# Specifying Authentication Protocols Using ELAN

Horatiu Cirstea

# Specifying Authentication Protocols Using ELAN

Horatiu Cirstea
LORIA
615, rue du Jardin Botanique, BP 101
54602 Villers-lès-Nancy Cedex, France
http://www.loria.fr/equipe/protheo.html
email: {Horatiu.Cirstea,Claude.Kirchner}@loria.fr

November 22, 1999

**Abstract**

Programming with rewrite rules and strategies has been already used for describing several computational logics. In this paper is described the way the Needham-Schroeder Public-Key protocol is specified in ELAN, the system developed in Nancy for executing rewrite programs. The protocol aims to establish a mutual authentication between an initiator and a responder that communicate via an insecure network. The protocol has been shown vulnerable and a correction has been proposed by G. Lowe. The behavior of the agents and of the intruders as well as the security invariants the protocol should verify are naturally described by conditional rewrite rules. The application of the rewrite rules is controlled by strategies. Similar attacks to those already described in the literature have been discovered. We show how different strategies using the same set of rewrite rules can improve the efficiency in finding the attacks and we compare our results to existing approaches.

## 1 Introduction

The Needham-Schroeder public-key protocol [NS78] has been already analyzed using several methodologies from model-checkers like FDR [Ros94] to approaches based on theorem proving like NRL [Mea96]. Although this protocol is simple it has been proved insecure only in 1995 by G. Lowe [Low95]. After the discover of the security problem and the correctness proof of a modified version in [Low96] several other approaches have been used in order to discover the attack and obtain correct versions like [Mea96, Mon99, DMT98].

The protocol aims to provide mutual authentication between two agents communicating via an insecure network. The agents use public keys distributed by a key server in order to encrypt their messages. In this paper we consider a simplified version like in [Low95] and we assume that each agent knows the public keys of all the other agents but it does not know their private keys.

The protocol is described by defining the messages exchanged between the participants. Each agent sends a message and goes into a new state in which it eventually expects an acknowledgement message. We can thus say that the protocol consists in the sequence of states describing the agents and the communication network. Therefore it seems natural to use rewrite rules in order to describe the transition from one state to another and strategies in order to describe the way these rules are applied.

The rewriting logic ([MOM96]), a particular case of general logics ([Mes92]), provide a logical framework in which many other logics can be represented. Therefore, unlike other approaches used for analyzing authentication protocols, the rewriting approach does not use a fixed model of concurrency.

In order to describe a computational version of a certain logic we use computational systems that can express the proof calculus of the given logic. A computational system ([KKV95]) is a combination of a rewrite theory and a strategy describing the intended set of computations. Since three decades a lot of work has been done on rewriting and efficient implementation techniques both on sequential models and distributed models have been described (e.g. [Vit96]). These ideas are implemented in the language ELAN ([BKK+96]) which allows to describe computational systems.

In our approach the whole formalization is done as the same level: the state transitions of the agents and of the intruder as well as the invariants the protocol should satisfy are described by ELAN rewrite rules. Furthermore, by making the formalization executable we allow either to directly use the specification for analyzing the protocol or to replay attacks or scenarios proposed by a third party.

The section 2 presents the basic notions concerning the rewriting logic and the computational systems. The way these ideas are implemented in ELAN is then briefly presented. The protocol is described in Section 3 together with an attack and a corrected version. In section 4 is presented the formalization in ELAN of the Needham-Schroeder public-key protocol. The data structures and the rewrite rules are presented together with the strategies used for discovering the attack. Some optimizations are proposed and some considerations related to existing approaches are presented. The last section of the paper contains the conclusions and give further perspectives for this work.

# 2 General Setting

This section presents the main concepts of rewriting logic that is fully described in [Mes92, MOM96]. The computational systems are then defined as rewrite theories in rewriting logic together with strategies that guide the computations. This provides the semantical foundation of the ELAN system as an environment in which computational systems can be described and executed.

## 2.1 Rewriting Logic

We just briefly recall the basic notions that are consistent with [JK91, DJ90] to which the reader is referred for a more detailed presentation. The definitions below are given in the one-sorted case, the many-sorted and order-sorted cases can be given a similar treatment.

We consider a set $\mathcal{F}$ of ranked function symbols, a set $\mathcal{X}$ of variables and the set of terms $\mathcal{T}(\mathcal{F},\mathcal{X})$ built on $\mathcal{F}$ using the variables in $\mathcal{X}$. A *signature* in rewriting logic is a pair ($\mathcal{F}$, $E$) with $\mathcal{F}$ a ranked alphabet of function symbols and $E$ a set of $\mathcal{T}(\mathcal{F},\mathcal{X})$-equalities. Rewriting will operate on equivalence classes of terms modulo the set of equalities $E$ that are typically structural axioms.

A *rewrite theory* is a 3-tuple $\mathcal{R}$=($\mathcal{F}$,$E$,$R$) where $\mathcal{F}$ is a ranked alphabet of function symbols, $E$ is the set of $\mathcal{T}(\mathcal{F},\mathcal{X})$-equalities and $R$ a set of rewrite rules of the form $l \to r$ where $l, r \in \mathcal{T}(\mathcal{F},\mathcal{X})$, $l$ is a non-variable term and $Var(r) \subseteq Var(l)$.

In this simple presentation we ignore rule labels and we say that a rewrite theory $\mathcal{R}$ entails a sequent $[t]_E \to [t']_E$, fact that is denoted by $\mathcal{R} \vdash [t]_E \to [t']_E$ (or $[t]_E \xrightarrow{*}_{\mathcal{R}} [t']_E$), if $[t]_E \to [t']_E$ is obtained by a finite application of the deduction rules described in Figure 1.

**Reflexivity**

$$\text{For each } t \in \mathcal{T}(\mathcal{F},\mathcal{X}) \qquad \frac{}{t \to t}$$

**Congruence**

$$\text{For each } f \in \mathcal{F}_n \qquad \frac{[t_i]_E \to [t'_i]_E,\ i = 1..n}{[f(t_1,\ldots,t_n)]_E \to [f(t'_1,\ldots,t'_n)]_E}$$

**Replacement by**

$$\text{For each rule } l(x_1,\ldots,x_n) \to r(x_1,\ldots,x_n) \in R \qquad \frac{[t_i]_E \to [t'_i]_E,\ i = 1..n}{[l(t_1,\ldots,t_n)]_E \to [r(t'_1,\ldots,t'_n)]_E}$$

**Transitivity**

$$\frac{[t_1]_E \to [t_2]_E \quad [t_2]_E \to [t_3]_E}{[t_1]_E \to [t_3]_E}$$

Figure 1: The deduction rules of unconditional rewriting logic

## 2.2 Computational Systems

The rewriting logic is proposed in [MOM96] as a logical framework in which other logics, like equational logic and Horn logic, can be represented and as a semantic framework for the specification of languages and systems.

The operational semantics for rewrite theories are presented in [KKV95] where is shown how computations in a rewrite theory can mirror computations in various logical systems.

