An Analysis of the Needham-Schroeder Public-Key Protocol with MGS

Olivier MICHEL*

LaMI umr 8042 CNRS – Université d'Évry
Tour Évry-2, 523 place des terrasses de l'agora
91000 France
E-mail: michel@lami.univ-evry.fr

Florent JACQUEMARD

LSV, CNRS UMR 8643 – ENS de Cachan
61 avenue du Président Wilson
94235 CACHAN Cedex, France
E-mail: florent.jacquemard@lsv.ens-cachan.fr

Abstract

In this paper, we develop an analysis of the Needham-Schroeder Public-Key Protocol (NSPK) using a P system approach. This analysis is used to validate the protocol and exhibits, as expected, a well known logical attack. The novelty of our approach is to use MGS to find the attack by a systematic state exploration.

The use of multiset rewriting has already been advocated for the development of protocol validation tools. In this work, we focus on the use of nested multisets (*i.e.* membranes). The use of membranes enables to tight the conditions for detecting an attack.

The two proposed versions of the analysis have been successfully implemented in MGS and we conclude the paper by a discussion on how the MGS programs can be translated into standard P systems.

1 Goal and Motivations

Since the 1994 landmark demonstration by Adleman of the possibilities of DNA to solve a class of combinatorial problems, biocomputing has often be advocated to develop "chemically combinatorial problem solvers". In this paper, we want to use an approach belonging to the membrane computing area to address a well known combinatorial problem: the analysis of a cryptographic protocol.

Our starting point is the logical analysis of the Needham-Schroeder Public-Key Protocol (NSPK). The goal of the logical analysis is to find an interleaving of elementary actions (sending and answering messages) that allows an intruder to obtain confidential information. We have chosen this problem because it is simple to explain, at the same time it requires sophisticated data-structures for the exploration of its state space, it is paradigmatic of this kind of applications, and its solution is well-known – hence we can validate our result.

*Corresponding author

The approach taken in this paper is brute force and consists in the exploration of the state space of the protocol for a systematic search of attacks. Indeed, we are interested in the study of the states representation and generation, rather than in designing a new and smart search strategy. This approach is motivated by the opinion that the representation of data is a central problem in biocomputing.

The `MGS` Programming Language. To validate our propositions, we have completely implemented and validated two versions of the logical analysis using the `MGS` programming language. `MGS` is a research project devoted to the design and the development of a programming language dedicated to the simulation of biological processes [2, 3]. Based on topological notions, `MGS` supports the notion of transformation: a localized computation specified by rules. One can for example defines multiset rewriting rules [18] that act on a nest of multisets (i.e., membranes). These rules can be used to move values from a multiset to another one, as well as to dissolve, divide or create new multisets. So, `MGS` can potentially be used to approximate the execution of a P system [1]. However, we outline that the `MGS` project focuses on the design of a programming language rather than the development of a well founded computational model. The programmer can use high-level constructions that ease the programming tasks but are not directly available in the usual P system formalism (e.g. lambda-expression, data-structure like arrays, arbitrary graphs, etc.). These constructions can often be seen as syntactic sugar over the core P system formalism (see section 6) but make difficult the direct expression of the `MGS` programs.

P Systems and Cryptographic Analysis. The idea of using the P system formalism in the area of communication protocol and security is not new but has not been applied before in the area of logical analysis. In [4], the author proposes the use of a P system to specify a message authentification protocol. The problem addressed in our paper focus on a different point: we do not specify new protocols but rather develop a verification tool for an existing one. References [15] and [12] study cryptographic functions (inverting one-way functions and breaking DES) which is not of our concern since encryption is handled here as a black box and is considered as unbreakable (see section 2).

