Rule-based Programs describing Internet Security Protocols*

Yannick Chevalier and Laurent Vigneron

LORIA - UHP-UN2-CNRS
Campus Scientifique, B.P. 239
54506 Vandœuvre-lès-Nancy Cedex, France
{chevalie,vigneron}@loria.fr

Abstract. We present a low-level specification language used for describing real Internet security protocols. Specifications are automatically generated by a compiler, from TLA-based high-level descriptions of the protocols. The results are rule-based programs containing all the information needed for either implementing the protocols, or verifying some security properties. This approach has already been applied to several well-known Internet security protocols, and the generated programs have been successfully used for finding some attacks.

1 Introduction

Internet is becoming everyday a more widely used medium for electronic commerce. This development is hampered by the natural insecurity of communications, as it is not possible to guarantee that some data exchanged is not listened by someone else, or even that it really originated from the claimed sender. This lack of security leads to the development of *security protocols*, that is small messages sequences, after which the author provides some properties to the user, such as the correct identification of the users (called *agents*) and the privacy of some data pieces.

There has been a significant amount of work toward the specification of security protocols in the recent years [23,1,17,7,19]. However, a large part of this work, including our own, is applied only to *toy protocols* in the *Alice&Bob* notation, *i.e.* as a linear scenario describing the messages exchanged.

Our main goal is to successfully handle complex protocols such as those under discussion at the IETF [18]. To this end, a new High-Level Protocol Specification Language (HLPSL) was developed in the AVISPA project, having in mind the constructions often found in the specification of these protocols [8]. We have written a compiler transforming a protocol specification in this language to a set of rewrite rules. We present in this article not the compiler itself, but the encoding of high-level properties in rewrite rules. We believe that rule-based systems are a natural framework to encode the properties encountered when studying cryptographic protocols.
We do not discuss in this paper about verification methods. For more information concerning them, see [12,10] for instance.

This paper is organized as follows: we first describe shortly the high-level specification language (Section 2); in Section 3, we describe how the initial specification is translated into a rule-based program, corresponding to a low-level specification; then, we list the examples of real Internet security protocols that have already been successfully compiled (Section 4), their rule-based specification being used by several verification tools. In the conclusion, we

* This work was partially funded by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2001-39252 AVISPA project, and the RNTL 03V360 Prouvé project.

compare our compiler with the MuCAPSL-MuCIL translator [13], based on the powerful language MuCAPSL [20].

2 Specifying Protocols and Intruder

In this section, we first present our objectives concerning protocols specifications, recalling what should be the properties of specification languages. Such objectives are achieved by the use of two languages: a high-level language that we describe shortly (see [8] for a complete description and semantics), to be used by protocol designers; and a low-level language (described in a further section) to be used by engineers.

2.1 Specification Languages

Studying security protocols is a very important domain nowadays. This is often done in a three steps process. First, protocols are specified in a high-level, easily understandable language. Then, this specification is analyzed to ensure that there are no trivial errors. If no flaws are found, the protocol is verified in a time-consuming last step. We are interested in this paper in the translation from the high-level language used in the first step to a language suitable for analysis.

A lot of high-level specification languages have been defined, some very simple (such as those based on the Alice&Bob notation [7,19]) and some dedicated to a specific tool [17,23]. But all of them either have a very limited expressiveness, or need a high level of expertise, or both.

Our aim in studying security protocols is based on the following objectives concerning the specification language: we want to consider real Internet protocols and to define a language that can be used by industrials; this language has to be able to express many security properties and has to have a clear semantics; in addition, it has to provide a basis for automated analysis.

These objectives are motivated by the fact that protocols specifications have to be used as documentations: in general, protocols are described in long documents (for example provided by the IETF); this makes them difficult to understand, and may lead to different interpretations according to the objectives of the reader (to implement the protocol, to verify it, or simply to understand it). Moreover, the underlying scientific foundations have to be clear in a protocol specification. This is very important for knowing if the protocol is easy to implement or not.