A *strategy* is a subset of the set of possible derivations of the current rewrite theory and is used to describe the computations one is interested in. The result of applying a strategy on a term is the set (possibly empty) of all terms that can be derived from the initial term using the strategy. We formally express this by giving a functional representation to the application of a strategy $\mathcal{S}$ on a term $t$:

$$\mathcal{S}(t) = \{[t']_E | \exists \pi \in \mathcal{S}, [t]_E \xrightarrow{\pi} [t']_E\}$$

A *computational system* is defined as a labeled rewrite theory $\mathcal{R}$=($\mathcal{F}$,$E$,$\mathcal{L}$,$R$) together with a strategy $\mathcal{S}$. [KKV95] gives examples of languages and systems designed in this formalism.

## 2.3 The ELAN environment

ELAN is a language for designing and executing computational systems. In ELAN, a logic can be expressed by specifying its syntax and its inference rules. The syntax can be described using mixfix operators and the inference rules are described by conditional rewrite rules. In order to guide the application of the rewrite rules strategies are introduced. A full description of the language and its implementation is given in [BKK+97] and a survey of several examples that have been developed using ELAN is presented in [BKK+96].

All rewrite rules are working on equivalence classes induced by the set of equations $E$ that in the case of ELAN is restricted to associativity and commutativity axioms, for the symbols defined to be associative-commutative.

A labeled rewrite rule in ELAN is defined as a pair of terms built on functional symbols and local variables and additionally it can be applied under some conditions and it can use some local assignments. The local assignments are let-like constructions that allow applications of strategies on some terms. The general syntax of an ELAN rule is:

$$[\ell] \quad l \Rightarrow r \qquad [\ \texttt{if}\ \ cond \quad | \quad \texttt{where}\ \ y := (S)u\ ]^*$$

and the rule is applied if the condition *cond* is satisfied and the strategy $S$ is successfully applied on the term $u$.

Several rules may have the same label, the resulting ambiguities being handled by the strategies. The rule label is optional. In this case it is the responsibility of the designer to provide a confluent and terminating set of unlabeled rewrite rules.

The application of the labeled rewrite rules is controlled by user-defined strategies while the unlabeled rules are applied according to a default normalization strategy. The normalization strategy consists in applying unlabeled rules at any position of a term until the normal form is reached, this strategy being applied after each reduction produced by a labeled rewrite rule.

The application of a rewrite rule in ELAN can yield several results due to the equational (associative-commutative) matching and to the `where` clauses that can return as well several results.

The non-determinism is handled mainly by two basic strategy operators: `dont care choose` (denoted $\texttt{dc}(s_1, \ldots, s_n)$) that returns the results of at most one non-deterministicly chosen unfailing strategy from its arguments and `dont know choose` (denoted $\texttt{dk}(s_1, \ldots, s_n)$) that returns all the possible results. A variant of the `dont care choose` operator is the `first choose` operator (denoted $\texttt{first}(s_1, \ldots, s_n)$) that returns the results of the first unfailing strategy from its arguments. The strategy operator `repeat*` applies sub-strategies in a loop until none of them is applicable and the operator `;` is used for the sequential composition of two strategies.

**Example 2.1** Let us consider the three rewrite rules $r1 : a \Rightarrow b$, $r2 : a \Rightarrow c$ and $r3 : c \Rightarrow d$ and the strategy $strat \Rightarrow dk(r1, r2); r3$.

When we apply this strategy on the term $a$ the strategy $dk(r1, r2)$ is first applied on $a$ and then $r3$ is applied on the result. The strategy $dk(r1, r2)$ applied on $a$ yields the two results $b$ and $c$. We try to apply $r3$ on $b$ but since this is not possible a backtrack is performed and $r3$ is applied on $c$ with the result $d$.

The syntactic definitions and the rewrite rules can be implemented in ELAN in a modular way. Each module represents a computational system that can be parameterized and combined with other modules in order to build the whole system.

The environment ELAN allows us to get the trace of the computations executed and to obtain statistics about the application of the rewrite rules. When the specification describing the Needham-Schroeder public-key protocol is executed and an attack is discovered, the detailed description of the attack can be obtained by analyzing the ELAN trace of the execution.

# 3 The Needham-Schroeder public-key protocol

The Needham-Schroeder public-key protocol [NS78] aims to establish a mutual authentication between an initiator and a responder that communicate via an insecure network. Each agent $A$ possesses a *public key* denoted $K(A)$ that can be obtained by any other agent from a key server and a *(private) secret key* that is the inverse of $K(A)$. A message $m$ encrypted with the public key of the agent $A$ is denoted by $\{m\}_{K(A)}$ and can be decrypted only by the owner of the corresponding secret key, that is by $A$.

In this paper we only consider the simplified version proposed in [Low95]. In this version we omit the steps describing the requests made by the agents to the server for the public keys of the other agents and the corresponding responses from the server. This corresponds to assuming that each agent knows at the beginning the public keys of all the other agents.

The protocol uses *nonces* that are fresh random numbers to be used in a single run of the protocol. We denote the nonce generated by the $A$ by $N_A$.

The simplified description of the protocol presented in [Low95] is:

1. $A \to B$: $\{N_A, A\}_{K(B)}$
2. $B \to A$: $\{N_A, N_B\}_{K(A)}$
3. $A \to B$: $\{N_B\}_{K(B)}$

The initiator $A$ seeks to establish a session with the agent $B$. For this $A$ sends a message to $B$ containing a newly generated nonce $N_A$ and its identity, message encrypted with its key $K(B)$. When such a message is received by the agent $B$, it can decrypt it and extract the nonce $N_A$ and the identity of the sender. The agent $B$ generates a new nonce $N_B$ and it sends it to $A$ together with $N_A$ in a message encrypted with the public key of $A$. When $A$ receives this response it can decrypt it and assumes that it has established a communication with $B$. The agent $A$ sends the nonce $N_B$ back to $B$ and when receiving this last message $B$ assumes that it has established a communication with $A$ since only $A$ could have decrypted the message containing $N_B$.

The main property expected for an authentication protocol like Needham-Schroeder public-key protocol is to prevent an intruder from impersonating one of the two agents.

The intruder is an user of the communication network and so, can initiate standard sessions with the other agents and can respond to messages sent by the other agents. The intruder can intercept any messages from the network and can decrypt the messages encrypted with its key. The nonces obtained from the decrypted messages can be used by the intruder for generating new messages. The intercepted messages that can not be decrypted by the intruder can be replayed as they are.

An attack on the protocol is presented in [Low95] where the intruder impersonates another agent $A$ in order to establish a session with an agent $B$. The attack involves two simultaneous runs of the protocol: one initiated by $A$ in order to establish a communication with the intruder $I$ and a second one initiated by $I$ that tries to impersonate $A$ (denoted by $I(A)$) in order to establish a communication with $B$.

The attack involves the following steps where $I.n$, $II.n$ represents steps in the first and second session respectively:

| | | |
|---|---|---|
| I.1. | $A \to I$ | : $\{N_A, A\}_{K(I)}$ |
| II.1. | $I(A) \to B$ | : $\{N_A, A\}_{K(B)}$ |
| II.2. | $B \to I(A)$ | : $\{N_A, N_B\}_{K(A)}$ |
| I.2. | $I \to A$ | : $\{N_A, N_B\}_{K(A)}$ |
| I.3. | $A \to I$ | : $\{N_B\}_{K(I)}$ |
| II.3. | $I(A) \to B$ | : $\{N_B\}_{K(B)}$ |

The agent $A$ tries to establish a session with the intruder $I$ by sending it a newly generated nonce $N_A$. The intruder decrypts the message and initiates a second session with $B$ but claiming to be $A$. The agent $B$ responds to $I$ with a message encrypted with the key of $A$ and the intruder intercepts and forwards it to $A$. $A$ is able to decrypt this last message and sends the appropriate response to $I$. The intruder can thus obtain the nonce $N_B$ and sends it encrypted to $B$. At this moment $B$ thinks that it has established a session with $A$ while this sessions has in fact been established with the intruder.