Organization of the Paper. The rest of this paper is organized as follows. In section 2 we give some background on the logical analysis of cryptographic protocol. Section 3 present briefly the `MGS` language. `MGS` is an experimental programming language handling various types of membrane structures in a homogeneous and uniform syntax. Section 4 describes precisely the Needham-Schroeder public-key protocol. Section 5 presents the technical meat of the paper. We develop two versions of the analysis of NSPK. The first version improves on an analysis initially proposed within the ELAN rewriting framework, with a more accurate representation of states using nesting. The second version goes further by generalizing the approach to the exploration of general state spaces. Finally, the last section summarizes our work and discuss the possible translation of the `MGS` program in the core P system formalism.

2 Logical Validation of Cryptographic Protocol

Cryptographic protocols are nowadays paramount in all the domains concerned with secure communications, like on-line banking, electronic commerce, enterprise servers, mobile telephony, pay-per-view video, *etc.* They define the exchange of a few messages between parties in order to distribute some secrete data like cryptographic keys or to authenticate

themselves. These messages are built with cryptographic primitives like encryption, signature or hash-functions, and therefore the security of protocols relies on the strength of the cryptographic algorithms in use. However, using strong algorithms, and even *perfect* (*i.e.* presumably unbreakable) ones, does not always prevent from attacks of a logical nature. For instance, the well known problems of the distribution of keys for symmetric cryptosystems like AES and the authenticity of public keys in PKIs are beyond the scope of the study of encryption functions.

Logical attacks occur when the cryptographic operators are seen as (unbreakable) black boxes, and the security is compromised by an unexpected interleaving of messages between honest agents and a malicious intruder which has some control over the network. Examples of control over the network are the ability to spy messages or to impersonate identities while sending new ones. Such attacks can be realized at almost no computational cost and hence can have disastrous consequences.

Various formal methods have been proposed for analyzing automatically the vulnerability of cryptographic protocols to logical attacks, both for searching of such flaws or for the formal proof of their absence. Several systems have been implemented in purpose for the search of flaws, *e.g.* [13, 29, 24]. But many general purpose languages and tools have also appeared appropriate in this setting, with the advantage of a greater expressive power, efficiency and maturity. To cite only a few examples, there are model checkers like FDR [27] or murφ [28], first order theorem provers [17, 11] and declarative languages used as model checkers [9, 7].

Our purpose in this paper is to describe an experiment with the language `MGS` for modeling a cryptographic protocol and finding of attacks by state exploration. `MGS` [22] is a declarative language that manipulates uniformly various data structures (sequences, sets, multisets, arrays...) seen from a topological point of view. The use of a declarative language like `MGS` is strongly advocated by the intruder-centric model which is generally considered in order to apply formal methods to cryptographic protocol verification. In this model, often referred as "Dolev-Yao model" [10], the agents executing the protocol communicate asynchronously via a unique channel which has been compromised by an intruder. The intruder is able to spy and divert every message on the channel, to analyze read messages, with the restriction that he must know the appropriate encryption key in order to decipher an encrypted message. He can also build and send new messages, possibly under a fake identity. The global state of the system can hence be represented by a heterogeneous set containing the local states of each agent (with a bounded memory), the messages and submessages known to the intruder and the messages sent and not yet received by an agent. The actions of the agents (receiving and sending messages) as well as of the intruder can be modeled with `MGS` rewriting rules (called *transformations*) on multisets. The sear of an interleaving leading to an attack can be coded very simply in `MGS` with an appropriate pattern expression to find sequences of value or arbitrary length. The representation of states by sets (as opposed to multisets), which are a built-in structure in `MGS`, permits an important reduction of the search space.

The problem of finding attacks of protocols is highly undecidable, the state space being infinite for several reasons: the unboundedness of the number of agents in presence, the ability of agents to generate fresh random data (nonces), the unlimited size of terms generated by the intruder. In order to restrict our exploration with `MGS` to a finite search space, while keeping our procedure reasonably complete, we shall apply on some theoretical results on protocol verification. It is shown in [16] that the problem of protocol security (non-existence of attacks) becomes decidable when the number of agents considered is bounded. Indeed, [16] shows that in this case, whenever there exists an attack, there

exists an attack involving messages of a bounded size. We can use this result here to ensure the completeness of our attack search procedure, given a finite number of agents.