For summing up, the requirements for a high-level protocol specification language are:

- Simplicity and comprehension: specifications have to be easy to write, to read and to understand.
- Flexibility: a modification in the protocol should not mean to rewrite the whole specification.
- Non-ambiguity: the semantics of the language should be clear enough for avoiding ambiguous interpretations.
- Modularity: a specification has to be modular; this permits to share some modules between several protocols, and possibly to hide some parts of the protocol.
- Expressiveness: this is the most difficult criteria to satisfy; it can be decomposed into the following points:

- Control flow: the language has to provide some primitives for controlling the reception and emission of messages, to describe a negotiation phase, for instance.
- Knowledge of the intruder and agents: the user has to be able to manage the knowledge of the participants to the protocol, and that of the intruder.
- Cryptographic primitives: the language has to permit the use of fresh information, such as nonces (*numbers used once*) or keys, of hash functions, and also of signatures.
- Complex initial states: sometimes, protocols do not start from scratch, and assume that some preliminary actions have already been done; so the language has to permit the use of complex initial states.
- Complex message types: in most languages messages are generally built with simple primitives, such as encryption, decryption, pairing, but some more complex data structures may be needed for describing internal data structures or messages of roles (e.g. sets, lists, records).
- Algebraic properties: in some protocols, the mechanisms for encryption and decryption or for creating keys is details and involves the use of algebraic operators, such as exclusive-or or exponentiation; such operators have to be recognized by the specification language, because they satisfy some properties that may be considered for implementing the protocols.

Because none of the existing languages satisfies all these crucial requirements, we have decided, with our partners of the AVISPA project (Siemens AG Münich, DIST Genova and ETH Zürich), to define a new specification language.

2.2 An Expressive Specification Language

We will illustrate our new specification language [8] with the well-known Needham and Schroeder Public Key (NSPK) protocol. This example is usually considered as very simple and far away from real protocols. But the version that can be seen in most papers is a simplified one. Our aim being to consider all the options that may rise during the execution of a protocol, we will consider a more complex variant of the NSPK protocol: the NSPK Key Server (NSPK-KS). This protocol is given as follows, using an Alice&Bob-based notation:

$$\begin{array}{l} \textit{if } A \textit{ does not know } K_B, \\ \qquad A \to S: \quad A, B \\ \qquad S \to A: \quad \{B, K_B\}_{K_S^{-1}} \\ A \to B: \quad \{N_A, A\}_{K_B} \\ \textit{if } B \textit{ does not know } K_A, \\ \qquad B \to S: \quad B, A \\ \qquad S \to B: \quad \{A, K_A\}_{K_S^{-1}} \\ B \to A: \quad \{N_A, N_B\}_{K_A} \\ A \to B: \quad \{N_B\}_{K_B} \end{array}$$

The main originality of this protocol is that agents A and B, needing to know the public key of each other for running NSPK, can ask it to a server S if they do not already have it. This means that some steps of the protocol are conditional.

Such a protocol is impossible to specify in other high-level dedicated protocol specification languages, because none of them permits to easily define such guarded transitions.

Modular specification using roles. Our specification language is modular: protocols are not given as a sequence of messages, but as a set of roles. There are basic roles, each one representing the behavior of one agent in the protocol. There are also composed roles, representing the composition of other roles or their instantiations to be considered.
Informally, basic roles correspond to an Alice&Bob description with control; composed roles correspond to the use of CSP-like operators.

In our example, there are three basic roles: Alice, initiator of the protocol and named A; Bob, responder, named B; and Server, S.

Each role is an independent process, with external information given as parameter, and a local environment.

```
role Server(S: agent, Ks: public_key,
            KeyMap: (agent.public_key) set,
            SND,RCV: channel (dy)) played_by S def=
  local A: agent, B: agent, Kb: public_key
  knowledge(S) = { inv(Ks) }
  transition
    step0. RCV(A'.B') /\ in(B'.Kb', KeyMap)
      =|> SND({B'.Kb'}inv(Ks))
end role
```

The local environment is given by a list of local variables, and a list of knowledge (for example, S knows the inverse of the public key Ks, i.e. the corresponding private key).

The messages are exchanged via channels (SND and RCV), parameterized by their level of security that corresponds to the model of the intruder to be used for them: dy stands for the standard Dolev-Yao model [15] (no specific protection); ota stands for the over-the-air model (no diverted message). For a role, there may be several channels for sending and receiving messages, depending on their security level and on the concerned agents.