The correction proposed in [Low95] consists in adding the identity of the sender in the second message of the protocol that becomes:

1. $A \to B$: $\{N_A, A\}_{K(B)}$

2. $B \rightarrow A$: $\{N_A, N_B, B\}_{K(A)}$
3. $A \rightarrow B$: $\{N_B\}_{K(B)}$

The identity of the responder in the encrypted part of the message allows to the initiator to verify that the message comes from the appropriate agent and the attack presented above can be avoided.

# 4 Encoding the Needham-Schroeder public-key protocol in ELAN

In this section we give a description of the protocol in ELAN. The ELAN rewrite rules correspond to transitions of agents from one state to another after sending and/or receiving messages.

Although in [Low96] it has been shown that the results obtained on the protocol involving one initiator, one responder and one intruder can be generalized to an arbitrary number of agents and intruders we have chosen to use a variable number of agents. Therefore, the number of initiators and responders is not fixed in the ELAN specification and it should be given at execution time. This allowed us on one hand to show the expressiveness of an ELAN specification and on the other hand to compare the results, in terms of efficiency, with other approaches like $Mur\phi$ [DDHY92].

The strategies guiding the rewrite rules describe a form of model-checking in which all the possible behaviors are explored. The strategies can be easily modified in order to obtain more efficient detection of the eventual attacks.

## 4.1 Data structures

The initiators and the responders are agents described by their identity, their state and a nonce they have created. In ELAN an agent is defined by the following data structure:

```
@ + @ + @  : ( AgentId SWC Nonce ) Agent;
```

There are three possible values of `SWC` states. An agent is in the state `SLEEP` if it has not sent nor received a request for a new session. In the state `WAIT` the agent has already sent or received a request and when reaching the state `COMMIT` the agent has established a session.

A nonce created by an agent $A$ in order to communicate with an agent $B$ is represented by `N(A,B)`. Memorizing the nonce allows the agent to know at each moment who is the agent with whom it is establishing a session. Since the nonce of an agent $A$ contains the identity of the destination agent, the agent $A$ can establish several parallel sessions with different agents. A dummy nonce is represented by `DN` and in order to have a uniform representation for all nonces we always reduce `DN` to its normal form `N(di,di)`.

A set of agents is described by the associative-commutative (AC) data structure `listAgent` defined by:

```
@       : (Agent) listAgent;
@ || @  : ( listAgent  listAgent ) listAgent (AC);
nnl     : listAgent;
```

where `nnl` represents the empty set of agents.

The agents exchange messages defined by:

```
@'-''-''>'@':'@'['@','@','@']' :
   (AgentId AgentId Key Nonce Nonce Address) message;
```

A message of the form `A-->B:K[N1,N2,Add]` is a message sent from `A` to `B` and contains the two nonces `N1` and `N2` together with the explicit address of the sender `Add`. The address contains in fact the identity of the sender but we give it a different type in order to have a clear distinction between the sender's identity in the encrypted part of the message and in the header of the message. The header of the message contains the source and destination address of the message but since they are not encrypted they can be faked by the intruder.

In an optimized version the messages are given types according to their relevant fields. For example, a message of the form `A-->B:K[N1,N2,Add]T:MNA` is used in the first step of the protocol where only the first nonce and the identity of the sender are encrypted with the key `K` and `N2` is just a dummy nonce.

The intruders does not only participate to normal communications but can as well intercept and create (fake) messages. Therefore a new data structure is used for intruders:

```
@ # @ # @ : ( AgentId listNonce network ) intruder;
```

where the first field represents the identity of the intruder, the second one is the set of nonces it knows and the third one the set of messages it has intercepted. In our specification we use only one intruder and thus, the first field can be replaced by a constant identifying the intruder.

A set of nonces is defined by an associative-commutative (AC) data structure:

```
@      : (Nonce) listNonce;
@ | @  : ( listNonce listNonce) listNonce (AC);
nl     : listNonce;
```

with `nl` the empty set of nonces.

The communication network is described by a possibly empty set of messages:

```
@      : (message) network;
@ & @  : (network  network) network (AC);
nill   :network;
```

with `nill` representing the network with no messages.

The ELAN rewrite rules are used to describe the modifications of the global state that consists of the states of all the agents involved in the communication and the state of the network and that is defined by:

```
@ <> @ <> @ <> @  :
   ( listAgent listAgent intruder network ) state;
```

where the first two fields represent the set of initiators and responders, the third one represents the intruder and the last one the network.

For reasons of efficiency a non-AC version of the specification has been developed as well. In this version the sets represented by AC data structures are replaced by lists. In this case the global state is defined by:

```
@ <> @ <> @ <> @  :
   ( list[Actor] list[Actor] intruder list[message] ) state;
```

where `list[X]` is a generic type defined in the ELAN libraries that describes the lists of elements of type `X`.

## 4.2 Rewrite rules

The rewrite rules describe the behavior of the honest agents involved in a session and the behavior of the intruder that tries to impersonate one of the agents. We will see that the invariants of the protocol are expressed by rewrite rules as well.

### 4.2.1 The agents

Each modification of the state of one of the participants to a session is described by a rewrite rule. At the beginning all the agents are in the state `SLEEP` waiting either to initiate a session or to receive a request for a new session.

When an initiator is in the state `SLEEP`, it initiates a session with one of the responders by sending the appropriate message as defined by the first step of the protocol. The following rewrite rule is used:

```
[initiator-1] x+SLEEP+resp  || E <> D <> w#l#ll <> ls                          =>
              x+WAIT+N(x,y) || E <> D <> w#l#ll <> x-->y:K(y)[N(x,y),DN,A(x)] & ls
                      where (Agent)y+std+init :=(extAgent) elemIA(D || w+SLEEP+DN)
end
```

In the above rewrite rule `x` and `y` are variables of type `AgentId` representing the initiator's identity and the responder's identity respectively. The initiator sends a nonce `N(x,y)` and its address (identity) encrypted with the public key of the responder and goes in the state `WAIT` where it waits for a response. Since only one nonce is necessary in this message a dummy nonce `DN` is used in the second field of the message. The message is sent by including it in the set of messages available on the network.

The responder's identity (`y`) is selected non-deterministicly from the set of responders or from the set of intruders; in our case only one intruder.

If the destination of the previously sent message is a responder in the state `SLEEP`, then this agent gets the message and decrypts it if it is encrypted with its key. Afterwards, it sends the second message from the protocol to the initiator and goes in the state `WAIT` where it waits for the final acknowledgement:

```
[responder-1] E <> y+SLEEP+init  || D <> I <> w-->y:K(y)[N(n1,n3),N(n2,n4),A(z)] & ls =>
              E <> y+WAIT+N(y,z) || D <> I <> y-->z:K(z)[N(n1,n3),N(y,z),A(y)] & ls
end
```

The nonce generated by the responder is built starting from the identity of the initiator found in the encrypted part of the message and not in the header.

