and thereby symbolically represents a set of possible instantiations. The given scenario is sufficient to detect, for instance, the man-in-the middle attack on NSPK described by Lowe in [19].

The instantiation also illustrates another feature of HLPSL: shared knowledge. We can easily associate a given variable, in this case the keyrings, to a particular agent, and share that variable across all protocol sessions in which the agent participates. Here, agent `a` plays in two parallel instances of `Alice`, and any public keys she learns in one will thus be available to her in the other. Such variable sharing is important for the modelling of group protocols and greatly enhances the expressiveness of our language.

Goals. We focus on safety temporal properties [4]. To specify goals, we can compose temporal formulae using the operators $\Box$ (“always”), $\blacklozenge$(“sometime in the past”), and $\ominus$ (“one time instant in the past”), as well as standard logical connectives such as conjunction and negation. Goal formulae are defined over so-called *goal events*: semantically, predicates that are true at the moment they appear on the RHS of a transition. Note that a rich set of security goals are expressible in this way, including secrecy, authentication, and a host of others described in [4]. Generality comes at a price, as the user must manually place the goal facts correctly or risk a false specification. However, standard goals like authentication and secrecy are well understood and require goal facts in a set of standard situations. The protocol modeller can thus be greatly helped by complete documentation. Moreover, correct

placement of the goal facts encourages a thorough understand of the protocol and its goals, which one might consider more of a feature than a bug.

Assume `Bob` should strongly authenticate `Alice` on her nonce `Na`. The user is free to define arbitrary goal facts and interpret them as is desired. By convention, we call our goal facts for authentication goals `witness` and `request`, whose intuitive meanings are as follows:

- `witness(A,B,na,Na)` should be read "agent `A` wants to execute the protocol with agent `B` and use value `Na` as her nonce." In order to specify for which variable of the protocol a particular value was meant (in this case the protocol variable `Na`), we specify a unique identifier for this variable (a so-called `protocol_id`), in this case `na`;
- `request(B,A,na,Na)` should be read "agent `B` accepts the value `Na` and now relies on the guarantee that agent `A` exists and agrees with him on this value for `protocol_id` `na`."

We express our desired security property in a `goal` declaration as follows:

```
goal
 % Strong authentication.
 □(request(B,A,na,Na) --> (⟐witness(A,B,na,Na) /\
                           not(⊙⟐request(B,A,na,Na))))
end goal
```

Intuitively, it is always true that a `request` event has been preceded by an accompanying `witness` event. Moreover, no agent should accept the same value twice from the same communication partner: that is, as of one time point before a `request` event, the same value had never been previously requested. This definition corresponds to Lowe's notion of agreement in [20].

3. HLPSL Semantics

The semantics of HLPSL is based on TLA, a powerful logic which is well-suited to the specification of concurrent systems like security protocols. TLA itself has an intuitive and easily understandable semantics, making it a formalism that protocol designers and engineers can find accessible.

3.1. Messages, Nonces, and the Intruder Model

We begin by specifying the structure of messages and the properties of the operations on the set Msg of all messages. For brevity, we focus on those operations needed for our running example: namely, pairing $Pair(M_1, M_2) = M_1.M_2$ and asymmetric encryption, $ACrypt(K, M) = \{M\}_K$; however, we can easily extend this model of messages to include other operators like symmetric encryption, exponentiation, and XOR, and their associated properties. As the basic data type we use the quotient algebra $\mathcal{T}_\Sigma/\approx$ of the free term algebra $\mathcal{T}_\Sigma$ modulo the equations $\approx$. As an example, $\approx$ explicitly includes an equation stating that pairing is associative: $Pair(Pair(m1, m2), m3) \approx Pair(m1, Pair(m2, m3))$; this property is essential for considering all the possible representations of the same term ($m1.m2.m3$ for example).

