Web Services Verification and Prudent Implementation

Tigran Avanesov, Yannick Chevalier, Mohammed Anis Mekki, Michael
Rusinowitch

Web Services Verification and Prudent Implementation

T. Avanesov, Y. Chevalier, M. A. Mekki, M. Rusinowitch

LORIA & INRIA Nancy Grand Est, France
E-mail: FirstName.LastName@loria.fr

Abstract. Alice&Bob notation is widely used to describe conversations between partners in security protocols. We present a tool that compiles an Alice&Bob description of a Web Services choreography into a set of servlets. For that we first compute for each partner an executable specification as prudent as possible of her role in the choreography. This specification is expressed in ASLan language, a formal language designed for modeling Web Services tied with security policies. Then we can check with automatic tools that this ASLan specification verifies some required security properties such as secrecy and authentication. If no flaw is found, we compile the specification into Java servlets that real partners can use to execute the choreography.

Keywords: Web Services, Security Policy, Automated deployment

1 Introduction

Alice&Bob notation (A&B) primarily intended for specifying Security Protocols permits also to capture complex and secured standard conversations between Web Services (WS's). Figure 1 shows a choreography view of the *Needham-Schroeder Public Key* security protocol (NSPK). In the following we discuss how such conversations can be extracted from the specification files documenting the services and the processes involved in a choreography.

1.1 Web Services

According to the Web Services Description Language (WSDL) a WS is defined as a list of *operations* which can be called to use a computation or processing skill proposed by the WS. These operations are organized into *ports* and each port can be bound to one or different networks addresses (and possibly to specific transport protocols). Each operation is described by giving the precise XML structure of three messages: the request message the operation expects when it is called, the response message the operation returns as a result of a regular execution and the third is the error message returned after a faulty one. Moreover security policies are possibly defined to constrain the use of a service. They address different levels of protection targeting either the entire service, a specific port, a given operation or even a specific message associated to an
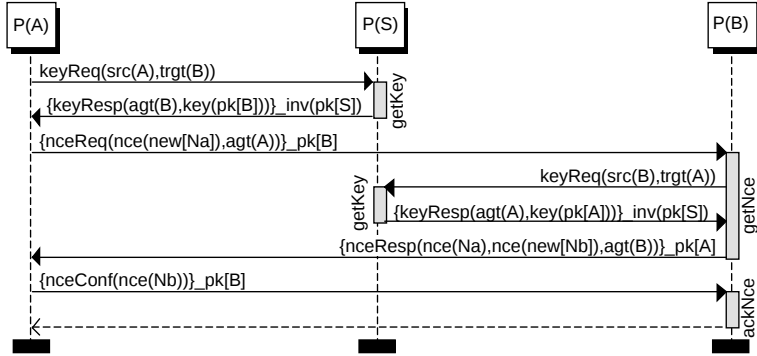
Fig. 1. The NSPK Choreography

operation. Within the WS-SecurityPolicy (WS-SP) standard, these policies can express either the need to use secure transport protocols to communicate with the service (*e.g.* SSL over HTTP) or the need for cryptographic protection for the operations' messages (using encryption or digital signature) but also for satisfying some access control rules by requiring some certificates to be present in the exchanged messages. Figure 2(a) shows a graphical representation of the *Key Server WS* (S). S exposes one operation called getKey which belong to a default port (not represented). We represent here the XML patterns bound to the input and output messages using first-order terms over a signature described in § 2.1. Moreover a security policy is associated to S and targets the outbound message of getKey which have to be digitally signed using a private key available to the service: #0. More generally such policies may specify which parts of one operation's message have to be digitally signed and/or ciphered. These parts are identified inside the corresponding XML schemata using XPATH [23] queries.

1.2 Business Processes

The *Business Process Execution Language* (*BPEL*) is an XML-based language describing *business processes* (BP's) as aggregation of WS's interactions achieving particular goals. While an *abstract* BPEL specification describes only the behavioral aspects of these interactions, *e.g.* following a specific order, an *executable* BPEL specification (also called WS *orchestration*) may describe in addition internal actions that are performed by the process, *e.g.* applying basic XML data transformation. An executable BPEL process can be executed by an orchestration engine which performs the WS's invocations and the internal actions in the order specified in its BPEL specification. The process itself can then be offered as a new WS, in this sense BPEL implements the composability principle of SOA by encompassing an aggregation of WS operation calls into a single one.

Figure 2(b) shows a graphical representation of $P(S)$ a BP playing the NSPK Choreography with partners $P(A)$ and $P(B)$. Each arrow represents a commu-
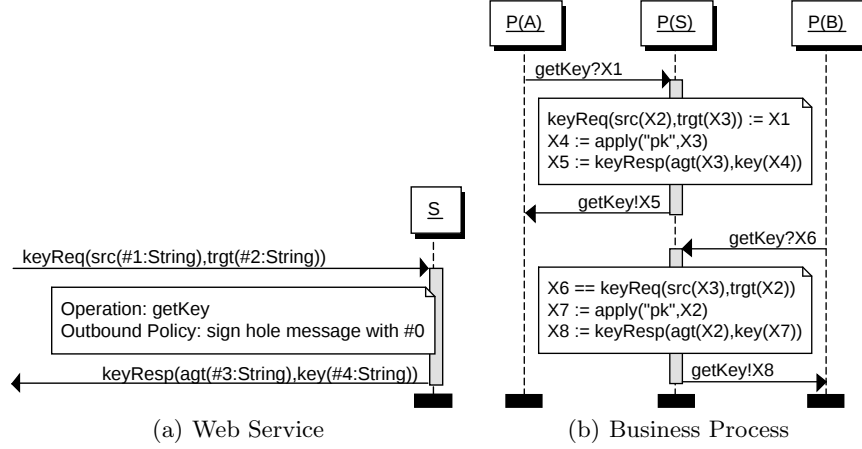
Fig. 2. The Key Server

nication activity involving $P(S)$. Annotations of the form getKey?Xi stands for receptions of requests Xi targeting the getKey operation of the service S while those of the form getKey!Xj stands for sending back the corresponding responses Xj returned by getKey to the caller. The Xi's are typed variables the orchestration engine manipulates during the process execution. These variables are either initially instantiated or assigned to values carried by received messages or to new values by applying simple XML transformations on already valuated variables. For example the line keyReq(src(X2),trgt(X3)) := X1 freshly assigns the variables X2 and X3 to the corresponding parts inside X1 which is implicitly supposed to be of the correct type. In the line X6 == keyReq(src(X3),trgt(X2)) all the variables are already valuated and thus it represents a condition to check. This check is important in order to play the NSPK Choreography correctly since $P(S)$ have to correlate the two requests *w.r.t.* the identities of the two partners $P(A)$ and $P(B)$. Correlation is explicitly supported by BPEL and naturally introduces the need of a multi-session handling which was not visible at WS level where operations are serving their answers in a synchronous way.