The initiators in the state `WAIT` expect as response a message similar to the one sent in the rule `responder-1` and two possibilities should be considered: either the correct nonce is received and the last message of the protocol is sent to the responder or the nonce is not the expected one and an error takes place. We can describe this behavior either by using two rewrite rules or only one with a `choose-try` construction:

```
[initiator-2] x+WAIT+N(x,v) || E <> D <> I <> w-->x:K(x)[N(n1,n3),N(n2,n4),A(z)] & ls =>
              S
  choose
    try
      if x==n1 and v==n3
      where S:=() x+COMMIT+N(x,v) || E <> D <> I <> x-->v:K(v)[N(n2,n4),DN,DA] & ls
    try
      if x!=n1 or v!=n3
      where S:=() ERROR
  end
end
```

If the correct nonce has been received the initiator changes its state to `COMMIT`, that represents the fact that it has established a session with the expected responder, and sends back to the responder the nonce received in the last message.

We can ignore the error case from the rule `initiator-2` and in this case the rule is just not applied if the message does not contain the expected nonce. The two solutions lead to similar results but having an error state allows us to analyze the failing attempts to attack the protocol.

The responder reacts similarly when it receives the last acknowledge message and either it changes its state to `COMMIT` or an error is obtained depending on the nonce received:

```
[responder-2] E <> y+WAIT+N(y,x) || D <> I <> w-->y:K(y)[N(n1,n3),N(n2,n4),A(v)] & ls =>
              S
                choose
                  try
                    if x==n3 and y==n1
                    where S:=()E <> y+COMMIT+N(y,x) || D <> I <> ls
                  try
                    if y!=n1 or x!=n3
                    where S:=() ERROR
                end
end
```

Once the responder has committed, a sessions is supposed to be established between the agents corresponding to the variables `x` and `y` and the nonce `N(y,x)` can be used as a symmetric encryption key for further communications between the two agents.

### 4.2.2 The intruder

The intruder can be viewed as a normal agent that can not only participate to normal sessions but that tries also to break the security of the protocol by obtaining information that are supposed to be confidential. The network that serves as communication support is common to all the agents and therefore all the messages can be observed or intercepted and new messages can be inserted in it.

Hence, the specification should describe an intruder that it is able to:

- observe and intercept any message in the network,
- decrypt messages encrypted with its key and store the obtained information,

- replay intercepted messages and generate fake messages starting from the information it has gained.

In our approach we do not make the difference between observing and, instead, intercepting and we intercept all the messages and eventually replay them unchanged.

The intruder intercepts all the messages in the network but the messages generated by itself and stores or decrypts them. If a message is encrypted with its key, it decrypts it and stores the obtained nonces:

```
[intruder-1] E <> D <> w#l#ll <> z-->x:K(w)[N(n1,n3),N(n2,n4),A(v)] & ls  =>
             E <> D <> w#N(n1,n3) | N(n2,n4) | l#ll <> ls
                 if w!=z  // not its messages
end
```

The dummy nonces that can be obtained are useless and should not be stored by the intruder. Instead of testing systematicly for dummy nonces we store all the nonces but we normalize the intruder's store of nonces according to an unlabeled rule that is applied after each application of the rule `intruder-1`:

```
        [] N(di,di) | l => l   end
```

If the message is not encrypted with the intruder's key and if it has not been sent by the intruder it is just stored in order to replay it later:

```
[intruder-2] E <> D <> w#l#ll <> z-->x:K(y)[N(n1,n3),N(n2,n4),A(v)] & ls   =>
             E <> D <> w#l#z-->x:K(y)[N(n1,n3),N(n2,n4),A(v)] & ll <> ls
                if w!=z  // not its messages
                if w!=y  // not encrypted with its key
end
```

The messages stored by the intruder are sent to all the agents without modifying the encrypted part but specifying that the message comes from the intruder:

```
[intruder-3] E <> D <> w#l# x-->y:K(z)[N(n1,n3),N(n2,n4),A(v)] & ll <> ls  =>
             E <> D <> w#l#ll <> w-->t:K(z)[N(n1,n3),N(n2,n4),A(v)] & ls
                where (Agent)t+std+dn :=(extAgent) elemIA(D || E)
end
```

By checking that the source of the messages intercepted by the intruder is not the intruder itself we avoid the intruder cycling in a loop generate-intercept on the same message. The destination of the message is selected from the union between the set of initiators and the set of responders.

The nonces obtained previously by the intruder are used in order to generate fake messages that are sent to all the agents. If only one nonce is available the intruder cannot generate fakes for messages sent by the responder in the second step of the protocol. These fake messages are sent to all the agents in the network and the intruder tries to impersonate all the agents by using a random address `xadd`.

```
[intruder-4] E <> D <> w # resp | l # ll <> ls
             E <> D <> w # l # ll <> w-->y:K(y)[resp,DN,A(xadd)] & ls   =>
                where (Agent)y+std+dn :=(extAgent) elemIA(D || E)
                where (Agent)xadd+std1+dn1 :=(extAgent) elemIA(D || E)
end

[intruder-4] E <> D <> w # resp | init | l # ll <> ls
             E <> D <> w # l # ll <> w-->y:K(y)[resp,init,A(xadd)] & ls   =>
                where (Agent)y+std+dn :=(extAgent) elemIA(D || E)
                where (Agent)xadd+std1+dn1 :=(extAgent) elemIA(D || E)
end
```

The destination of the message and the address in the encrypted part of the message are obtained with an undeterministic strategy `extAgent` that selects at each application a new agent from the set given as argument. At each application of the rule a different message is generated and is eventually sent to a different destination. If the current message does not lead to an attack a backtrack is performed and the next message is sent. We go on like this until an attack is discovered or no new messages can be generated. This allows the intruder to generate all the possible messages and send them to all the possible agents.

### 4.2.3 The invariants

Now, we present the invariants used to specify the correctness condition of the protocol. First, the authenticity of the responder can be tested by verifying that if an initiator `A` committed with a responder `B`, then `B` has really been involved in the protocol. Instead of specifying this condition we define its negation that can be seen as a violation of the authenticity of the protocol. The rewrite rule describing the negation of the invariant checks if an initiator is in the state `COMMIT` while the corresponding responder (that is not an intruder) has neither committed nor sent an appropriate response. Thus, we should check that there exists no responder neither in the state `COMMIT` nor in the state `WAIT` and waiting for an acknowledgement from the initiator.

```
[attack-1] x+COMMIT+N(x,y) || E <> D <> I <> ls =>
           ATTACK
                if y!=i
                if not(existAgent(y+WAIT+N(y,x),D)) and
                   not(existAgent(y+COMMIT+N(y,x),D))
end
```

For the authenticity of the initiator we verify that if a responder committed with an initiator then the initiator have committed as well and with the appropriate responder. We proceed similarly as for the first invariant and we specify the negation of the invariant by a rewrite rule:

```
[attack-2] E <> y+COMMIT+N(y,x) || D <> I <> ls =>
           ATTACK
                if x!=i
                if not(existAgent(x+COMMIT+N(x,y),E))
end
```

If one on these two rewrite rules can be applied during the execution of the specification then the authenticity of the protocol is not ensured and an attack can be described from the trace of the execution. If the rewrite rule `attack-1` can be applied we can conclude that the responder has been impersonated by the intruder and if the rewrite rule `attack-2` can be applied we can conclude that the initiator has been impersonated by the intruder.