Composed roles are used for describing how to combine roles: this is possible to run roles in parallel or in sequence. For example, in the following NSPK role, Alice and Bob roles are run in parallel, and in as many instances as required. The Server role does not appear in this composed role because there will be only one, and it will be launched in another composed role, in parallel with NSPK.

```
role NSPK(SC, RC, S_SRV, R_SRV: agent -> channel (dy),
          Ks: public_key,
          Instances: (agent.agent.public_key.public_key) set,
          KeySet: agent -> (agent.public_key) set) def=
  composition
    /\_{ in(A.B.Ka.Kb, Instances) }
          Alice(A,B,Ka,Ks,KeySet(A),SC(A),RC(A),S_SRV(A),R_SRV(A))
          /\ Bob(A,B,Kb,Ks,KeySet(B),SC(B),RC(B),S_SRV(B),R_SRV(B))
end role
```

Control flow: guarded transitions. The main part of a role is the description of a transition system. Its originality is that transitions are not ordered: they are of the form condition =|> action, where condition and action are multisets of facts; a transition can

be applied as soon as its condition is satisfied. So, in a role, several transitions can be applicable at the same time.
We illustrate the use of guarded transitions with the role Alice.

```
role Alice(A,B: agent, Ka,Ks: public_key,
           KeyRing: (agent.public_key) set,
           SND_B,RCV_B,SND_S,RCV_S: channel (dy)) played_by A def=
   local State: nat, Na: text(fresh),
           Nb: text, Kb: public_key
   init State = 0
   knowledge(A) = { inv(Ka) }
   transition
      step1a.  State = 0
            /\ in(B'.Kb', KeyRing)
            /\ RCV_B(start)
          =|> State' = 2
            /\ SND_B({Na'.A}Kb')
      step1b.  State = 0
            /\ not(in(B.Kb',KeyRing))
            /\ RCV_B(start)
          =|> State' = 1
            /\ SND_S(A.B)
      step2.   State = 1
            /\ RCV_S({B.Kb'}inv(Ks))
          =|> State' = 2
            /\ KeyRing' = cons(B.Kb',KeyRing)
            /\ SND_B({Na'.A}Kb')
      step3.   State = 2
            /\ RCV_B({Na.Nb'}Ka)
          =|> State' = 3
            /\ SND_B({Nb'}Kb)
end role
```

The condition can contain comparisons, Boolean expressions over lists or sets, messages receptions. The action can contain messages sendings, assignments of variables.
A transition is a change of state, *primed variables* representing the values of the variables in the next state. So primed variables can be assigned in the right-hand side of transitions. However, if the new value of a variable is learned in the left-hand side of a transition (in a received message, in a comparison, or in a set expression, for examples), then its primed name is used (see for example step0 of role Server).

The definition of roles is based on a rich type system. Many types are available for describing protocols: agent, channel, text, message, public key, symmetric key, Boolean, integer, hash function, enumeration. Some variables of these types may be “fresh”, i.e. their value is generated at running time; this happens when the primed variable appears only in the right-hand side of a transition, not assigned.
This is also possible to use type constructors: function, pair, list, set. And some algebraic operators can be used for representing cryptography properties: *xor*, *exp*.

Verification of properties. In the specification, this is possible to precise a goal section, indicating a list of properties to be checked. The supported properties are:

- secrecy: some information (keys, nonces, messages,...) have to remain secret, i.e. an intruder should not get this information;
- authentication: two roles identify each other w.r.t. an information that they send to each other; this property exists in two versions: weak authentication and strong authentication, the second one proposing a protection against the replay of a protocol.

These properties are kinds of macros, without the need to add some information in the transitions of roles.
This is however possible to specify LTL formulas that will not be interpreted by the compiler, and to add some user-defined facts in the transitions that will be carried by the compiler.

A more complete description of this high-level language and of its semantics as TLA formulas is given in [8].

3 Towards a Rule-based Program

Specifications of protocols are compiled into a rule-based program. During this compilation phase, the syntax and the semantics of the initial specification are verified. If some goals are specified, the compiler can either generate one program containing all the properties to be checked, or it can generate one program for each goal.

The generated program contains basically three parts: rules describing the intruder's behavior; rules describing role transitions and compositions[1]; the initial state, describing the instances to be considered of the protocol.

All this information is divided in several files: a prelude file containing all the protocol independent information, and at least one protocol specific file. These files represent a complete and detailed low-level specification of the initial protocol, where all variables and constants are typed: this is a rule-based program, a rule being of the form:

step rule_name (list_of_variables_involved) :=
    left_hand_side => right_hand_side

The left-hand side and right-hand side of a rule are multisets of terms. So, the multiset constructor '.' is associative and commutative. This permits to handle the *non-determinism* when matching the current state of the protocol against the left-hand side of a rule. This also permits to consider the run in *parallel* of several instances of the protocol [19].

3.1 General Information

The prelude file contains all the general information necessary for obtaining a self-contained program. This information is divided into several sections that we are going to describe.