A BPEL orchestration engine is not in charge of enforcing the security policies associated to the invoked services or to the executed process. This is too restrictive from the general point of view of security protocols. Indeed it discards processes that receive messages they cannot totally decipher at the moment of reception but which could be more finely analyzed in a further step after receiving the right keys. The tool we propose in this paper overcomes this restriction and permits thus to correctly handle such protocols. For this we need to take into account the effect of the security policies on messages exchanged by the process and we sketch in the following how we can proceed. First we reconstruct from the BPEL process specification the terms t_i corresponding to the XML schematas of its exchanged messages m_i. Each t_i have its minimal subterms (or

leaves) ranging over a set of fresh variables. Then we identify in the t_i variables that correspond to the same value by analyzing the effect of the correlation relations and the internal actions (the assignments) of the process on the m_i. Finally we apply to each t_i the effect of the security policy attached to the operation of m_i. This amounts to first identify the subterms of t_i corresponding to parts in m_i that have to be cryptographically protected and then to replace them in t_i by terms expressing this protection. For $P(S)$ these transformations lead to the following sequence of interactions:

```
1 ?msg(par(A),opt(getKey),pld(keyReq(src(Y1),trgt(Y2))))
2 !msg(par(A),opt(getKey),pld({keyResp(agt(Y2),key(pk[Y2]))}_Y0))
3 ?msg(par(B),opt(getKey),pld(keyReq(src(Y2),trgt(Y1))))
4 !msg(par(B),opt(getKey),pld({keyResp(agt(Y1),key(pk[Y1]))}_Y0))
```

where $\{\#1\}\_\#2$ (resp. $\#1[\#2]$) is a shorthand for the asymmetric encryption of the message $\#1$ by the key $\#2$ (resp. a shorthand for apply("$\#1$",$\#2$)).

1.3 Choreographies

The choreography is a collaborative effort focused on exchange of messages from a global point of view where all participating services are treated equally [7]. This definition contrasts with the one for orchestration where a BPEL process is distinguished as a central coordinator of the WS's interactions. The *Web Services Choreography Description Language* (WS-CDL) [24] is a standard language that permits to define multi-party contracts describing only the externally observable behaviors of WS's the description of the messages they exchange. WS-CDL models are not meant to be directly executed but are often used in conjunction with executable BPEL models especially to implement choreographies. When this is the case, the WS-CDL specification may be used to distinguish the subset of the desired collaborations between a set of existing BPEL processes within the set of all the possible ones. Indeed WS-CDL permits to define execution in sequence, in parallel or in choice mode of structured and possibly (conditioned or repetitive) abstract interactions which can be matched with local communication activities of the involved BPEL processes.

For the NSPK Choreography the WS-CDL file simply interconnects in sequence the BPEL processes representing the involved partners: $P(A)$, $P(B)$ and $P(S)$. We claim that after collecting all the BPEL specifications involved in the NSPK Choreography (and enriched by the effect of security policies on the exchanged messages as explained in the end of § 1.2) and after analyzing the WS-CDL specification file we obtain the result in Figure 1 which is a graphical illustration of an A&B specification of the NSPK Choreography.

1.4 Contributions

We propose a tool that:

- translates an A&B specification of a WS choreography played by a set of BP's into its formal specification permitting its verification against classical security properties (such as secrecy, authentication);

– compiles the A&B specification into Java servlets that partners can use to execute the choreography as safely as possible.

We stress out here the fact that both results can be reused in the security protocol world, obtaining thus an automated procedure permitting to formalize protocols described using the A&B semi-formal notation but also to obtain their operational implementations. Moreover the obtained formal specification and Java servlets implement *as prudently as possible* the different business processes involved in the choreography permitting to establish strong security guarantees for the generated code (the *prudence* property is defined in § 3.1). Finally, we show how in some interesting cases the prudence property is sufficient to use the generated servlets in a multi-sessions-aware context.

The remainder of this paper is organized as follows: in § 2 we provide the different formal models considered by the tool. In § 3 we give the principle of our translation procedure which we illustrate through experimentation in § 4 and we finally conclude in § 5 by an overview of the related and prospective works.

2 Formal Models

2.1 Conversations

We use the A&B semi-formal notation to specify WS's *conversations*. A conversation is a sequence of messages exchanged by WS's recorded in the order they are to be sent [10]. We formally define it by the following grammar:

$$
\begin{aligned}
\mathit{Conversation} \ &::= \ \mathit{TypeDec}^* \ \mathit{Ik}^* \ \mathit{Com}^* \\
\mathit{TypeDec} \ &::= \ \textbf{Type} \ \mathit{Msg} : \mathit{Typ} \\
\mathit{Ik} \ &::= \ \mathit{Ag} \ \textbf{knows} \ \mathit{Term} \ (\ ,\mathit{Term})^* \\
\mathit{Com} \ &::= \ \mathit{Ag} - \mathit{Op} \rightarrow \mathit{Ag} : \mathit{Msg} \\
\mathit{Typ}, \mathit{Msg}, \mathit{Ag}, \mathit{Op} \ &::= \ \mathit{Term}
\end{aligned}
$$

Each line in the conversation is either an initial knowledge declaration *ik* for a partner, a communication activity *Com* where two partners exchange a message via a given operation or a type declaration *TypeDec* binding some message to a type. We consider three basic types: *message* for general character strings, *bool* for booleans and *nat* for integers. Partners, operations and messages are represented here by the non-terminal symbol *Term* corresponding to first order terms as defined below.

Terms We consider an infinite set of free constants C, an infinite set of variables $\mathcal{X}$ and the set of function symbols defined below:

$$
\begin{aligned}
\mathcal{F} = \ &\left\{ node_l^n, child_{\frac{i}{l}}^n \mid i \leq l \in \mathbb{N}^*, n \in \mathrm{C} \right\} \\
&\cup \{ apply, scrypt, sdcrypt, crypt, dcrypt, inv, invtest \}
\end{aligned}
$$

We denote by $\mathrm{T}(\mathcal{F})$ (resp. $\mathrm{T}(\mathcal{F}, \mathcal{X})$) the set of terms over $\mathcal{F} \cup \mathrm{C}$ (resp. $\mathcal{F} \cup \mathrm{C} \cup \mathcal{X}$). The former is called the set of ground terms (or messages) over $\mathcal{F}$,

while the latter is simply called the set of terms over $\mathcal{F}$. A substitution σ is an idempotent mapping from $\mathcal{X}$ to $T(\mathcal{F}, \mathcal{X})$ such that $\{x \mid \sigma(x) \neq x\}$ is a finite set. The application of a substitution σ to a term t (resp. a set of terms E) is denoted $t\sigma$ (resp. $E\sigma$) and is equal to the term t (resp. E) where all variables x have been respectively replaced by the term $x\sigma$.

An *equational theory* $\mathcal{E}$ is defined by a set E of equations $u = v$ with $u, v \in T(\mathcal{F}, \mathcal{X})$. We write $s =_{\mathcal{E}} t$ as the congruence relation between two terms s and t. Terms are manipulated by applying *operations* on them. These operations are defined by the set $\mathcal{F}_p = \mathcal{F} \setminus \{inv\}$ called the *set of public symbols*. Their semantics are defined with the $\mathcal{E}_{XML}$ equational theory defined above:

$$\begin{cases} sdcrypt(y, scrypt(y, x)) = x & (D_s) \\ dcrypt(y, crypt(inv(y), x)) = x & (D_{as}) \\ child_{\frac{i}{l}}^n(node_l^n(x_1, \ldots, x_l)) = x_i & (P_{\frac{i}{a}}) \\ invtest(x, inv(x)) = \top & (I_v) \end{cases}$$

A context $C[x_1, \ldots, x_n]$ is a term in which all symbols are public and such that its nullary symbols are the variables $x_1, \ldots, x_n$. $C[x_1, \ldots, x_n]$ is also denoted C when there's no ambiguity and n is called its *length*. We define the *application* of a context C of length n over the sequence of messages $m_1, \ldots, m_n$ to be the image of $C[X_1, \ldots, X_n]$ by the substitution $\{X_j \rightarrow m_j\}_{1 \leq j \leq n}$.