3 A Brief Introduction to the MGS Language

We briefly present in this section the MGS language. We do not detail all the features of the language but we rather focus on the notions required to understand the rest of the paper.

3.1 MGS as a Functional Language

MGS embeds a complete, impure, dynamically typed, strict, functional language. We only describe here the major differences between the constructions available in MGS with respect to functional languages like OCAML [25] or HASKELL [23].

3.1.1 Values

Atomic values (like integers, floats, booleans, strings,...) with their usual functions, are available. Constants are denoted with a backquote: `‘REQ` (they are reminiscent of LISP symbols). The only operations allowed on a constant is to store it or to compare it for equality with another value.

Records (cartesian products with labels) are defined using braces: {`x=0, y=1`} creates a pair with label `x` and `y` (MGS record are similar to Pascal's record or C's struct). The fields are accessible using the dot notation: `let v = {x=0, y=1} in v.x` has value `0`. Since records are used in MGS to define a particular state of an entity, MGS allows the definition of predicates based upon the fields found in a record. The keyword `record` is used to define such predicates:

```
record agent = {id, ni, nr, pc}
```

defines the predicate `agent` that holds only if applied on a record value that has at least all the fields `id, ni, nr` and `pc`. Record `alice` defined as `record alice = {dest} + agent` extends predicate `agent` with the additionally required field `dest`. So far, the record predicates only required to have the fields to hold. The predicate `req` defined as `record req = {pc = ‘REQ}` holds only if its argument has a field `pc` with a value equal to the constant `‘REQ`.

3.1.2 Imperative Variables and Sequencing

Variables in a functional languages are not true variables: they refer to values and cannot be updated. MGS has a notion of *imperative* variable (also called *mutables*) that can be updated. The `:=` operator allows to define such variables. For example `imp := 0` defines `imp` with value `0` that can be later updated with the same construction.

The semi column operator `;` is used to express the sequencing of expressions: the value of `f();g()` is the value returned by `g()` but `f()` has been computed before.

3.1.3 Functions

Since MGS is a functional language, it has functions as first-class values. Functions are defined either using the construction `fun` like in `fun max(x, y) = if (x > y) then x`

`else y fi` or using the classical lambda notation as in `\x.\y.if (x > y) then x else y fi`

Computations by fixpoints are heavily used in applications like simulations or state space explorations. `MGS` provides an operator to compute iterations and fixpoints of functions. Let `f` be a function, then `f[iter = n](x)` computes $f^n(x)$ and `f[*](x)` denotes the fixpoint of `f` starting from `x`.

Functions together with mutables and iterations allows to define functions that pass informations between calls. For example, function `f` defined as `fun f[acc=0](x)=(acc := acc+1; x+acc)` allows to define an accumulator `acc` which stores a value that is incremented between each call. The value of `f['iter = 10, acc = 0](1)` is `56`.

3.2 Topological Collections and their Transformations

The distinctive features of the `MGS` language is its handling of entities structured by *abstract topologies* using *transformations* [21]. A set of entities organized by an abstract topology is called a *topological collection*. Topological means here that each collection type defines a neighborhood relation inducing a notion of *sub-collection*. A sub-collection B of a collection A is a subset of connected elements of A and inheriting its organization from A.

3.2.1 Collection Types

Many different predefined and user-defined collection types are available in `MGS`. We won't describe them here since sets, multisets and sequences are the only collection type used in this paper.