[1] Note that currently we only consider parallel compositions; sequential compositions are accepted in the hight-level specifications, but not yet converted into rules.

Type symbols. The list of type names is given in this section.

section typeSymbols:
agent, text, symmetric_key, public_key, function, table,
message, fact, nat, set, protocol_id

Signature. This section contains the subtyping information; for example, agents, keys and nonces are subtypes of messages.

section signature:
message > agent
message > nonce
message > symmetric_key
message > public_key

In addition, each primitive used for constructing messages is declared, such as:

pair : message ∗ message −> message
crypt : message ∗ message −> message
scrypt : message ∗ message −> message
inv : message −> message
apply : message ∗ message −> message

corresponding to pair construction, asymmetric and symmetric encryption of a message with a key, key inverse (this is a notation, not an applicable algorithm), and function application to a message.
More advanced primitives are also declared, such as intruder's knowledge, belonging constraints, and goal facts:

iknows : message −> fact
contains : message ∗ message −> fact
secret : message ∗ agent −> fact
witness : agent ∗ agent ∗ protocol_id ∗ message −> fact

Declaration of variables. The type of each variable used in this prelude file (see the following sections) is declared in this section.

section types:
F,K,M,M1,M2,M3 : message

Note that all of them are declared of type message for sake of generality. For example, K is used as a key, but in case of symmetric encryption it could be a compound message.

Equational properties. Protocols specifications are often based on some hypotheses over the message construction or the cryptography. This section permits to list the equational properties considered. For example, messages are built by concatenating sub-messages, forming tuples. But for a more simple representation in the rules, tuples will be represented by pairing; this choice is correct if pairing is associative.
Another example concerns the keys used for encrypting messages: given a public (resp. private) key k, its corresponding private (resp. public) key is denoted $inv(k)$; a consequence is that the inverse of the inverse of a key is the key itself.

section equations:

```
pair(M1,pair(M2,M3)) = pair(pair(M1,M2),M3)
inv(inv(K)) = K
```

For some protocols, the perfect cryptography hypothesis[2] is relaxed by describing how to generate some keys using the Diffie-Hellman exponentiation [21,9],

```
exp(exp(M1,M2),M3) = exp(exp(M1,M3),M2)
exp(exp(M1,M2),einv(M2)) = M1
```

or by describing the encryption mechanism using exclusive-or.

```
xor(M1,xor(M2,M3)) = xor(xor(M1,M2),M3)
xor(M1,M2) = xor(M2,M1)
xor(M,M) = 0
xor(0,M) = M
```

Intruder model. The intruder is described by a set of messages that it knows and by rules over this set. We describe the behavior of an intruder, following the standard Dolev-Yao model [15], independently of the protocol considered. This general behavior is first the ability to generate messages from its knowledge:

section intruder:

```
step gen_pair (M1,M2) :=
    iknows(M1).iknows(M2) => iknows(pair(M1,M2))
step gen_crypt (M1,M2) :=
    iknows(K).iknows(M) => iknows(crypt(K,M))
step gen_scrypt (M1,M2) :=
    iknows(K).iknows(M) => iknows(scrypt(K,M))
step gen_apply (M1,M2) :=
    iknows(F).iknows(M) => iknows(apply(F,M))
```

The intruder may also analyze messages in its knowledge, for trying to get new information by decomposing them, if possible:

```
step ana_pair (M1,M2) :=
    iknows(pair(M1,M2)) => iknows(M1).iknows(M2)
step ana_crypt (K,M) :=
    iknows(crypt(K,M)).iknows(inv(K)) => iknows(M)
step ana_scrypt (K,M) :=
    iknows(scrypt(K,M)).iknows(K) => iknows(M)
```

Finally, the intruder is able to generate fresh information. This is described by the following rule, where the left-hand side is empty, the right-hand side corresponds to the addition of a message M in the intruder's knowledge, and the arrow of the rule contains the information that M has to be generated at running time: its value has to be an unused value of the type of M.

```
step generate (M) :=
    =[exists M]=> iknows(M)
```

All this information in independent of the protocol to be considered. This independence guarantees an objective and general description of the intruder's behavior.

[2] An encrypted message can only be decrypted by the adequate key.

3.2 Protocol Information

We describe in this section how a high-level specification is translated into a rule-based program, and illustrate it with the NSPK-KS protocol.

The high-level specification of a protocol is mainly a list of roles of two kinds: basic roles, each one representing the behavior of a participant; composed roles, describing the environment of the basic roles, i.e. how to compose them and which instantiations to consider.