XML messages An XML node is represented by a term $node_l^n(u_1, \ldots, u_l)$ where $node_l^n$ is a function symbol of arity l, n is the tag of the node, and the u_i are terms representing the children of the node. In order to model security constraints holding over XML messages, we represent the usual cryptographic primitives with the function symbols: $scrypt/sdcrypt$ for symmetric encryption and decryption, $crypt/dcrypt$ for asymmetric encryption and decryption (used also to represent digital signature and its verification), inv to denote key inverses and $invtest$ permitting to test whether a pair of terms $\{t, t'\}$ verifies $t' = inv(t)$. Finally we use the function symbol $apply$ to represent the application of a given function represented by a term f to a term t by the term $apply(f, t)$.

2.2 Communication Scenarios

We now focus on the local behaviors of all partners involved in the choreography. We call such a behavior a *partner* and use *strands* [17] a standard notion in cryptographic protocol modeling to describe it. These notions are defined below.

Definition 1. *A* strand *s is a finite sequence of messages each labelled with* ! *or* ?. *The size of the sequence is called the* length *of s and messages with label* ! *(resp. ?) are said to be "sent" (resp. "received"). A strand is* positive *if and only if all its labels are* ?. *For any strand $s = \frac{!}{?}m_1, \ldots, \frac{!}{?}m_n$, the* length *of s denoted by $length(s)$ is n.*

Given a strand $s = \frac{!}{?}m_1, \ldots, \frac{!}{?}m_n$ we also denote in the following respectively by s^i and s_i the prefix of length i of s and the message m_i.

Definition 2. *A* partner *is a triplet* $(n, \mathcal{IK}, s)$ *where* n *is a constant called the* name, $\mathcal{IK}$ *is a set of messages called the* initial knowledge *and* s *is a strand called the* communication scenario.

Given a conversation we proceed by *end-point projection* [11] to collect the local behaviors of all the involved partners. This amounts to collecting first all the participating partner names and their respective initial knowledge from the conversation's *Ik* items. Then an iteration through all the *Com* items permits to build the communication scenarios as follows: if *Com* is $p_1 - o \rightarrow p_2 : m$ then append to the communication scenario of p_1 and p_2 respectively $!msg(par(p_2), opt(o), pld(m))$ and $?msg(par(p_1), opt(o), pld(m))$, where $node_3^{msg}$, $node_1^{par}$, $node_1^{opt}$ and $node_1^{pld}$ are wrapping function symbols permitting to hold the other partner's name in the communication activity, the called operation in addition to the exchanged message (the payload). Since our objective is to provide an executable implementation for a given role we need to keep track of these information for all its communication activities in order to compute the network address of the receiver in the case of an emission or the called operation in the case of a reception

For example, given the conversation of Fig. 1, our algorithm computes a partner $P(S) = (S, \{pk[S], inv(pk[S])\}, s)$ where s is the strand:

```
1  ?msg(par(A),opt(getKey),pld(keyReq(src(A),trgt(B))))
2  !msg(par(A),opt(getKey),
3     pld({keyResp(agt(B),key(pk[B]))}_inv(pk[S])))
4  ?msg(par(B),opt(getKey),pld(keyReq(src(B),trgt(A))))
5  !msg(par(B),opt(getKey),
6     pld({keyResp(agt(A),key(pk[A]))}_inv(pk[S])))
```

2.3 The ASLan Language

ASLan [3] (see also Appendix 6.1) is a formal language for specifying security-sensitive service-oriented architectures, the associated security policies, as well as their trust and security properties. We translate partners into ASLan roles where an ASLan role is defined by a transition system and an initial state. States are sets of facts, where facts are first order terms over a given signature. By convention, some facts in the states are used to hold the knowledge of a role r: $state\_r(step, knowledge)$ whereas some others are used to signal a new message emission: $iknows(scrypt(k, m))$. A transition τ is defined by two sets of facts: its left and right hand sides denoted respectively by $LHS(\tau)$ and $RHS(\tau)$. A transition τ may bring some state s to a state s', if $LHS(\tau)$ can be unified with a subset of s (via some substitution σ). Then s' is defined to be $(s\sigma \setminus LHS(\tau)\sigma) \cup RHS(\tau)\sigma$. The language allows also to guard the transitions by conditions like equality $(equal(t, t'))$ between terms (t, t').

3 Compilation Procedure

We present in the following a compilation procedure to translate partners to
ASLan roles. In § 3.1 we give an execution model for partners then in § 3.3 we
provide a concrete solution to generate their executable implementations and
finally in § 3.3 we show how these implementations are expressed in ASLan.

3.1 Execution model

To provide an executable implementation of Partner p we need to analyze which
values can be extracted by p from her received messages and specify how these
values should be processed by her to construct the next messages to be sent. Since
conversations involve cryptographically-protected messages one has to take into
account information asymmetries introduced by this protection. For example if
two partners exchange a cipher-text $scrypt(k, t)$, they may be seeing it differently
if only one of them detains the deciphering key k. Since one of our objectives is
to provide an executable implementation for a given partner p this point turns
out to be critical in our case. Note that this aspect was not considered by related
works like [11] on end-point projection.

Our approach can be seen as an application of [15] where informally speak-
ing an *implementation of a communication scenario* specifies the set of public
symbols a partner has to apply to its current knowledge in order to construct the
messages she has to send according to the scenario. A *prudent implementation*
performs in addition all possible checks or security verifications by analyzing and
correlating all received messages by her at that point.

We illustrate these notions by commenting a prudent implementation of the
partner $P(S)$ given below (and where pi−n is a shortcut for $child^n_{\bar{i}}$):

```
1 Step.0={X0:=pk; X1:=Ks; X2:=inv(Ks)}
2 Step.1={receive(X3) on getKey;
3    X4:=p1−keyReq(X3); X5:=p1−src(X4);
4    X6:=p2−keyReq(X3); X7:=p1−trgt(X6);
5    X8:=src(X7); X9:=trgt(X5); X10:=keyReq(X8,X9); X11:=agt(X7);
6    X12:=apply(X0,X7); X13:=key(X12); X14:=keyResp(X11,X13);
7    X15:=crypt(X2,X14); X16:=agt(X5); X17:=apply(X0,X5);
8    X18:=key(X17); X19:=keyResp(X16,X18); X20:=crypt(X2,X19)}
9 Step.2={send(X15) to X5}
10 Step.3={receive(X21) on getKey; X21?=X10}
11 Step.4={send(X20) to X7}
```

For example, in Step.0, $P(S)$ initializes her knowledge by the function symbol
pk and her public and private keys. Step.1 corresponds to her first reception
targeting the operation getKey and where the received message is stored in the
variable $X3$. Lines $3 − 4$ describe how parts are extracted from $X3$. For example
in line 4, $X6$ is assigned the second child of $X3$. All the messages that could be
further used by $P(S)$ are precomputed by her as soon as possible thus reducing
sending steps to emitting precomputed values (like in Step.2 and Step.4). We note

that $P(S)$ being in Step.1 is also able to precompute the message corresponding
to her next expected reception. Then in Step.3 the received message (stored in
$X21$) is compared with the precomputed expected value stored in $X10$. This
check is not necessary to the execution of $P(S)$, since $X20$ the message to be
sent by her before finishing her task is already precomputed. Ignoring this test
we still result in an executable implementation of $P(S)$ which is not prudent *i.e.*
one that does not check its input messages as thoroughly as possible.