In HLPSL, variables may be tagged as being `fresh`, that is, each time that they are updated they take a new unseen and unguessable value. $Nonce\_vars$ is the collection of

$$
\begin{array}{lcl}
Read & \triangleq & \exists_m \ \wedge SND(m) \ \wedge \ IK' = IK \cup \{m\} \\
ASplit & \triangleq & \exists_{m1,m2} \ \wedge Pair(m1,m2) \in IK \ \wedge \ IK' = IK \cup \{m1,m2\} \\
AAdec & \triangleq & \exists_{k,m} \ \wedge ACrypt(k,m) \in IK \ \wedge \ inv(k) \in IK \ \wedge \ IK' = IK \cup \{m\} \\
GPair & \triangleq & \exists_{m1,m2} \ \wedge m1 \in IK \ \wedge \ m2 \in IK \ \wedge \ IK' = IK \cup \{Pair(m1,m2)\} \\
GAcrypt & \triangleq & \exists_{k,m} \ \wedge k \in IK \ \wedge \ m \in IK \ \wedge \ IK' = IK \cup \{ACrypt(k,m)\} \\
GFresh & \triangleq & \exists_x \ \wedge x \notin Used \ \wedge \ IK' = IK \cup \{x\} \ \wedge \ Used' = Used \cup \{x\}
\end{array}
$$

Figure 4. Dolev-Yao intruder knowledge formulae.

all such variable names. Now we need a means of enforcing freshness of nonces in TLA. For this, we simply keep track of those nonce values that have already been used:

$$
\begin{array}{lcl}
Nonce\_Prop & \triangleq & \Box \ \wedge Used \subset Used' \\
 & & \quad \wedge \textit{ for all variables } \theta \in Nonce\_vars : \quad \theta' \neq \theta \Rightarrow (\theta' \notin Used \wedge \theta' \in Used')
\end{array}
$$

We formalise the capabilities of the intruder as a set of rules the intruder may execute. We focus here on the well-known Dolev-Yao (DY) intruder model of [13] but note that the definition of alternate intruder models is a simple matter of axiomatically describing their capabilities. In this way, we can easily model a system in which the intruder has full DY capabilities over certain communication channels, can only listen on others, and has no access to a third set of channels.

For simplicity, we assume here that there are only two channels, SND and RCV, shared by all honest agents to send and receive messages, respectively.[6] The intruder reads SND (every message that the agents write), analyses the messages, (i.e. generates terms and messages based on them), and inserts the composed messages into RCV. The "knowledge of the intruder", IK, is the set of all terms that the intruder may create. The initial value of this variable is explicitly set in HLPSL (and is augmented by the initial knowledge of any agent roles played by the intruder) and changes according to the formulae of Figure 4: when the intruder reads new messages, $Read$, when he analyses his knowledge (decomposing a pair into its components, $ASplit$, or decrypting encrypted terms, if he possesses the appropriate key, $AAdec$), or when he composes new terms (generating pairs, $GPair$, encrypting a message using a known key, $GAcrypt$, or generating fresh nonces, $GFresh$). IK may only change when one of these actions happens, allowing the intruder to introduce his knowledge into the network. The intruder behaviour is thus formalised by the following formula:

$$
\begin{array}{lcl}
Intruder_{DY} & \triangleq & \Box \ \wedge IK' \neq IK \Rightarrow \vee Read \vee ASplit \vee AAdec \\
 & & \qquad \vee GPair \vee GAcrypt \vee GFresh \\
 & & \wedge RCV(msg) \Rightarrow msg \in IK'
\end{array}
$$

3.2. Translating HLPSL Roles to TLA

In this subsection, we show how HLPSL specifications are mapped into TLA. For simplicity, in this section we do not distinguish between variables that are local to a role (or module) and variables that are owned by the role. More precisely, we rename local variables to avoid name clashes with the environment and later we replace them by *owned* shared variables. They are in principle visible to the environment, but due to the renaming have absolutely no effect on it. Thus this transformation preserves the semantics.

[6] We also use a "signal", syntactically of the same form as a channel, but passing no value, the START signal. Each occurrence of START represents an independent event. In the translation to TLA, we first rename each occurrence of START( ) to a different START_i, for $i = 1, 2, \ldots$. START_i is then an event that can happen at any time.