In the resulting program, a basic role, which is initially presented as a module, is then considered as a state. The environment roles will permit us to generate initial role states; the transitions in a basic role will describe how to change the state of that role.
What was the local environment of a basic role becomes a list of parameters of the state. However, in the current version of the compiler, only one kind of channels is considered: channel on which the Dolev-Yao model of the intruder is applied. So, for avoiding useless complex notation in the generated rules, all sent messages are directly added to the knowledge of the intruder (iknows(...)).

Signature. The generated program contains a section with the signature of each role state primitive, representing the internal data structure of the role. Note that the first argument of a role state is the name of its player. This information may be useful for tools that have to use this program, in particular if they want to manage the knowledge of the agents.

section signature:

```
state_Bob: agent * agent * public_key * public_key * set * nat * text * text
    * public_key * nat -> fact
state_Alice: agent * agent * public_key * public_key * set * nat * text * text
    * public_key * nat -> fact
state_Server: agent * public_key * set * agent * agent * public_key * nat
    -> fact
```

In those role states, natural numbers are used as labels for distinguishing steps, and they are also used for ensuring the uniqueness of agents.

Declarations. Then, all the variables and constants used in the program are declared. Note that, as in the high-level language, variables always start with a capital letter, and constants with a small letter or with a digit.

section types:

```
nb, na : protocol_id
kb, ka, ks, ki, Ka, Kb, Ks, Dummy_Ka, Dummy_Kb, dummy_pk : public_key
CID, CID2, CID1, State, 0, 1, 2, 3, 4, 5, 6 : nat
Nb, Na, Dummy_Nb, Dummy_Na, dummy_nonce : text
MGoal, KeySet, start : message
AGoal, b, a, s, A, B, S, i, Dummy_B, Dummy_A, dummy_agent : agent
Instances, KeyMap, KeyRing, local_62, local_89, local_104, local_111, local_116 : set
```

Initialization. The initialization of the protocol is put in a specific section. It contains the initial states of basic roles, obtained by flattening the composed roles; note that some

parameters of those states correspond to variables that were declared locally (and not initialized) in the roles, so they are initialized with specific constants (dummy_...). The knowledge of the intruder is also initialized in this section, using the knowledge declared for it in the composed roles, and the knowledge necessary for playing its assigned roles in the instantiations. The start message is also put in the intruder's knowledge.
A *state of the protocol* is a set of roles states and the set of knowledge of the intruder.
Note that in the generated program, the intruder is never assigned as player of a state role. This is due to the Dolev-Yao intruder model: each message sent by an agent is directly added to the knowledge of the intruder; so, an agent gets a message by taking it in the intruder's knowledge; and as the intruder is able to decompose and compose messages, it can build messages it is supposed to build when playing a honest role.

```
section inits:
  initial_state init1 :=
    contains(pair(i,ki),local_62).iknows(local_62).
    iknows(ki).iknows(inv(ki)).
    iknows(ks).iknows(a).iknows(i).
    iknows(start).
    state_Server(s,ks,local_89,dummy_agent,dummy_agent,dummy_pk,2).
    state_Alice(a,b,ka,ks,local_104,0,dummy_nonce,dummy_nonce,dummy_pk,4).
    state_Bob(b,a,kb,ks,local_111,0,dummy_nonce,dummy_nonce,dummy_pk,5).
    state_Alice(a,i,ka,ks,local_116,0,dummy_nonce,dummy_nonce,dummy_pk,6).
    contains(pair(a,ka),local_89).contains(pair(b,kb),local_89).
    contains(pair(i,ki),local_89).
    contains(pair(a,ka),local_104).
    contains(pair(b,kb),local_111).
    contains(pair(a,ka),local_116)
```

The initial state is therefore a term corresponding to the set of the initial role states and the set of the intruder's initial knowledge.

In the example given above, only constants are used. However, in the high-level specification, variables can be used for describing that some information is *shared* by several role states. For example, a will play twice the role Alice, and a variable could have been used for storing its key set. The translation of this in the initialization section would have been to used only one constant instead of local_104 and local_116.