3.2 Solution principle

A conversation does not explicitly specify what parts of a message being received
by a partner should be extracted by him and in which order. Similarly it does
not give information on how a message being sent by some partner is effectively
constructed given its current knowledge. The approach proposed in [15] is to
extract this information from a conversation and reduces the problem to classical
decision problems in protocol analysis, namely the *reachability* and the *finite
basis* problems. In this approach the conversation is projected on every partner
to generate her view. Then for each partner p (with s its communication scenario)
the following analysis is performed:

Sending case for all sent message m_i find a *context* C, *i.e.* a sequence of op-
erations that permit to construct m_i given all the messages received so far
$m_1, \ldots, m_{i-1}$. If such a context exists we say that m_i is *reachable* by p and
we call C an *extraction context* for m_i from the communication scenario s. We
then extend this definition to any subterm of s. If for all sent message m_i there
exists a context C_i such that $C_i[m_1, \ldots, m_{i-1}] =_{\mathcal{E}_{XML}} m_i$, s is said *executable*
and one is given an operational implementation of it, provided that all the public
symbols functions are implemented.

Reception case for all message m_i compute a finite generating set of all the pairs
of contexts C, C' such that the equality $C[m_1, \ldots, m_i] =_{\mathcal{E}_{XML}} C'[m_1, \ldots, m_i]$
holds. This finite set if it exists is called *the finite basis* of the communication
scenario s^i. If this computation succeeds for all received messages m_i then one
can obtain a prudent implementation of s by checking after each reception m_i
the messages relations specified by the finite basis.

3.3 Solving the problem

Our approach is an application of [15] to the $\mathcal{E}_{XML}$ equational theory and is
summarized below. Given a communication scenario s we first look for extraction
contexts for all subterms of s. Since the sent messages are particular subterms
we can obtain an implementation of s. For that we first define a reachability
relation.

Definition 3. *Given a term t a partner $p = (n, \mathcal{IK}, s)$ and a step i at most equal
to the length of s. We consider a list $\langle m_1, \ldots, m_l \rangle$ of messages in the union of*

$\mathcal{IK}$ and the set of all received messages in s^l. We say that t is reachable *by p at step i* if there is a context C such that $C[m_1, \ldots, m_l] =_{\mathcal{E}_{XML}} t$ (and also that C is an extraction context *for t from p*).

We also define the function *reach* as follows:

$$reach(p, t) = \{min(i) \mid t \text{ is reachable by } p \text{ at step } i\}$$

Given a partner p we compute all the subterms reachable by p at all steps of its communication scenario. For that we introduce the notion of sequents to store relations holding between subterms of p.

Definition 4. *Given a partner $p = (n, \mathcal{IK}, s)$ we call γ a sequent of p (and denote it by $t_1, \ldots, t_k \vdash_f t_0$) an equality $t_0 =_{\mathcal{E}_{XML}} f(t_1, \ldots, t_k)$ where f is a public symbol and $t_0, \ldots, t_k$ is a possibly empty sequence of subterms of s. We call respectively t_0, f and the sequence $t_1, \ldots, t_k$ the* head, *the* symbol *and the* tail *of γ and denote them respectively by $h(\gamma)$, $s(\gamma)$ and $t(\gamma)$.*

We say that γ is *valid* at some step i when for all $0 \le j \le k$, $reach(p, t_j) \le i$. We denote the set of all sequents of p by $\Gamma(p)$ and the set of all valid sequents at some step i by $\Gamma_i(p)$. If a sequent γ is valid at step i then its head is also reachable at step i by taking $f(C_1, \ldots, C_k)$ as an extraction context for t_0 from p, assuming that $C_1, \ldots, C_k$ are respectively extraction contexts for $t_1, \ldots, t_k$.

We now solve the reachability and the finite basis problems for a given partner $p = (n, \mathcal{IK}, s)$. First we compute $\Gamma(p)$ by running through all the subterms of s and collecting the corresponding sequents. For example a subterm of the form $t = scrypt(k, m)$ will provide two entries: $k, m \vdash_{scrypt} t$ and $k, t \vdash_{sdrcypt} m$. For each computed sequent γ we define an integer called its *readiness* and initially set to the size of $t(\gamma)$. This integer is used to compute the validity of a sequent as explained further in this paragraph. We also define for each subterm t of s a list of sequents $sequents(t)$ which is initialized by all the sequents γ' such that t appears in the tail of γ'. The detailed solution is illustrated by Algorithm 1 which relies on Algorithm 2 and both are given below.

Algorithm 1 Reachability

Require: $\mathcal{IK}, s$
1: **for all** $t \in \mathcal{IK}$ **do**
2: $\quad$ $deduce(t, 0)$
3: **end for**
4: **for all** $?m_k \in s$ **do**
5: $\quad$ $deduce(m_k, k)$
6: **end for**

Algorithm 2 *deduce*

Require: $t : subterm, i : step$
1: **if** $reach(p, t) > i$ **then**
2: $\quad$ $reach(p, t) \leftarrow i$
3: $\quad$ **for all** $\gamma \in sequents(t)$ **do**
4: $\quad\quad$ $\gamma.readiness - -$
5: $\quad\quad$ **if** $\gamma.readiness = 0$ **then**
6: $\quad\quad\quad$ $\Gamma_i(p).add(\gamma)$
7: $\quad\quad\quad$ $deduce(h(\gamma), i)$
8: $\quad\quad$ **end if**
9: $\quad$ **end for**
10: **end if**

The idea is to perform a fix-point computation per each step i corresponding to the set $\Gamma_i(p)$. We start from subterms that are trivially reachable at some given

step: elements in the initial knowledge which are all reachable at the initial step 0 and all the received messages clearly reachable at their corresponding reception step and try to deduce the newly reachable ones by checking whether there exists some sequents having their tails made only of reachable subterms. In order to select these sequents we make use of the *readiness* field attached to each sequent which is decremented each time one element in its tail is discovered to be reachable (Algorithm 2, line 4). Since the *readiness* field is initialized by the cardinality of its tail thus whenever $\gamma.readiness$ equals zero at some step then the sequent is also valid at that step.

First, Algorithm 1 solves the the reachability problem since the value of $reach(p, t)$ is computed for all subterms of s. Second the sets $\Gamma_i(p)$ computed for all $1 \leq i \leq length(s)$ permit one to build a finite basis for all prefixes s^j where s is the communication scenario of p, namely the sets $\{h(\gamma) \stackrel{?}{=} s(\gamma)(t(\gamma))|\gamma \in \Gamma_i(p)\}$ for all $1 \leq j \leq length(s)$. We emphasize here the redundancy of these finite basis, since $\Gamma_{i-1}(p) \subseteq \Gamma_i$ for all $1 \leq i \leq length(s)$. This means that all the checks corresponding to sequents that are valid at some step are performed again in all the subsequent steps. We let $new(p, i)$ be the subset of elements in $\Gamma_i(p) \setminus \Gamma_{i-1}(p)$ for all $1 \leq i \leq length(s)$. The set $new(p, i)$ contains sequents that are valid only at step i and can be used as an optimized finite basis for the prefix s^i provided that $new(p, k)$ is also used as a finite basis for s^k for all $1 \leq k \leq i - 1$.

Algorithm 1 runs in linear time in the size of p represented as a directed acyclic graph (DAG) since the number of instructions performed by the algorithm is proportional to the sum of the sizes of $\mathcal{IK}$ and s represented as DAGs.