## 4.3 Strategies

The rewrite rules used to specify the behavior of the protocol and the invariants should be guided by a strategy describing their application. Basically, we want to apply repeatedly all the above rewrite rules in any order and in all the possible ways until one of the attack rules can be applied.

For example, the rule `initiator-1` describes how an initiator sends a message to only one responder or to the intruder, but it can be used by the strategy in order to describe how each initiator sends the message to each responder.

The strategy is easy to define in ELAN by using the undeterministic choice operator `dk`, the `repeat*` operator representing the repeated application of a strategy and the `;` operator representing the sequential application of two strategies:

```
[]attStrat  =>  repeat*(
                        dk(
                           attack-1, attack-2,
                           intruder-1, intruder-2, intruder-3, intruder-4,
                           initiator-1, initiator-2, responder-1, responder-2)
                );
                attackFound
end
```

The strategy tries to apply one of the rewrite rules given as argument to the `dk` operator starting with the rules for attacks and intruders and ending with the rules for the honest agents. If the application succeeds the state is modified accordingly and the `repeat*` strategy tries to apply a new rewrite rule on the result of the rewriting. When none of the rules is applicable, the `repeat*` operator returns the result of the last successful application. Since the `repeat*` strategy is sequentially composed with the `attackFound` strategy, this latter strategy is applied on the result of the `repeat*` strategy.

The strategy `attackFound` is nothing else but the rewrite rule:

```
[attackFound]   ATTACK   =>   ATTACK              end
```

If an attack has not been found and therefore the strategy `attackFound` cannot be applied a backtrack is performed to the last rule applied successfully and another application of the respective rule is tried. If this is not possible the next rewrite rule is tried and if none of the rules can be applied a backtrack is performed to the previous successful application.

If the result of the strategy `repeat*` reveals an attack the strategy `attackFound` can be applied and the overall strategy succeeds. The trace of the attack can be recovered in the ELAN environment.

**Example 4.1** Let us consider that we have only one initiator `a` and one responder `b` trying to establish a session while the intruder `i` tries to impersonate one of the two agents. The initial state is represented by

```
a+SLEEP+N(a,a)||nnl<>b+SLEEP+N(b,b)||nnl<>i#nl#nill<>nill
```

According to the strategy `attStrat` the following sequence of rewrite rules is applied in the `repeat*` loop:

```
initiator-1: a+WAIT+N(a,b)||nnl  <>  b+SLEEP+N(b,b)||nnl  <>  i#nl#nill
             <>  a-->b:K(b)[N(a,b),DN,A(a)]&nill
responder-1: a+WAIT+N(a,b)||nnl  <>  b+WAIT+N(b,a)||nnl  <>  i#nl#nill
             <>  b-->a:K(a)[N(a,b),N(b,a),A(b)]&nill
initiator-2: a+COMMIT+N(a,b)||nnl  <>  b+WAIT+N(b,a)||nnl  <>  i#nl#nill
             <>  a-->b:K(b)[N(b,a),DN,DA]&nill
responder-2: a+COMMIT+N(a,b)||nnl  <>  b+COMMIT+N(b,a)||nnl  <>  i#nl#nill
             <>  nill
```

At this moment, no rewrite rule can be applied on the result of the rule `responder-2`, that represents a state in which a correct session between `a` and `b` has been established. Therefore, the strategy `repeat*` returns it as result and the strategy `attackFound` is tried without success. A backtrack is then performed to the last successful application that is `responder-2` and another rewrite rule is tried on the state

```
a+COMMIT+N(a,b)||nnl  <>  b+WAIT+N(b,a)||nnl  <>  i#nl#nill
           <>  a-->b:K(b)[N(b,a),N(di,di),A(di)]&nill
```

The rewrite rule `intruder-2` can be executed and the state becomes

```
intruder-2: a+COMMIT+N(a,b)||nnl  <>  b+WAIT+N(b,a)||nnl
            <>  i#nl#a-->b:K(b)[N(b,a),N(di,di),A(di)]&nill  <>  nill
```

but this application will not lead to an attack.

Several backtracks are performed without reaching an attack until an alternative rewrite rule application is tried for the initial term. The local assignment of the rule `initiator-1`

```
(Agent)y+std+init :=(extAgent) elemIA(D || w+SLEEP+DN)
```

assigns to `y` one of the agents from `D` or the intruder from the variable `w`. In our case the only agent in `D` is `b` and an application of the rule using it has been already tried. Thus, the next possible value for `y` is the instantiation of the variable `w`, that in our case is `i`, and the following application is obtained:

```
initiator-1: a+WAIT+N(a,i)||nnl  <>  b+SLEEP+N(b,b)||nnl  <>  i#nl#nill
             <>  a-->i:K(i)[N(a,i),DN,A(a)]&nill
```

Unlike the backtrack shown previously where a different rewrite rule has been applied, in this latter case the backtrack has been performed in the local assignments of the last successfully applied rule.

This application is nothing else but the first step of the attack presented in Section 3 and in next section we will see the trace of the attack in ELAN.

## 4.4 An attack and the appropriate correction

Starting from the last rule application presented in Example 4.1 an attack is discovered and the following sequence of applications describes it:

```
initiator-1: a+WAIT+N(a,i)||nnl  <>  b+SLEEP+N(b,b)||nnl  <>  i#nl#nill
             <>  a-->i:K(i)[N(a,i),DN,A(a)]&nill
intruder-1:  a+WAIT+N(a,i)||nnl  <>  b+SLEEP+N(b,b)||nnl  <>  i#nl|N(a,i)#nill
             <>  nill
intruder-4:  a+WAIT+N(a,i)||nnl  <>  b+SLEEP+N(b,b)||nnl  <>  i#nl#nill
             <>  i-->b:K(b)[N(a,i),DN,A(a)]&nill
responder-1: a+WAIT+N(a,i)||nnl  <>  b+WAIT+N(b,a)||nnl  <>  i#nl#nill
             <>  b-->a:K(a)[N(a,i),N(b,a),A(b)]&nill
intruder-2:  a+WAIT+N(a,i)||nnl  <>  b+WAIT+N(b,a)||nnl
             <>  i#nl#b-->a:K(a)[N(a,i),N(b,a),A(b)]&nill  <>  nill
intruder-3:  a+WAIT+N(a,i)||nnl  <>  b+WAIT+N(b,a)||nnl  <>  i#nl#nill
             <>  i-->a:K(a)[N(a,i),N(b,a),A(b)]&nill
initiator-2: a+COMMIT+N(a,i)||nnl  <>  b+WAIT+N(b,a)||nnl  <>  i#nl#nill
             <>  a-->i:K(i)[N(b,a),DN,DA]&nill
intruder-1:  a+COMMIT+N(a,i)||nnl  <>  b+WAIT+N(b,a)||nnl  <>  i#nl|N(b,a)#nill
             <>  nill
intruder-4:  a+COMMIT+N(a,i)||nnl  <>  b+WAIT+N(b,a)||nnl  <>  i#nl#nill
             d<>  i-->b:K(b)[N(b,a),N(b,a),A(a)]&nill
responder-2: a+COMMIT+N(a,i)||nnl  <>  b+COMMIT+N(b,a)||nnl  <>  i#nl#nill
             <>  nill
attack-2:    ATTACK
attackFound: ATTACK
```

This ELAN trace describes exactly the attack shown in Section 3, only that to each message sent by the intruder corresponds, in ELAN, the interception and the generation of the message. Hence for the message $I.1.$ we apply the rule `initiator-1`, for $II.1.$ the rules `intruder-1` and `intruder-4`, for $II.2.$ the rule `responder-1`, for $I.2.$ the rules `intruder-2` and `intruder-3`, for $I.3.$ the rule `initiator-2`, for $II.3.$ the rules `intruder-1` and `intruder-4` and finally `b` commits due to the rule `responder-2`.