Rules. The main section of the generated program is the section of rules. Each rule correspond to a state transition for one of the basic roles. For example, in NSPK-KS, the server has only one possible transition:

```
section rules:
  step step_0 (S,Ks,KeyMap,Dummy_A,Dummy_B,Dummy_Kb,A,B,Kb,CID) :=
    state_Server(S,Ks,KeyMap,Dummy_A,Dummy_B,Dummy_Kb,CID).
    iknows(pair(A,B)).
    contains(pair(B,Kb),KeyMap)
    =>
    state_Server(S,Ks,KeyMap,A,B,Kb,CID).
    iknows(crypt(inv(Ks),pair(B,Kb)))
```

In each rule, the left-hand side contains the general pattern of the role state and the facts representing conditions for firing the transition; the awaited message has to be in the intruder's knowledge. After automatic diversion, the reply is immediately put into the intruder's knowledge. So, in the right-hand side, there is the new role state plus some facts describing new knowledge for the intruder or the modification of the value of a complex variable (a set, for example).

The translation of step1b and step2 of role Alice generates the following two rules:

```
step step_2 (A,B,Ka,Ks,KeyRing,Na,Nb,Dummy_Kb,CID) :=
    state_Alice(A,B,Ka,Ks,KeyRing,0,Na,Nb,Dummy_Kb,CID).
    iknows(start).
    not(contains(pair(B,Kb),KeyRing))
    =>
    state_Alice(A,B,Ka,Ks,KeyRing,1,Na,Nb,Dummy_Kb,CID).
    iknows(pair(A,B))

step step_3 (A,B,Ka,Ks,KeyRing,Dummy_Na,Nb,Dummy_Kb,Na,Kb,CID) :=
    state_Alice(A,B,Ka,Ks,KeyRing,1,Dummy_Na,Nb,Dummy_Kb,CID).
    iknows(crypt(inv(Ks),pair(B,Kb)))
    =[exists Na]=>
    state_Alice(A,B,Ka,Ks,KeyRing,2,Na,Nb,Kb,CID).
    iknows(crypt(Kb,pair(Na,A))).
    contains(pair(B,Kb),KeyRing).
    secret(Na,A).secret(Na,B)
```

In the last one, the nonce Na has to be created at running time (this is a fresh information). The notation =[*exists* Na]=> means that a fresh value will have to be generated each time that this rule is applied.
In that rule, there are also terms for indicating that Na is supposed to remain secret, only shared by agents A and B. This information has been added because the secrecy of Na has been required as goal property in the high-level specification.
In both transitions, the old state contains some variables named Dummy_Kb, for example. Such variables capture the old value of the corresponding variable (e.g. Kb) when either a new value will be assigned in the new state (e.g. in step_3), or when the name of this variable has to be used in the transition without considering its value (e.g. in step_2).

Goals. The last section is devoted to the description of goal properties. For example, if the secrecy of a term has to be checked, the goal section will be:

```
section goals:
  goal secrecy_of (MGoal,AGoal) :=
    secret(MGoal,AGoal).
    iknows(MGoal).
    not(secret(MGoal,i))
```

This description means that the secrecy property is not satisfied if a message MGoal, declared as a secret shared by agent AGoal, is in the intruder's knowledge, and the intruder (whose name is i) is not supposed to share it.

Similar goals are automatically generated for describing authentication properties.

4 Use of the Resulting Rule-based Programs

In this section, we list some of the protocols that we have already been able to analyze with our compiler. We also cite the tools that are using the generated rule-based programs for trying to find some attacks.
Note that in this paper, we have not illustrated our compiler with one of the industrial security protocols cited in the following because most of them are complex, involving many roles. The NSPK-KS protocol has the advantage of being based on a protocol known by everybody, but this variant is an excellent example for illustrating the importance of the environment of each agent, and of the control that has to be set.
We also do not list the standard toy protocols of the Clark-Jacob library [11]; they can of course be considered by our compiler.

4.1 Already Handled Protocols

In order to assess the compilation process, we have built a list of protocols from various sources. The aim here is to demonstrate that our compiler is able to handle a wide variety of protocols, known as important in different areas. We have selected both low-level protocols, such as TLS, as well as high-level ones, such as UMTS-AKA. There are also protocols already recommended by the IETF as well as protocols still under work.

To handle these protocols does not mean just to be able to specify them in our high-level language. What we have detailed in the previous sections is that we have defined a methodology for analyzing such specifications and for translating them into low-level specifications. All the subsequently given protocols have been automatically compiled, their specification being as verbatim as possible from their informal definition.

Core security mechanism [6]. Transport Layer Security (TLS) [14]. This two-part protocol aims at providing low-level privacy and data integrity. We have currently modeled and compiled the TLS record protocol.

Useful but not core mechanism [6]. Kerberos, ChapV2. The first one is well-spread and well-known. ChapV2 is an extension by Microsoft of the CHAP protocol used for PPP authentication.