ASLan generation We shortly describe our procedure and leave the technical details in Appendix 6.2. Given a partenr $p = (n, \mathcal{IK}, c)$ we propose to generate its corresponding ASLan role $r(p)$. First we associate a unique fresh variable name X_i for each reachable subterm t_i. These variables X_i will represent the parameters of the *state_n* fact. Then we define the initial state of $r(p)$ which will contain a *state_n* fact where all the variable X_j corresponding to values in $\mathcal{IK}$ are respectively replaced by these values. Finally we compute the transition system describing the communication behavior of $r(p)$. Each step is then translated to an ASLan transition that reflects the communication activity and the partner's knowledge evolution through the use of variables X_i while being possibly guarded by equality conditions involving these variables. We provide in Appendix 6.3 the ASLan specification obtained for the partner $P(S)$.

4   Experimental Results

This section is organized as follows: in § 4.1 we describe the deployment target for the prudent implementations obtained in § 3.3 then in § 4.2 we explain how our solution can be adapted for a multi-session setting and finally in § 4.3 we present an application of our approach for generating runnable Java distributions for mediators leading WS's orchestrations.

4.1 From ASLan to servlets

We propose to realize each partner by a java servlet partially described below.

```
1 class Partner extends HttpServlet{
2   String[] X; int step;
3   void dispatch(String msg){
4     if(accept(msg)){step++; sendRemaining();}
5   }
6 }
```

First the array X is a holder for all the variables X_i defining the state of the considered role and the step field is an integer pointing to the current execution step already reached by the role. The core of the servlet is the dispatch method which is the handler of the received messages. dispatch supplies the received message msg to the accept method (not represented) which returns a positive answer only when *(i)* the step field corresponds to a reception activity and *(ii)* msg matches the parsing and the possibly inherent checks for the step step. In this case accept updates the array X possibly with newly computed values before returning. Then dispatch increments the step counter and calls sendRemaining (not represented) which is in charge of sending any message required to be sent by the role till reaching the next reception step or the end of the communication scenario.

We note that the code for the accept method can be easily generated from the ASLan specification by transforming each equality condition appended to a reception transition either to the instantiation of a new variable or to the check of a certain relation between already valuated variables. For example the portion of code to be executed by the method accept at step=3 is the following:

```
1 if(msg.equals(X[10])){X[21] = msg; return(true);}
```

4.2 Multi-session handling

Let us remark that the servlet presented in § 4.1 is not able to handle different interleaved sessions of the corresponding role. We propose here to leverage this restriction by modifying the generated servlet which now will hold a list of pairs $(X, step)$ representing the different sessions' states. We have then to define a *dispatch strategy* telling the servlet how messages are routed to the right sessions. As discussed in [19] a naive strategy can be applied if one is assured that all messages received by a partner have a *distinguishing value* (uniquely originating nonces) that is reachable by the partner at the reception step and such that it unambiguously defines the targeted session. Given this assumption and the prudence of the implementation we provide, one can use a naive yet correct algorithm to dispatch messages: try all the sessions for consuming an incoming message. Assuming that every reception leads to computing a distinguishing value and that the implementation performs all the possible correlation checks upon receptions we can be sure that at most one session is candidate to consume an incoming message and we call this property *unique dispatch* (UD).

We finally note that the NSPK protocol does not have the UD property, since $P(S)$ receives messages that does not contain distinguishing values (even if all other partners do). Nevertheless if every partner plays his expected part in the interleaving of two sessions of the NSPK protocol then $P(S)$ still reaches a satisfactory state where she also ends up correctly her two sessions. Therefore we think our approach stays applicable and globally correct even if the distinguishing value assumption is relaxed.

4.3 Testing benchmark

In this section we describe a direct application of our approach for generating a prudent implementation for a *mediator* leading an *orchestration* between WS's.

In previous works [14, 13] we presented an automated approach for solving an orchestration problem which we informally define here as synthesizing a new service we call a *mediator* permitting to answer all the requests of a given client while relying on a community of available services. Services are represented as security protocol roles and the orchestration problem is then encoded as the reachability of a satisfying state of the client starting from a configuration containing the community of services and given Dolev-Yao intruder capabilities for the mediator. This problem can be solved by a variety of tools from the security protocol literature and we choose CL-Atse [22] (a constraint-solver) for this task. The result returned by CL-Atse contains an A&B description of the communication (the intruder acting as) the mediator has to perform with (a subset) of the community of WS's and the client to effectively answer all the requests of the latter.

We propose to reuse the approach presented in this paper to *(i)* generate an ASLan description of the mediator and *(ii)* generate the mediator's BPEL process and thus permit leading and using the newly discovered orchestration. Objective *(i)* has been already fulfilled and our approach has been implemented and integrated to the AVANTSSAR Validation Platform [5]. Concerning *(ii)* we recall that all the generated mediators could not be brought to standard implementations and we stress the fact that our approach could at least answer the problem in a standard way whenever this is possible.

We successfully assessed our tool against three case-studies from the test library [4] of the AVANTSSAR platform putting the focus on those representing orchestration problems. Our tool generated and deployed all the servlet needed by all the partners involved in each case-study and we effectively observed the conversation taking place between servlets as intended by the A&B specifications.

These case-studies tackle several aspects of security (*e.g.* probative value of digitally signed documents) and interoperability concerns and are summarized below:

- *Digital Contract Signing (DCS):* DCS represents a contract signing procedure carried out by two partners through secure access to a trusted third party web site, a business portal.

- *Public Bidding (PB)*: PB illustrates a secure document exchange, and aims at providing a web portal to manage an online call for tender, and also Bidders' proposal submissions.
- *Car Registration Process (CRP):* CRP models an e-government scenario, where a citizen have a secure access point, enabling communication with government offices and service providers in an easily usable and secure way.

Table 1 illustrates the obtained results for each case-study and for our running example where CS is the name of the case-study, $SIZE$ is the the sum of the sizes of all the partners, AGT (resp. SGT) is the time needed to generate the ASLan specification (resp. the Java servlets) for the case-study and finally AET (resp. SET) is the time needed to execute the generated ASLan specification (resp. the time needed by the generated Java servlets to end up a session).

Table 1. Tool Execution Times

CS	SIZE	AGT	SGT	AET	SET	CS	SIZE	AGT	SGT	AET	SET
NSPK	166	163 ms	1 s	708 ms	30 s	PB	596	4 s	2 s	18 m	1 m
DCS	340	659 ms	2 s	3 m	1 m	CRP	790	3 s	3 s	4 m	2 m

5   Conclusions

We present a tool that compiles an Alice&Bob description of a WS's choreography into their formal models enabling their automatic verification and if the models are secure into a set of java servlets that real partners can use to execute securely the choreography. We also developed a web application version of the tool that permits particularly to automatically deploy the servlets generated from an A&B specification provided through a web form.

5.1   Related works

Several works in the literature addressed the problem of translating A&B specifications of security protocols to formal languages in order to enable their verification or execution.

In [12] the authors translates A&B to CKT5 [9], a modal logic of communication, knowledge and time supporting symmetric encryption and thus adapted for security protocols. This permits to obtain a complete formal specification of the protocol, describing in particular the internal actions that should be taken by the involved partners to play their respective roles while enforcing a property equivalent to prudence [15].

In [21] the author translates a variant of the A&B notation defined over an arbitrary algebraic theory to the *IF* [6] formal language which is at the basis of ASLan. Using explicit destruction symbols this translation defines unambiguously how the protocol is supposed to be executed by honest agents according

to algebraic properties of the operators. In this sense, this work is very close to ours and we can reuse its results (in particular the decidability of the finite basis problem in presence of algebraic reasoning) to extend our approach to cover using the XOR operator or modular exponentiation.