The message $II.2.$ is a reply to message $II.1.$ sent by the intruder impersonating $A$. The message $II.2.$ can be intercepted by the intruder and forwarded to $A$ or received directly by $A$. In both cases $A$ decrypts it and sends to its responder (i.e. $I$) the last acknowledgement. The intruder can decrypt this last message and forwards it to $B$ but encrypted with $B$'s public key.

Since the rules of the intruder are tried earlier than those of the normal agents in the strategy `attStrat`, the intruder intercepts the message $II.2.$ before sending it to $A$. The strategy can be slightly modified in order to try the agents' rules before the rules of the intruder:

```
[]attStrat  =>  repeat*(
                        dk(
                           attack-1, attack-2,
                           initiator-1, initiator-2, responder-1, responder-2,
                           intruder-1, intruder-2, intruder-3, intruder-4)
                );
                attackFound
end
```

The sequence of rewrite rules applied in order to obtain the attack in this case is:

```
initiator-1: a+WAIT+N(a,i)||nnl  <>  b+SLEEP+N(b,b)||nnl  <>  i#nl#nill
             <>  a-->i:K(i)[N(a,i),N(di,di),A(a)]&nill
intruder-1:  a+WAIT+N(a,i)||nnl  <>  b+SLEEP+N(b,b)||nnl  <>  i#nl|N(a,i)#nill
             <>  nill
intruder-4:  a+WAIT+N(a,i)||nnl  <>  b+SLEEP+N(b,b)||nnl  <>  i#nl#nill
             <>  i-->b:K(b)[N(a,i),N(a,i),A(a)]&nill
responder-1: a+WAIT+N(a,i)||nnl  <>  b+WAIT+N(b,a)||nnl  <>  i#nl#nill
             <>  b-->a:K(a)[N(a,i),N(b,a),A(b)]&nill
initiator-2: a+COMMIT+N(a,i)||nnl  <>  b+WAIT+N(b,a)||nnl  <>  i#nl#nill
             <>  a-->i:K(i)[N(b,a),N(di,di),A(di)]&nill
intruder-1:  a+COMMIT+N(a,i)||nnl  <>  b+WAIT+N(b,a)||nnl  <>  i#nl|N(b,a)#nill
```

```
            <>  nill
intruder-4:  a+COMMIT+N(a,i)||nnl  <>  b+WAIT+N(b,a)||nnl  <>  i#nl#nill
            <>  i-->b:K(b)[N(b,a),N(b,a),A(a)]&nill
responder-2: a+COMMIT+N(a,i)||nnl  <>  b+COMMIT+N(b,a)||nnl  <>  i#nl#nill
            <>  nill
attack-2:    ATTACK
attackFound: ATTACK
```

that corresponds to the description:

| | | |
|---|---|---|
| I.1. | $A \to I$ | : $\{N_A, A\}_{K(I)}$ |
| II.1. | $I(A) \to B$ | : $\{N_A, A\}_{K(B)}$ |
| I+II.2. | $B \to A$ | : $\{N_A, N_B\}_{K(A)}$ |
| I.3. | $A \to I$ | : $\{N_B\}_{K(I)}$ |
| II.3. | $I(A) \to B$ | : $\{N_B\}_{K(B)}$ |

The message $I.2.$ has been eliminated from the session between the agent $A$ and the intruder $I$ but the message $II.2.$ is seen by the agent $A$ as the second message of its session and handled accordingly.

As we have seen, small modifications in the strategy can lead to different but still equivalent attacks. The strategy can influence the efficiency of finding the first attack. The efficiency is improved if we try to attack the protocol before trying the rules of a normal session and thus, is better to try the application of the rules describing the intruder before trying the rules of normal agents. Since the search space is explored exhaustively the strategy is not important if no attacks are possible on the protocol.

In the correction shown sound in [Low96] the responder introduces its identity in the encrypted part of the message and the initiator checks if this identity corresponds to the agent he has started the session with. The corrected protocol can be written:

1. $A \to B$: $\{N_A, A\}_{K(B)}$
2. $B \to A$: $\{N_A, N_B, B\}_{K(A)}$
3. $A \to B$: $\{N_B\}_{K(B)}$

The ELAN specification is modified in order to reflect the new version of the protocol. Since in the rule `responder-1` the responder sends its identity all we have to do is to check it in the rule `initiator-2` that becomes:

```
[initiator-2] x+WAIT+N(x,v) || E <> D <> I <> w-->x:K(x)[N(n1,n3),N(n2,n4),A(z)] & ls  =>
S
  choose
    try
      if v==z              // expected responder
      if x==n1 and v==n3
      where S:=() x+COMMIT+N(x,v) || E <> D <> I <> x-->v:K(v)[N(n2,n4),DN,DA] & ls
    try
      if v!=z              // not expected responder
      if x!=n1 or v!=n3
      where S:=() ERROR
  end
end
```

As expected, when the specification is executed with this modified rule no attacks are detected.

## 4.5 Optimizations of the current implementation

In this section we present some optimizations used in order to improve the efficiency of finding the attack or showing that there is no attack.

First, each message is given a type according to its relevant fields like in [MMS97]. For example, the first message sent by the initiator to the responder contains only one nonce and the identity of the sender. We say that its type is `MNA`. The second message sent by the responder contains the two nonces and, in the corrected

version, the identity of the responder. The type of this message is `MNNA`. The last message sent in a session contains only one nonce and its type is `MNA`.

The definition of a message becomes:

```
@'-''-''>'@':'@'['@','@','@']''T'':'@ :
      (AgentId AgentId Key Nonce Nonce Address TypMes) message;
```

If the fields of the message that are not used can be identified by some special values, then the type of a message can be easily inferred without using a special field. An empty nonce field has been represented by the dummy nonce `DN` and the empty address field is represented by the dummy address `DA`. In order to have an uniform representation we define `N(di,di)` and `A(di)` as the normal forms of `DN` and `DA` respectively by the two unlabeled rules:

```
        [] DN => N(di,di)  end

        [] DA => A(di)  end
```

where `di` represents the dummy identity.

For example, the rule `responder-1` can be easily changed in order to listen only for messages of type `MNA`:

```
[responder-1] E <> y+SLEEP+init  || D <> I <> w-->y:K(y)[N(n1,n3),N(di,di),A(z)] & ls
              E <> y+WAIT+N(y,z) || D <> I <> y-->z:K(z)[N(n1,n3),N(y,z),A(y)]   & ls
                        if n1!=di and n3!=di and z!=di
end
```

The use of an explicit type field avoids the need for a condition verifying the type of the message and keeps the specification clear and concise. In this case the rule `responder-1` becomes:

```
[responder-1] E <> y+SLEEP+init  || D <> I <> w-->y:K(y)[N(n1,n3),N(n2,n4),A(z)]T:MNA & ls
              E <> y+WAIT+N(y,z) || D <> I <> y-->z:K(z)[N(n1,n3),N(y,z),A(y)]T:MNNA  & ls
end
```

Second, a more important optimization proposed in [MMS97] is to make the initiator and responder listen only to messages coming from the intruder. Since all the messages are intercepted by the intruder and are then forwarded to the other agents, no message is lost when using the above optimization. The search branches starting with a message received by a normal agent from a normal agent are not explored but the equivalent branch where the corresponding message has passed by the intruder is analyzed.