Here is the structure of basic roles and composed roles in HLPSL, where A and B are roles (the sequential composition, the loop construct and the acceptance conditions are not discussed here for reasons of space):

```
role Basic(...) ...def=
  owns Θ
  init Init
  transition
    event_1 =|> action_1
    event_2 =|> action_2
    ...
end role
```

```
role Par(...) def=
  owns Θ
  init Init
  composition
    A ∧ B
end role
```

We will proceed inductively translating to TLA, starting with $Basic$ and then giving the translation of the composed role Par in terms of the translation of its components, A and B.

Let us first define the TLA translation of a basic role.

$$TLA(Basic) \triangleq Init(Basic) \wedge \Box[\bigwedge_i (event_i \Rightarrow action_i) \wedge (\bigwedge_{\theta \in \Theta} \theta' \neq \theta \Rightarrow Mod(\theta, Basic))]$$

Initially, $Init$ holds, and in every step, if an event is triggered, then the changes specified by the corresponding action take place. Moreover, if one of the variables owned by the role changes, then the variable is actually modified by this role. It is our convention that if a role owns a variable then this variable is never modified by any role "outside" the current one.

An agent may simultaneously participate both in different roles and in different sessions of the protocol. In this case, the two role instances could share some internal variables of the agent. This variable sharing is not done via some channel.

In order to explain how to construct the predicates $Init(Role)$ and $Mod(\theta, Role)$, let us first define what it means that $Transition_i$ (that is, $event_i \Rightarrow action_i$) modifies the variable X. We begin by syntactically transforming any transitions of the form $RCV(\ldots, X', \ldots) \wedge \ldots \Rightarrow \ldots$ into the equivalent form: $RCV(\ldots, \xi, \ldots) \wedge \ldots \Rightarrow (X' = \xi) \wedge \ldots$. Here, ξ is a fresh variable that is only used during this transition and is not needed to construct the state space. Our (simplified) convention is that $Transition_i$ modifies X if (after this transformation) X' appears free on the RHS of the transition, but not on the LHS. Given this convention, we can define the required predicates, as well as the set of variables owned by $Role$, $\Theta(Role)$, as follows:

$$\begin{array}{lcl} \Theta(Basic) & \triangleq & \Theta \\ Init(Basic) & \triangleq & Init \\ Mod(x, Basic) & \triangleq & \bigvee_i \{event_i \mid Transition_i \ modifies \ x\} \end{array}$$

Note that a transition that has to refer to the already known value of a variable will use the name of this variable, without prime, in any side of the transition.
The TLA translation of the parallel composition of A and B is the conjunction of $TLA(A)$, $TLA(B)$ and some extra terms accounting for extra initial constraints or taking ownership of variables. The TLA translation of the parallel composition is as follows:

$$\begin{array}{lcl} \Theta(Par) & \triangleq & \Theta(A) \cup \Theta(B) \cup \Theta \\ Init(Par) & \triangleq & Init(A) \wedge Init(B) \wedge Init \\ Mod(x, Par) & \triangleq & Mod(x, A) \vee Mod(x, B) \\ TLA(Par) & \triangleq & Init(Par) \wedge TLA(A) \wedge TLA(B) \wedge (\bigwedge_{\theta \in \Theta} \theta' \neq \theta \Rightarrow Mod(\theta, Par)) \end{array}$$

The TLA translation of sequentially composed roles, which we omit here for space reasons, is analogous. One must augment the translation with an auxiliary flag recording which of the two roles, A or B, is executing and take into account the acceptance conditions. Once A has reached an accepting state, we toggle the flag to indicate that B should begin execution.

4. Experimental Modelling Results

HLPSL has already proven itself to be an expressive and versatile language for modelling security protocols. While we restrict ourselves to the example of NSPK in this paper for brevity's sake, HLPSL has been used to formalise protocols that are significantly more complex, both in terms of agent behaviour and in terms of their intended security goals. All modelled protocols have, in turn, been analysed with the AVISPA tool.