In [8] the authors propose a high-level language for specifying multiparty sessions and a compiler translating these high-level specifications to the cryptographic protocols implementing them. The considered sessions specify patterns of the message exchanges between distributed partners and their data accesses to a common database while ignoring their local behaviors and internal actions. The compiler then adds custom cryptographic protections to ensure authentication, integrity and freshness of the exchanged messages and generates the (ML) code for sending and receiving messages while enforcing the required cryptographic operations and type checks against an active adversary. The main differences with our approach is that *(i)* we consider executable choreographies and generate ready to run programs implementing them and *(ii)* we do not add security protection to the messages exchanged within the choreography in order to enforce security properties but rather permit the modeler to check whether her existing message-level policies are sufficient to enforce these properties by providing a verifiable formal specification of the choreography.

In [20] the authors generate Java code to implement cryptographic network protocols specified in CAPSL [16]. The output includes code for each party and for a demonstration environment that permits user examination and control of communication. The generated code relies on Standard Java cryptographic providers (which at the time was not supporting public key cryptography).

In [19] the authors describe a compiler that uses constraint-based analysis to produce multi-session server programs from cryptographic protocols formalized in CPPL [18]. The compilation succeeds whenever each message received by an agent playing the protocol contains a distinguishing value that can be extracted by the agent at the moment of reception. We reuse their compilation success criteria to analyze whether all partners have the UDP property before safely using our naive dispatch strategy.

5.2 Future works

We conclude by a non-exhaustive list of possible extensions of the current work which we leave for future works:

- provide prudent implementations supporting WS standards. To be more expressive in this direction we consider also generating enhanced SOAP engines for partners that possibly mixes cryptographic treatment and XML parsing.
- extend the notion of prudent implementation to non-linear roles, *e.g.* by allowing roles to branch on dis-equalities between subterms;
- sharpen the cryptographic functions definitions, *e.g.* parametrize *crypt* by the algorithm used (RSA for the current version) and any other parameter like the key size (current RSA key default length is 2048);
- support more cryptographic primitives (*e.g.* XOR).

References

1. AVANTSSAR. Deliverable 2.1: Requirements for modelling and ASLan v.1. `www.avantssar.eu`, 2008.
2. AVANTSSAR. Deliverable 5.1: Problem cases and their trust and security requirements. `www.avantssar.eu`, 2008.
3. AVANTSSAR. Deliverable 2.3: ASLan final version with dynamic service and policy composition. `http://www.avantssar.eu/pdf/deliverables/avantssar-d2-3.pdf`, 2010.
4. AVANTSSAR. Deliverable 5.4: Assessment of the AVANTSSAR Validation Platform. Available at `http://www.avantssar.eu`, 2010.
5. AVANTSSAR. The AVANTSSAR Validation Platform. Available at `http://www.avantssar.eu`, 2010.
6. AVISPA. Deliverable 2.3: The Intermediate Format. `www.avispa-project.org`, 2003.
7. Alistair Barros, Marlon Dumas, and Phillipa Oaks. A Critical Overview of the Web Services Choreography Description Language (WS-CDL). BPTrends, 2005.
8. Karthikeyan Bhargavan, Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, and James J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium*, pages 124–140, Washington, DC, USA, 2009. IEEE Computer Society.
9. P. Bieber. A logic of communication in hostile environment. In *Computer Security Foundations Workshop III, 1990. Proceedings*, pages 14 –22, June 1990.
10. T. Bultan, J. Su, and X. Fu. Analyzing conversations of web services. *Internet Computing, IEEE*, 10(1):18 – 25, 2006.
11. Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In *Proceedings of the 16th European conference on Programming*, ESOP'07, pages 2–17, Berlin, Heidelberg, 2007. Springer-Verlag.
12. U. Carlsen. Generating formal cryptographic protocol specifications. In *Research in Security and Privacy, 1994. Proceedings., 1994 IEEE Computer Society Symposium on*, pages 137 –146, May 1994.
13. Y. Chevalier, M.A. Mekki, and M. Rusinowitch. Orchestration under security constraints. In *Sixth International Workshop on Formal Aspects in Security and Trust (FAST2009) Eindhoven, the Netherlands, November 5-6, 2009*, 2009.
14. Yannick Chevalier, Mohammed Anis Mekki, and Michaël Rusinowitch. Automatic composition of services with security policies. In *Proceedings of the 2008 IEEE Congress on Services - Part I*, SERVICES '08, pages 529–537, Washington, DC, USA, 2008. IEEE Computer Society.
15. Yannick Chevalier and Michaël Rusinowitch. Compiling and securing cryptographic protocols. *Inf. Process. Lett.*, 110(3):116–122, 2010.
16. G. Denker and J. Millen. CAPSL integrated protocol environment. In *DARPA Information Survivability Conference (DISCEX 2000)*, pages 207–221. IEEE Computer Society, 2000.
17. F.J.T. Fabrega, J.C. Herzog, and J.D. Guttman. Strand spaces: why is a security protocol correct? In *Security and Privacy, 1998. Proceedings. 1998 IEEE Symposium on*, pages 160–171, May 1998.
18. Joshua D. Guttman, Jonathan C. Herzog, John D. Ramsdell, and Brian T. Sniffen. Programming cryptographic protocols. In *In Trust in Global Computing*, pages 116–145. Springer, 2005.

19. J. McCarthy and S. Krishnamurth. Trusted multiplexing of cryptographic protocols. In *Sixth International Workshop on Formal Aspects in Security and Trust (FAST2009) Eindhoven, the Netherlands, November 5-6, 2009*, 2009.

20. J. Millen and F. Muller. Cryptographic protocol generation from CAPSL. Technical Report SRI-CSL-01-07, SRI International, December 2001.

21. Sebastian Mödersheim. Algebraic properties in alice and bob notation. In *ARES*, pages 433–440. IEEE Computer Society, 2009.

22. Mathieu Turuani. The CL-Atse Protocol Analyser. In *Term Rewriting and Applications (Proceedings of RTA'06)*, LNCS 4098, pages 277–286, 2006.

23. W3C Consortium. XML Path Language (XPath) 2.0 (Second Edition). `http://www.w3.org/TR/xpath20/`, 14 December, 2010.

24. World Wide Web Consortium. Web Services Choreography Description Language Version 1.0. `http://www.w3.org/TR/ws-cdl-10/`, 9 November, 2005.

6 Appendix

6.1 ASLan Overview

Background [1] ASlan (*AVANTSSAR Specification Language*) is defined by extending the *Intermediate Format* (IF) [6]. IF is an expressive language for specifying security protocols and their properties, based on multiset rewriting. As described in detail in [1], ASLan extends IF with a number of important features so as to express diverse security policies, security goals, communication and intruder models at a suitable abstraction level, and thereby allow for the formal specification and analysis of complex services and service-oriented architectures.

Most notably, ASLan extends IF with support of Horn clauses and LTL formulas. For instance invariants of the system can be defined by a set of (definite) Horn clauses. Horn clauses allow us not only to capture the deduction abilities of the attacker in a natural way, but also, and most importantly, they allow for the incorporation of authorization logics in specifications of services. Moreover, complex security properties can be specified in Linear Temporal Logic. As shown, for instance, in [2], this allows us to express complex security goals that services are expected to meet as well as assumptions on the security offered by the communication channels.