The rewrite rules that should be changed in order to handle this optimization are `responder-1`, `initiator-2` and `responder-2`:

```
[responder-1] E <> y+SLEEP+init  || D <> I <> w-->y:K(y)[N(n1,n3),N(n2,n4),A(z)] & ls =>
              E <> y+WAIT+N(y,z) || D <> I <> y-->z:K(z)[N(n1,n3),N(y,z),A(y)]   & ls
                    if w==i
end

[initiator-2] x+WAIT+N(x,v) || E <> D <> I <> w-->x:K(x)[N(n1,n3),N(n2,n4),A(z)] & ls =>
              S
  if w==i
  choose
    try
      if x==n1 and v==n3
      where S:=() x+COMMIT+N(x,v) || E <> D <> I <> x-->v:K(v)[N(n2,n4),DN,DA] & ls =>
    try
      if x!=n1 or v!=n3
      where S:=() ERROR
  end
end

[responder-2] E <> y+WAIT+N(y,x) || D <> I <> w-->y:K(y)[N(n1,n3),N(n2,n4),A(v)] & ls =>
              S
```

```
  if w==i
  choose
    try
      if x==n3 and y==n1
      where S:=()E <> y+COMMIT+N(y,x) || D <> I <> ls
    try
      if y!=n1 or x!=n3
      where S:=() ERROR
  end
end
```

This condition in the rules of the normal agents together with a strategy where the intruder tries to impersonate the agents as soon as possible (i.e. the rules of the intruder are applied before the rules of the other agents) can improve significantly the efficiency in finding the attack or, if an attack does not exist, in exploring the search space and show that there is no attack.

In order to reduce the number of messages sent by the intruder we have proposed some modifications in the intruder's rules handling the generation of messages. When an encrypted message is forwarded in rule `intruder-3`, the respective is not sent to all the agents, but only to those agents that are able to handle it:

```
[intruder-3] E <> D <> w#l# x-->y:K(z)[N(n1,n3),N(n2,n4),A(v)]T:st      & ll <> ls  =>
             E <> D <> w#l#ll <> w-->t:K(z)[N(n1,n3),N(n2,n4),A(v)]T:st & ls
                choose
                  try
                    if st==MNA or st==MN
                    where (Agent)t+std+dn :=(extAgent) elemIA(D)
                  try
                    if st==MNNA
                    where (Agent)t+std+dn :=(extAgent) elemIA(E)
                end
end
```

The messages of type `MNA` or `MN` are interesting only for the responders from the set `D`, while the messages of type `MNNA` are handled only by the initiators from the set `E`.

Similarly, when messages of a certain form are generated in rule `intruder-4` they are sent only to agents that might be interested in the respective messages:

```
[intruder-4] E <> D <> w # resp | l # ll <> ls  =>
             E <> D <> w # l # ll <> w-->y:K(y)[resp,DN,A(xadd)]T:MNA & ls
                where (Agent)y+std+dn :=(extAgent) elemIA(D)
                where (Agent)xadd+std1+dn1 :=(extAgent) elemIA(E)
end

[intruder-4] E <> D <> w # resp | l # ll <> ls  =>
             E <> D <> w # l # ll <> w-->y:K(y)[resp,DN,DA]T:MN & ls
                where (Agent)y+std+dn :=(extAgent) elemIA(D)
end

[intruder-4] E <> D <> w # resp | init | l # ll <> ls  =>
             E <> D <> w # l # ll <> w-->y:K(y)[resp,init,A(xadd)]T:MNNA & ls
                where (Agent)y+std+dn :=(extAgent) elemIA(E)
                where (Agent)xadd+std1+dn1 :=(extAgent) elemIA(D)
end
```

Here, not only that the destination of messages is established according to the type of message the intruder sends, but the addresses encrypted in the message are generated depending on the message type as well.

Without these modifications the messages are sent to all agents and read by them, but they are handled only by the appropriate agents. Therefore, the number of states that are not explored due to this optimization is not very important. Still, the number of backtrack choice points is reduced this way and the efficiency improved.

## 4.6 Comparing to existing approaches

The specification technique we have proposed in this paper allows us to explore a finite state space and we can consider it as a form of model-checking. This kind of approaches are usually used for detecting the attacks against the protocol and not for proving its correctness.

In [Low96] G. Lowe used FDR [Ros94], a model checker for CSP [Hoa85], to analyze the properties of the Needham-Schroeder public-key protocol. He described the possible attacks on the protocol and proposed the appropriate corrections. The new specification is automatically proved correct on a finite model and a proof is given for showing that the properties are verified for an unbounded model.

An approach based on rewriting logic has been proposed in [DMT98]. The rewrite rules are used for specifying the protocol and strategies are used for exploring the state space. Comparing to our specification the Maude approach allows multiple sessions for an agent. An agent can participate simultaneously in several sessions acting as an initiator or as a responder. Although this behavior can be easily described in the ELAN specification we preferred to use our model for the reasons explained below.

First, let us consider an agent participating in two sessions, in one as an initiator and in the other as a responder. Since the type of the message can be obtained explicitly or implicitly when the agent receives a message it knows which is the session concerned by the message. Hence, the two sessions do not interfere and we consider that there are the sessions of two different agents, one acting as an initiator and the other as a responder. Similarly, the messages exchanged by an agent involved in several session but with a precise role (initiator or responder) can be assigned to the corresponding sessions according to the nonces they contain and thus, the sessions can be considered completely separated.

Of course that a detailed specification should consider multiple concurrent sessions, but in practice this is not possible due to the explosion of the state space and some other technique like a hand proof should be employed. In the Maude specification, although we have the possibility to specify multiple sessions for an agent, only two agents are considered when the specification is run. In this case each agent can have only two session, one as initiator and one as responder, but the time for proving the correctness of the protocol is already very important.

The ELAN specification can be modified in order to describe multiple sessions by simply merging the pools of initiators and senders. The modifications in the rewrite rules are minor and if we want to test the protocol with two agents with at most two parallel sessions the following query should be used:

```
a+SLEEP+N(a,a) || a+SLEEP+N(a,a) || b+SLEEP+N(b,b) || b+SLEEP+N(b,b) || nnl
<> i#nl#nill <> nill
```

The expected attack is discovered immediately but when checking the corrected version very long times are needed.