Candidate protocols for formalisation were selected in collaboration with representatives from the IETF (see [4]). Although, in this paper, we focus on HLPSL itself and not on analysis results, we note that our experimentation with the AVISPA tool has also discovered both new and previously known attacks on some of the candidate protocols.

The industrial protocols we have modelled in HLPSL include TLS, Kerberos, various versions of EKE, UMTS-AKA, IKEv2 and AAA for Mobile IP. Such protocols have been studied with respect to a variety of security properties (described in more detail in [4]) such as entity and message authentication, replay protection, authorisation (by a trusted third party), key authentication and confirmation, and confidentiality. In addition to the protocols mentioned above, we have also modelled a range of different EAP protocols (see `http://www.ietf.org/html.charters/eap-charter.html`). We note that each of the goals shown is expressible as a safety temporal property like those already discussed.

Industrial-scale protocol suites are composed of smaller sub-protocols whose interaction should allow for the achievement of a variety of complex security goals. The modelling of such protocols is considerably simplified by HLPSL's modularity. Depending on the current state of the protocol, one sub-protocol can be executed either in place of another or loop until a particular condition is satisfied. Here, the flow control and composition constructs of HLPSL are most useful when modelling protocols like, for instance, Kerberos [16].

Other protocols modelled, in particular those intended to be executed in a mobile setting like those of the Mobile IP suite [7], can greatly benefit from the flexible way in which HLPSL allows one to change the channel and intruder models for the analysis of the protocol in a variety of different settings.

5. Related Work

Several specification languages have been used for protocol analysis. They include domain-specific languages specifically designed for protocols analysis (like HLPSL itself) as well as more generic modelling languages (e.g. LOTOS [18] or PROMELA, the input language to the SPIN [15] model checker). An exhaustive survey is beyond the scope of this paper, but we present a brief comparison of HLPSL to a selection of the most closely related work.

A powerful protocol specification language must fulfil several requirements. Among them we count

the ability to explicitly specify different channel types (i.e. different intruder models). CASPER [14], CAPSL [11], and MuCAPSL [23] (an extension of CAPSL based on strand spaces and particularly suited for group protocols) leave the intruder model implicit, while in HLPSL the intruder model can be specified as a parameter of the `channel` type (where each intruder model corresponds to a set of axioms that define its behaviour), allowing the modeller to describe heterogeneous network settings.

Another important issue in protocol design is the capability to model a wide range of goals. The aforementioned languages restrict themselves to secrecy and authentication goals, whereas HLPSL's underlying logic enables us to model very general security goals in a flexible way using temporal logic formulae.

Moreover, the possibility to specify compound keys allows the protocol designer to model a wide range of protocols, for example those based on the Diffie-Hellman exponentiation. Like CASPER, Winskel's SPL [9], a language based on Petri nets semantics, provides neither this feature nor the possibility to employ control-flow patterns.

6. Conclusion

The decision to base HLPSL on TLA affords 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.

As indicated in Section 4., the HLPSL language has already proven itself to be an effective language for modelling security protocols: many protocols of varying levels of complexity – from the NSPK example presented here to more complex industrial-scale protocols such as IKE and TLS – have already been formalised in HLPSL. Features like modularity, control flow patterns, the specification of alternative intruder models, and the generality of temporal-logic based goals give the protocol specifier great flexibility both to construct faithful models and to experiment with different assumptions about the environment in which the protocol should be executed.

In our experience, we have found that HLPSL is powerful yet readable and intuitive to work with. The fact that users from varied backgrounds, including students, have found HLPSL easy to use testifies to the language's accessibility, which was one of our primary design objectives from the outset.

Acknowledgements. We would like to express sincere thanks to Luca Viganò, Daniel Plasto, and David von Oheimb for their helpful feedback and assistance during the preparation of this paper.