Syntax and Semantics Here, we recall the main features of ASLan, pointing the reader to [3] for more details on the language.

An ASLan file consists of several sections, among which:

Section Inits contains one or more initial states of the transition system. A state of a transition system is a set of variable-free facts.

Section Rules specifies the transitions of the transition system. A *transition* is a rule containing two parts, a left-hand side (LHS) and right-hand side (RHS). The rule can fire in a state whenever its LHS holds in that state. Moreover, a transition can be labeled with a list of existentially quantified variables whose purpose is to introduce new constants representing fresh data (e.g. nonces).

Example 1. Sample transition.

```
1 step sampleTransition ( BankAgent ) :=
2  state_BankingService ( BankAgent ,1) .
3  iknows ( request )
4  =>
5  state_BankingService ( BankAgent ,2) .
6  iknows ( response )
```

where

---

[1] excerpts from [3]

- step is a keyword used to define a new transition;
- sampleTransition is a transition name;
- BankAgent is a parameter of the transition;
- state_BankingService(BankAgent,1), iknows(request),
  state_BankingService(BankAgent,2), iknows(response) are facts;
- state_BankingService(BankAgent,1).iknows(request) is the LHS of the transition;
- state_BankingService(BankAgent,2).iknows(response) is the RHS of the transition.

This transition represents the behavior of a banking service that receives a request and then reacts by replying with a response and moving to another state. More precisely, the transition can be fired if there exists a value val of variable BankAgent such that state_BankingService(val,1) and iknows(request) are in the current state. The result of firing the transition is to replace the fact state_BankingService(val,1) by the fact state_BankingService(val,2) and add a new fact iknows(response).

Message sending and receiving are specified using iknows facts: the iknows in the LHS of a transition stands for receiving a message, while in the RHS of a transition it stands for sending a message. The fact iknows(request) of Example 1 will not disappear from the current state, because the predicate iknows is persistent: once a message is emitted, it becomes a part of the knowledge of the environment (i.e., of the network or of the intruder) and the environment does not "forget" it.

If the LHS of a transition holds in the current state, then it is assumed that the knowledge (represented by a set of ground facts) of the corresponding service is enough to build the messages stated in iknows in the RHS of the transition.

In order to specify service states, we use one predicate per service. By convention the predicate name starts with state_ followed by the service name, e.g. state_BankingService from Example 1.

Section Goals contains security goals that can be defined as attack states (special states of the transition system) or by means of LTL formulae.

Example 2. Sample attack state.

```
1    attack_state stateName(Msg):=
2        fact1(Msg).
3        fact2(Msg)
```

Here, attack state stateName is reached, if there exists a value val of variable Msg such that fact1(val) and fact2(val) are in the current state of the transition system.

Section HornClauses contains a finite set of Horn clauses. They can specify, for instance, the authorization logic.

6.2 ASLan generation details

We give here the details of the ASLan generation procedure introduced at the end of § 3.3.

For each partner $p = (n, \mathcal{IK}, c)$ we build a transition system specified in ASLan and representing a prudent implementation of p. First we consider a list $A = \langle a_1, \ldots, a_n \rangle$ of all the subterms of c reachable by p such that for all $1 \leq i < j \leq n$, $reach(p, a_i) \leq reach(p, a_j)$ and we associate a fresh variable name per each atom in A through a bijective mapping $\sigma^{-1} : a_i \mapsto X_i$. We then create a state fact $state\_n$ for p with the following profile type:

$$type(a_1) * \ldots * type(a_n) \rightarrow fact$$

For each reception $?msg(par(pa), opt(op), pld(m))$ in c we generate an ASLan transition τ having only the fact $iknows(\sigma^{-1}(m))$ in its RHS. We note that $\sigma^{-1}(m)$ is well defined, since every message m received by p is trivially reachable by her. For each emission $!msg(par(pa), opt(op), pld(m))$ in c we generate an ASLan transition τ having only the fact $iknows(\sigma^{-1}(m))$ in its LHS. Again we note here that if c is executable then every message m sent by the partner is reachable and thus $\sigma^{-1}(m)$ is well defined.

We introduce the variable renaming functions $\{VName_j\}_{1 \leq j \leq length(c)}$ to distinguish whether a value has been effectively assigned to the variable X_m or not yet in a transition. For each transition labeled by step j we respectively append to its LHS and RHS the facts $state\_n(\langle VName_{j-1}(X_i) \rangle_{1 \leq i \leq n})$ and $state\_n(\langle VName_j(X_i) \rangle_{1 \leq i \leq n})$ where the functions $VName_j$ map variables to ASLan variable names as follows:

$$VName_j(X_i) = \begin{cases} NI\_X_i, & \text{if } reach(p, \sigma(X_i)) \geq j; \\ X_i, & \text{otherwise.} \end{cases}$$

If the considered transition labeled by j corresponds to a reception, we append to its LHS all conditions of the form $equal(\sigma^{-1}(t_0), f(\sigma^{-1}(t_1), \ldots, \sigma^{-1}(t_k)))$ where $t_1, \ldots, t_k \vdash_f t_0 \in new(p, j)$. If the transition corresponds to an emission, we possibly guard it by conditions expressing the creation of a set of fresh nonces. These nonces correspond to all the values that are not reachable by the partner before the emission step and are bound in the transition to the set of variables X_k such that $reach(p, X_k) = j$. Finally we specify the initial state of the partner: $state\_n(\langle name(X_i) \rangle_{1 \leq i \leq n})$ where the functions $CName_j$ transform variables to ASLan constant names as follows:

$$CName_j(X_i) = \begin{cases} ni\_X_i, & \text{if } reach(p, \sigma(X_i)) > 0; \\ \sigma(X_i), & \text{otherwise.} \end{cases}$$

Informally speaking we initialize (with dummy values) the variables corresponding to atoms that does not appear in the initial knowledge of p but rather will be seen in received messages or generated as nonces in messages to be sent. On the other hand, variables corresponding to the atoms seen in the initial knowledge will be instantiated by the values specified by the modeler.