The Needham-Schroeder public-key protocol is also analyzed together with another two authentication protocols in [MMS97] by using a general state enumeration tool, Mur$\varphi$. Almost the same methodology has been used in the Mur$\varphi$ and ELAN specifications and the following results have been obtained for the corrected version of the protocol:

| initiators | responders | network | Mur$\varphi$ | ELAN simple | ELAN(Mur$\varphi$ like) | ELAN optimized |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 2578 rules<br>0.56s | 7442 rules<br>0.167s | 4308 rules<br>0.125s | 711 rules<br>0.064s |
| 1 | 1 | 2 | 136273 rules<br>18.40s | 453514 rules<br>7.007s | 333552 rules<br>4.980s | 6700 rules<br>0.162s |
| 2 | 1 | 1 | 25701 rules<br>6.81s | 575101 rules<br>8.571s | 257087 rules<br>3.946s | 26011 rules<br>0.483s |
| 2 | 2 | 1 | 557430 rules<br>303.46s | 118214389 rules<br>1658.296s | 22985807 rules<br>333.096s | 753785 rules<br>12.392s |

Besides the methods using testing and model-checking that are very convenient for attack discovery, other approaches based on theorem proving are appropriate for proving the correctness of protocols. The NRL Protocol Analyser described in [Mea96] uses an approach based on narrowing. In [JRV99] an approach based on theorem proving is used.

Tree automata has been used in [Mon99] together with a completion mechanism for verifying cryptografic protocols with a bound number of agents. A method based on tree automata and rewriting has been proposed in [GK99] where the protocol is described by a set of rewrite rules and the initial set of communication requests by a tree automaton. In this latter approach the classical properties are proved by checking that the intersection between an over-approximation of the exchanged messages and a set of prohibited behaviors is the empty set.

# 5 Conclusions

We have shown how computational systems can be used as a logical framework for representing the Needham-Schroeder public-key protocol. This approach can be easily extended to other authentication protocols and an implementation of the TMN protocol has been already developed. The rules describing the protocol are naturally represented by conditional rewrite rules. The associative-commutative operators allows us to handle easily the random selection of agents from a set of agents or of a message from a set of messages. By using associative-commutative operators and simple unlabeled rewrite rules we can immediately describe some properties of the protocol, like the uniqueness of messages in the network.

The strategy used for guiding the application of the rewrite rules is important when an attack on the protocol exists. In this case we can improve the efficiency by trying to attack the protocol as soon as possible. This corresponds to a strategy that applies as soon as possible the rewrite rules corresponding to the intruders and that verifies at each step the invariants of the protocol. The ELAN strategy is easily modifiable and we have seen that small changes can lead to different, but still equivalent, attacks.

The behavior of intruders can be easily modified by changing the corresponding rules. We can imagine, for example, that the intruders cannot fake the source and destination addresses of the messages. We can also modify the rewrite rules in order to improve the efficiency of the implementation. For the specification of the Needham-Schroeder public-key protocol presented in this paper the modification of agents in order to receive only messages from the intruders has a major impact on the efficiency of the implementation.

A drawback of an approach based on model-checking is that the state space is finite and thus, some other methods should be used in order to show that properties proved for the finite model can be lifted to an unbounded model. Therefore, the approach like the one we have proposed is usually used for detecting the attacks against the protocol and not for proving its correctness. The proved properties can be generalized using hand-proofs like in [Low96] or techniques based on theorem proving like in [JRV99].

An obvious continuation of this work would be the use of a specification language like CAPSL ([Mil97, DM99]) or Casper ([Low98]) for our implementation. Such an interface would allows us to have in the same framework an efficient detection of attacks using our approach and efficient proofs of the the correctness of the modified protocols using theorem provers.

**Acknowledgements** We sincerely thank Florent Jacquemard and Michael Rusinowitch for helpful discussions on the verification of authentication protocols and Pierre-Etienne Moreau for their useful hints and suggestions concerning the implementation in ELAN. We thank Azzouz Ezzaiem who helped us in the development of the ELAN implementation.

# References

[BKK+96] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Marian Vittek. ELAN: A logical framework based on computational systems. In José Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4 of *Electronic Notes in TCS*, Asilomar (California), September 1996.

[BKK+97] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Marian Vittek. ELAN *V 2.0 User Manual.* Inria Lorraine & Crin, Nancy (France), first edition, May 1997.

[DDHY92] D.L. Dill, A.J. Drexler, A.J. Hu, and C.H. Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[DJ90] N. Dershowitz and J.-P. Jouannaud. *Handbook of Theoretical Computer Science*, volume B, chapter 6: Rewrite Systems, pages 244–320. Elsevier Science Publishers B. V. (North-Holland), 1990. Also as: Research report 478, LRI.

[DM99] G. Denker and J. Millen. Capsl intermediate language. In *Formal Methods and Security Protocols*, 1999. FLOC '99 Workshop.

[DMT98] G. Denker, J. Meseguer, and C. Talcott. Protocol specification and analysis in Maude. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of the 2nd International Workshop on Rewriting Logic and its Applications, WRLA'98*, volume 15, Pont-à-Mousson (France), September 1998. Electronic Notes in Theoretical Computer Science.

[GK99] T. Genet and F. Klay. Rewriting for cryptografic protocols verification. Technical report, CNET, 1999.

[Hoa85] C. A. R. Hoare. *Communicating sequential processes.* Prentice-Hall, 1985.

[JK91] J.-P. Jouannaud and Claude Kirchner. Solving equations in abstract algebras: a rule-based survey of unification. In Jean-Louis Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 8, pages 257–321. The MIT press, Cambridge (MA, USA), 1991.

[JRV99] F. Jacquemard, M. Rusinowitch, and L. Vigneron. Compilation et surreduction des protocoles d'authentification. Technical report, Loria, 1999.

[KKV95] Claude Kirchner, Hélène Kirchner, and Marian Vittek. Designing constraint logic programming languages using computational systems. In P. Van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming. The Newport Papers.*, chapter 8, pages 131–158. The MIT press, 1995.

[Low95] G. Lowe. An attack on the Needham-Schroeder public key authentication protocol. *Information Processing Letters*, 56:131–133, 1995.

[Low96] G. Lowe. Breaking and fixing the Needham-Schroeder public key protocol using CSP and FDR. In *Proceedings of 2nd TACAS Conf.*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166, Passau (Germany), 1996. Springer-Verlag.

[Low98] G. Lowe. Casper: a compiler for the analysis of security protocols. *Journal of Computer Security*, 6(1):53–84, 1998.

[Mea96] C.A. Meadows. Analyzing the Needham-Schroeder Public Key Protocol: A Comparison of two Approaches. In *Proceedings of 4th ESORICS Symp.*, volume 1146 of *Lecture Notes in Computer Science*, pages 351–364, Rome (Italy), 1996. Springer-Verlag.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[Mil97] J. Millen. CAPSL: Common Authentication Protocol Specification Language. Technical Report MP 97B48, The MITRE Corporation, 1997.

[MMS97] J. C. Mitchell, M. Mitchell, and U. Stern. Analysis of cryptographic protocols using murphi. In *IEEE Symposium on Security and Privacy*, pages 141–153, Oakland, 1997.

[MOM96] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. In José Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.

[Mon99] D. Monniaux. Abstracting cryptographic protocols with tree automata. In *Proceedings of 6th SAS*, Venezia (Italy), 1999.

[NS78] R.M. Needham and M.D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[Ros94] A.W. Roscoe. Model-checking CSP. In *A Classical Mind, Essays in Honour of C.A.R.Hoare.* Prentice-Hall, 1994.

[Vit96] Marian Vittek. A compiler for nondeterministic term rewriting systems. In Harald Ganzinger, editor, *Proceedings of RTA'96*, volume 1103 of *Lecture Notes in Computer Science*, pages 154–168, New Brunswick (New Jersey), July 1996. Springer-Verlag.