Authentication mechanism for the Internet. In the survey [22], several protocols were recommended for authentication with password over the Internet. Among these, we have already analyzed the protocols EKE, EKE2, SPEKE and the SRP protocols. Among them, the *EKE protocol family is a family of zero-knowledge protocols for authentication on a password. We have also successfully analyzed the UMTS-AKA (Authentication and Key Agreement UMTS) protocol. The SRP protocol was designed by *Siemens* and presented at the IETF.

Protocols under development. We also work, in conjunction with Siemens, on the analysis of protocols under development. Among these, we have already analyzed the AAA MobileIP protocol, which is a sub-protocol of Mobile IP. See [16] for a description of the protocol its goal. Note that the Mobile IP protocol is a collection of protocols in development since 1998. We hope to contribute to this development by speeding up part of its analysis. We have also analyzed the IKEv2 main protocol.

Other sources. Finally, we have considered protocols from different sources, such as the ISO/IEC public key protocols. We have also analyzed the two party signature RSA protocol [5].

4.2 Verifying the Rule-based Programs

The programs that have been generated for the protocols listed in the previous section have been studied by several tools:

- OFMC [4]: an on-the-fly model checker developed at ETH, Zürich;
- SATMC [3]: a SAT-based model checker developed at DIST, Genova;
- CL-Atse [24]: a constraint logic-based protocol analyzer developed at LORIA, Nancy.

The first results obtained are still preliminary ones, but some attacks have been found on several protocols, some of them being new ones (see [8]).

This connection of three very different tools, done for the AVISPA project, demonstrates the flexibility and expressiveness of the rule-based specifications of protocols that we generate automatically.

5 Conclusion

We have presented in this paper a low-level framework for expressing protocol specifications. The security protocols are initially described with a simple, flexible, modular, non-ambiguous and expressive high-level language [8]. The generated specifications are rule-based programs with very detailed information: a full typing of variables, constants and primitives; a precise description of role transitions and of the initial state; the independent description of the intruder's model.

The generated rules permit to consider both the parallelism of the agents, and the non-determinism when applying a rule [19].

The compiler described is written in OCaml and documented with OCamlWeb (140 pages). It has been defined for the AVISPA project, for considering real industrial Internet security protocols. The first experiments generate very good results, so we are going to continue its development for being able to handle even more protocols. This is a considerable improvement compared to the first compiler that was realized for the AVISS project [2], where only simple protocols could be considered [11].

The only other known attempt to go beyond Alice&Bob notation in a clear high-level language is the MuCAPSL-MuCil translator [13]. Initial specifications are written in MuCAPSL [20], a new version of the CAPSL language [7] dedicated to the specification of group communication protocols. While CAPSL was an Alice&Bob notation language, MuCAPSL is completely different: it is based on roles, and provides a large scale of primitives and data structures; each role is typed (a type contains some attributes, functions, ...) and contains a sequence of instructions, including DO...UNTIL loops.

Comparing MuCAPSL with our specification language, this is clear that MuCAPSL offers many more data structures, primitives and instructions. However, it is unclear how to specify that some roles share some information, or how many instances of a role are created, since there is no environment role and roles do not have parameters. In addition, with our language, the transitions are guarded and are not ordered: their application is not deterministic, and transitions can be applied several times if possible; this is not the case with

MuCAPSL. So, this is not clear how the example given in this paper, NSPK Key Server, could be specified in MuCAPSL. About properties to check, MuCAPSL, like our language, proposes secrecy and authentication.
The MuCAPSL-MuCIL translator generates a specification in MuCIL that contains the declaration of the used symbols, and a multiset of conditional rewriting rules.

For concluding, MuCAPSL-MuCIL and our compiler have both their own advantages, and if our specification language is more simple, this is because we have designed it with the objective to provide it to industrial partners. It is powerful enough for specifying rather easily most of the Internet security protocols. And the aim of our compiler is to provide rule-based specifications of protocols to industrials for helping them to implement the protocols that they design, and also for verifying those protocols, by plugging any kind of verification tool, as it has already been done with AVISPA for three very different tools.

Acknowledgements: we thank the referees for their relevant remarks that have helped us to improve this paper.