6.3 ASLan Specification of $P(S)$

```
 1 %%%%%%%%%%%%%%%%%%%
 2 section signature:
 3 %%%%%%%%%%%%%%%%%%%
 4 keyReq: message*message−>message
 5 p1_keyReq: message−>message
 6 p2_keyReq: message−>message
 7 src: message−>message
 8 p1_src: message−>message
 9 trgt: message−>message
10 p1_trgt: message−>message
11 keyResp: message*message−>message
12 p1_keyResp: message−>message
13 p2_keyResp: message−>message
14 agt: message−>message
15 p1_agt: message−>message
16 key: message−>message
17 p1_key: message−>message
18 msg: message*message*message−>message
19 p1_msg: message−>message
20 p2_msg: message−>message
21 p3_msg: message−>message
22 par: message−>message
23 p1_par: message−>message
24 opt: message−>message
25 p1_opt: message−>message
26 pld: message−>message
27 p1_pld: message−>message
28
29 state_S:
30   nat*hash_func*message*message*
31   keyReq(src(message),trgt(message))*
32   src(message)*message*
33   trgt(message)*message*
34   src(message)*trgt(message)*
35   keyReq(src(message),trgt(message))*
36   agt(message)*message*key(message)*
37   keyResp(agt(message),key(message))*
38   crypt(message,
39     keyResp(agt(message),key(message)))*
40   agt(message)*message*key(message)*
41   keyResp(agt(message),key(message))*
42   crypt(message,
43     keyResp(agt(message),key(message)))*
44   keyReq(src(message),trgt(message))
45 −>fact
46 %%%%%%%%%%%%%%%
47 section types:
48 %%%%%%%%%%%%%%%
```

```
49 execOK : message
50 getKey : message
51 pk : hash_func
52 ks : message
53 dummy : message
54 X0 : hash_func
55 X1 : message
56 X2 : message
57 X3 : keyReq ( src ( message ) , trgt ( message ) )
58 X4 : src ( message )
59 X5 : message
60 X6 : trgt ( message )
61 X7 : message
62 X8 : src ( message )
63 X9 : trgt ( message )
64 X10 : keyReq ( src ( message ) , trgt ( message ) )
65 X11 : agt ( message )
66 X12 : message
67 X13 : key ( message )
68 X14 : keyResp ( agt ( message ) , key ( message ) )
69 X15 : crypt ( message ,
70   keyResp ( agt ( message ) , key ( message ) ) )
71 X16 : agt ( message )
72 X17 : message
73 X18 : key ( message )
74 X19 : keyResp ( agt ( message ) , key ( message ) )
75 X20 : crypt ( message ,
76   keyResp ( agt ( message ) , key ( message ) ) )
77 X21 : keyReq ( src ( message ) , trgt ( message ) )
78 NI_X3 : keyReq ( src ( message ) , trgt ( message ) )
79 NI_X4 : src ( message )
80 NI_X5 : message
81 NI_X6 : trgt ( message )
82 NI_X7 : message
83 NI_X8 : src ( message )
84 NI_X9 : trgt ( message )
85 NI_X10 : keyReq ( src ( message ) , trgt ( message ) )
86 NI_X11 : agt ( message )
87 NI_X12 : message
88 NI_X13 : key ( message )
89 NI_X14 : keyResp ( agt ( message ) , key ( message ) )
90 NI_X15 : crypt ( message ,
91   keyResp ( agt ( message ) , key ( message ) ) )
92 NI_X16 : agt ( message )
93 NI_X17 : message
94 NI_X18 : key ( message )
95 NI_X19 : keyResp ( agt ( message ) , key ( message ) )
96 NI_X20 : crypt ( message ,
97   keyResp ( agt ( message ) , key ( message ) ) )
98 NI_X21 : keyReq ( src ( message ) , trgt ( message ) )
```

```
 99 %%%%%%%%%%%%%%%%
100 section inits:
101 %%%%%%%%%%%%%%%%
102 initial_state init :=
103  state_S(0,pk,ks,inv(ks),
104   keyReq(src(dummy),trgt(dummy)),
105   src(dummy),dummy,
106   trgt(dummy),dummy,
107   src(dummy),trgt(dummy),
108   keyReq(src(dummy),trgt(dummy)),
109   agt(dummy),dummy,key(dummy),
110   keyResp(agt(dummy),key(dummy)),
111   crypt(dummy,
112    keyResp(agt(dummy),key(dummy))),
113   agt(dummy),dummy,key(dummy),
114   keyResp(agt(dummy),key(dummy)),
115   crypt(dummy,
116    keyResp(agt(dummy),key(dummy))),
117   keyReq(src(dummy),trgt(dummy)))
118  .iknows(getKey)
119 %%%%%%%%%%%%%%%%
120 section rules:
121 %%%%%%%%%%%%%%%%
122 step step_01_S(X0,X1,X2,X3,X4,X5,X6,X7,
123  X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,
124  X18,X19,X20,NI_X3,NI_X4,NI_X5,NI_X6,
125  NI_X7,NI_X8,NI_X9,NI_X10,NI_X11,NI_X12,
126  NI_X13,NI_X14,NI_X15,NI_X16,NI_X17,
127  NI_X18,NI_X19,NI_X20,NI_X21) :=
128  state_S(0,X0,X1,X2,NI_X3,NI_X4,
129   NI_X5,NI_X6,NI_X7,NI_X8,NI_X9,
130   NI_X10,NI_X11,NI_X12,NI_X13,
131   NI_X14,NI_X15,NI_X16,NI_X17,
132   NI_X18,NI_X19,NI_X20,NI_X21)
133  .iknows(msg(par(X5),opt(getKey),
134    pld(X3)))
135 &equal(X4,p1_keyReq(X3))
136 &equal(X5,p1_src(X4))
137 &equal(X6,p2_keyReq(X3))
138 &equal(X7,p1_trgt(X6))
139 &equal(X8,src(X7))
140 &equal(X9,trgt(X5))
141 &equal(X10,keyReq(X8,X9))
142 &equal(X11,agt(X7))
143 &equal(X12,apply(X0,X7))
144 &equal(X13,key(X12))
145 &equal(X14,keyResp(X11,X13))
146 &equal(X15,crypt(X2,X14))
147 &equal(X16,agt(X5))
148 &equal(X17,apply(X0,X5))
```

```
149  &equal (X18, key (X17))
150  &equal (X19, keyResp (X16, X18))
151  &equal (X20, crypt (X2, X19))
152    =>
153   state_S (1 ,X0,X1,X2,X3,X4,X5,X6,X7,X8,
154    X9,X10,X11,X12,X13,X14,X15,X16,X17,
155    X18,X19,X20,NI_X21)
156 step  step_02_S (X0,X1,X2,X3,X4,X5,X6,X7,
157  X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,
158  X18,X19,X20,NI_X21)  :=
159   state_S (1 ,X0,X1,X2,X3,X4,X5,X6,X7,X8,
160    X9,X10,X11,X12,X13,X14,X15,X16,X17,
161    X18,X19,X20,NI_X21)
162    =>
163   state_S (2 ,X0,X1,X2,X3,X4,X5,X6,X7,X8,
164    X9,X10,X11,X12,X13,X14,X15,X16,X17,
165    X18,X19,X20,NI_X21)
166  . iknows (msg ( par (X5), opt (getKey), pld (X15)))
167 step  step_03_S (X0,X1,X2,X3,X4,X5,X6,X7,
168  X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,
169  X18,X19,X20,NI_X21 ,X21)  :=
170   state_S (2 ,X0,X1,X2,X3,X4,X5,X6,X7,X8,
171    X9,X10,X11,X12,X13,X14,X15,X16,X17,
172    X18,X19,X20,NI_X21)
173  . iknows (msg ( par (X7), opt (getKey), pld (X21)))
174  &equal (X10 ,X21)
175    =>
176   state_S (3 ,X0,X1,X2,X3,X4,X5,X6,X7,X8,
177    X9,X10,X11,X12,X13,X14,X15,X16,X17,
178    X18,X19,X20,X21)
179 step  step_04_S (X0,X1,X2,X3,X4,X5,X6,X7,
180  X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,
181  X18,X19,X20,X21)  :=
182   state_S (3 ,X0,X1,X2,X3,X4,X5,X6,X7,X8,
183    X9,X10,X11,X12,X13,X14,X15,X16,X17,
184    X18,X19,X20,X21)
185    =>
186   state_S (4 ,X0,X1,X2,X3,X4,X5,X6,X7,X8,
187    X9,X10,X11,X12,X13,X14,X15,X16,X17,
188    X18,X19,X20,X21)
189  . iknows (msg ( par (X7), opt (getKey), pld (X20)))
190  . iknows (execOK)
191 %%%%%%%%%%%%%%%
192 section  goals :
193 %%%%%%%%%%%%%%%
194 attack_state  executionOK ():=
195  iknows (execOK)
```