References

[1] A. Armando, D. Basin, M. Bouallagui, Y. Chevalier, L. Compagna, S. Mödersheim, M. Rusinowitch, M. Turuani, L. Viganò, and L. Vigneron. The AVISS Security Protocol Analysis Tool. In E. Brinksma and K. G. Larsen, editors, *Computer-Aided Verification CAV'02*, LNCS 2404, pages 349–354. Springer-Verlag, Heidelberg, 2002. URL of the AVISS and AVISPA projects: `www.avispa-project.org`.

[2] A. Armando, L. Compagna, and P. Ganty. SAT-based Model-Checking of Security Protocols using Planning Graph Analysis. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Proceedings*

of the 12th International Symposium of Formal Methods Europe (FME), LNCS 2805, pages 875–893. Springer-Verlag, 2003. www.avispa-project.org/publications.html.

[3] AVISPA. Deliverable 2.3: The Intermediate Format. www.avispa-project.org/delivs/2.3, 2003.

[4] AVISPA. Deliverable 6.1: List of selected problems. www.avispa-project.org/delivs/6.1, 2003.

[5] D. Basin, S. Mödersheim, and L. Viganò. An On-The-Fly Model-Checker for Security Protocol Analysis. In E. Snekkenes and D. Gollmann, editors, *Proceedings of ESORICS'03*, LNCS 2808, pages 253–270. Springer-Verlag, 2003. www.avispa-project.org.

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

[7] P. Calhoun, J. Loughney, E. Guttman, G. Zorn, and J. Arkko. RFC 3588: Diameter Base Protocol, Sept. 2003. Status: Proposed Standard.

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

[9] F. Crazzolara and G. Winskel. Events in security protocols. In *Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 96–105. ACM Press, 2001.

[10] S. Creese, M. Goldsmith, B. Roscoe, and I. Zakiuddin. The attacker in ubiquitous computing environments: Formalising the threat model. In *Proc. of the 1st Intl Workshop on Formal Aspects in Security and Trust*, pages 83–97, Italy, 2003.

[11] G. Denker and J. Millen. CAPSL integrated protocol environment. In *DARPA Information Survivability Conference (DISCEX 2000)*, pages 207–221. IEEE Computer Society, 2000.

[12] G. Denker, J. Millen, A. Grau, and J. Filipe. Optimizing protocol rewrite rules of CIL specifications. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop (CSFW '00)*, pages 52–63. IEEE, July 2000.

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

[14] B. Donovan, P. Norris, and G. Lowe. Analyzing a Library of Security Protocols using Casper and FDR. In *Proceedings of the Workshop on Formal Methods and Security Protocols*, 1999.

[15] G. J. Holzmann. The Spin Model Checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[16] J. Kohl and C. Neuman. RFC 1510: The Kerberos Network Authentication Service (V5), Sept. 1993. Status: Proposed Standard.

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

[18] G. Leduc and F. Germeau. Verification of Security Protocols using LOTOS – Method and Application. *Computer Communications, special issue on "Formal Description Techniques in Practice*, 23(12):1089–1103, 2000.

[19] G. Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. In T. Margaria and B. Steffen, editors, *Proceedings of TACAS'96*, LNCS 1055, pages 147–166. Springer-Verlag, 1996.

[20] 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.

[21] G. Lowe. Casper: a Compiler for the Analysis of Security Protocols. *Journal of Computer Security*, 6(1):53–84, 1998.

[22] C. Meadows. The NRL Protocol Analyzer: An Overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

[23] J. Millen and G. Denker. MuCAPSL. In *DISCEX III, DARPA Information Survivability Conference and Exposition*, pages 238–249. IEEE Computer Society, 2003.

[24] 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.

[25] L. C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *Journal of Computer Security*, 6(1):85–128, 1998.

[26] D. Song. Athena: A new efficient automatic checker for security protocol analysis. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop (CSFW '99)*, pages 192–202. IEEE Computer Society Press, 1999.

[27] M. Turuani. *Sécurité des Protocoles Cryptographiques: Décidabilité et Complexité*. Phd, Université Henri Poincaré, Nancy, December 2003.