References

1. M. Abadi and A. D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. In *Fourth ACM Conference on Computer and Communications Security*, pages 36–47. ACM Press, 1997.
2. 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. Guldstrand Larsen, editors, *Computer-Aided Verification, CAV'02*, LNCS 2404, pages 349–354, Heidelberg, 2002. Springer. URL of the AVISS and AVISPA projects: `http://www.avispa-project.org`.
3. A. Armando and L. Compagna. Abstraction-driven SAT-based Analysis of Security Protocols. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, LNCS 2919, pages 257–271, Santa Margherita Ligure, Italy, May 2003. Springer.
4. D. Basin, S. Mödersheim, and L. Viganò. An On-The-Fly Model-Checker for Security Protocol Analysis. In Einar Snekkenes and Dieter Gollmann, editors, *Proceedings of ESORICS'03*, LNCS 2808, pages 253–270. Springer, 2003.
5. M. Bellare and R. Sandhu. The security of a family of two-party RSA signature schemes. Technical Report 2001/060, Cryptography ePrint Archive, 2001. `http://eprint.iacr.org/2001/060/`.
6. S. Belovin. Report of the iab security architecture workshop, April 1998.
7. Common Authentication Protocol Specification Language. `http://www.csl.sri.com/~millen/capsl/`.
8. Y. Chevalier, L. Compagna, J. Cuellar, P. H. Drielsma, J. Mantovani, S. Mödersheim, and L. Vigneron. An high level protocol specification language suited for industrial security-sensitive protocols. Technical report, DIST, Genova, Italy, 2004. Submitted.
9. Y. Chevalier, R. Küsters, M. Rusinowitch, and M. Turuani. Deciding the Security of Protocols with Diffie-Hellman Exponentiation and Products in Exponents. In *Proceedings of the Foundations of Software Technology and Theoretical Computer Science, FST TCS'03*, LNCS 2914, page ??? Springer, December 2003.
10. Y. Chevalier and L. Vigneron. Strategy for Verifying Security Protocols with Unbounded Message Size. *Journal of Automated Software Engineering*, 11(2):141–166, April 2004.
11. J. Clark and J. Jacob. A Survey of Authentication Protocol Literature: Version 1.0, 17. Nov. 1997. `http://www.cs.york.ac.uk/~jac/papers/drareview.ps.gz`.
12. G. Delzanno and P. Ganty. A Survey on the State-of-the-art Methods for the Automatic Verification of Security Protocols. In *Proc. 1st Workshop on Issues in Security and Petri Nets (WISP)*, Eindoven, The Netherlands, 2003.
13. G. Denker and J. Millen. Modeling Group Communication Protocols using Multiset Term Rewriting. In *4th Int. Workshop on Rewriting Logic and its Applications (WRLA 2002)*, Pisa, Italy, 2002. (invited talk). ENTCS vol. 71 (2003), Elsevier Science Publishers.
14. T. Dierks and C. Allen. RFC 2246: The TLS Protocol Version 1.0, January 1999. Status: Proposed Standard.

15. D. Dolev and A. Yao. On the Security of Public-Key Protocols. *IEEE Transactions on Information Theory*, 2(29), 1983.
16. S. Glass, T. Hiller, S. Jacobs, and S. Perkins. RFC 2977: Mobile IP Authentication, Authorization, and Accounting Requirements, October 2000. Status: Informational.
17. G. J. Holzmann. The Spin Model Checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
18. IETF: The Internet Engineering Task Force. `http://www.ietf.org`.
19. F. Jacquemard, M. Rusinowitch, and L. Vigneron. Compiling and Verifying Security Protocols. In M. Parigot and A. Voronkov, editors, *Proceedings of LPAR 2000*, LNCS 1955, pages 131–160. Springer, 2000.
20. J. Millen and G. Denker. MuCAPSL. In *DISCEX III, DARPA Information Survivability Conference and Exposition*, pages 238–249. IEEE Computer Society, 2003.
21. J. K. Millen and V. Shmatikov. Symbolic Protocol Analysis with Products and Diffie-Hellman Exponentiation. In *Proceedings of the 16th Computer Security Foundations Workshop (CSFW'03)*, pages 47–61. IEEE Computer Society Press, 2003.
22. E. Rescorla. A Survey of Authentication Mechanisms, October 2003. Draft. `http://www.ietf.org/internet-drafts/draft-iab-auth-mech-03.txt`.
23. A. W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *Proceedings of the 8th IEEE Computer Security Foundations Workshop CSFW'95*, pages 98–107. IEEE Computer Society Press, 1995.
24. M. Turuani. *Sécurité des Protocoles Cryptographiques : Décidabilité et Complexité*. Phd thesis, Université Henri Poincaré – Nancy 1, December 2003.