For any collection type `T`, the corresponding empty collection is written `():T`. The name of a collection type is also a predicate used to test if a value is of this type: `T(`v`)` holds only if v is of type `T`. Each collection type can be subtyped. The type declaration `collection U = T` introduces a new collection type `U` which is a subtype of `T`. The new type `U` shares the same topology as `T`. However, a value of type `U` can be distinguished from a value of type `T` using the `U` predicate (i.e., the subtyping relation implies that `U`$(u) \Rightarrow$ `T`(u), for any value u, but not the reverse). Elements in a collection can be of any type, including collections.

3.2.2 Operations on Collections

The join of two collections C_1 and C_2 (written by a comma: C_1`,` C_2) is the main operation on collections. The comma operator is overloaded in `MGS` and can be used to build any collection (the type of the arguments disambiguates the collection built). So, the expression `1, 1+1, 2+1, ():set` builds the set with the three elements $1, 2$ and 3, while the expression `1, 1+1, 2+1, ():bag` makes a bag (a set that allows multiple occurrences of the same value) with the same three elements.

3.2.3 Transformations

The *global transformation* of a topological collection C consists in the *parallel application* of a set of *local transformations*. A local transformation is specified by a rewriting rule r that specifies the replacement of a sub-collection by another one. The application of a rewriting rule $\beta \Rightarrow f(\beta, ...)$ to a collection A:

1. selects a sub-collection B of A whose elements match the *pattern* β,
2. computes a new collection C as a function f of B and its neighbors,

3. and specifies the insertion of C in place of B into A.

One should pay attention to the fact that, due to the parallel application strategy of rules, *all distinct instances B_i of the sub-collections matched by the β pattern are "simultaneously replaced" by the $f(B_i)$*. This is very different from the evaluation strategies followed by classical rewriting tools like MAUDE [8], ELAN [5], Murφ [20], MSR [6], etc.

The `MGS` experimental programming language implements the idea of transformations of topological collection into the framework of a simple dynamically typed functional language. Collections are just new kind of values and transformations are functions acting on collections and defined by a specific syntax using rules. Transformations (like functions) are first-class values and can be passed as arguments or returned as the result of an application.

3.2.4 Sub-collection Patterns

A transformation is defined by a set of rules (listed between braces). A pattern β that appears in the left hand side of a rule is an expression used to select a sub-collection to be replaced. Several operators are available; we will review here only few of them:

- **literal**: a literal value matches an element with the same value. For example, `123` matches an element with the integer value 123.
- **variable**: a pattern variable a matches exactly one element. The variable a can then occur elsewhere in the rest of the rule and denotes the value of the matched element. The identifier of a pattern variable can be used only once in a pattern. To match an element without giving it a name, an underscore `_` can be used.
- **alias**: the pattern p `as` X associates the variable X to the value matched by the pattern p. X is a regular variable than can be used as previously described.
- **neighbor**: the pattern b `,` p matches a sub-collection composed of an element matched by b neighbor of a sub-collection matched by p.
- **guard**: p/exp matches a sub-collection matched by p such that the predicate exp hold. For instance, x`,`y `/` y`>`x matches two neighbor elements such that the second one is greater than the first one.
- **repetition**: p`*` matches a sub-collection made of a (possibly empty) repetition of sub-collections matched by p. If p is a pattern variable, then its value refers to the sequence of matched elements and not to one of the individual values. For example, `3+` matches a non-empty sub-collection made only of `3`'s.

3.2.5 `MGS` and Multiset rewriting

The `MGS` framework embeds the rewriting of multisets (or sets) in the following way. In a multiset, an element is susceptible to interact with any other element, so the abstract topology of a multiset is the topology of a complete connected graph: the neighbors of an element are all the other elements in the multiset. Then, a pattern β can select an arbitrary sub-multiset and a multiset rewriting rule is simply a local transformation in this topology.

3.3 Example : Computing all the n-tuple in a Set

Let S be a set of values. To compute all the n-tuples (a 2-tuple is a pair, a 3-tuple is a triple...) one can use the transformation:

```
trans n_tuple[acc, n] = {
(_* as c) / size(c) == n / (acc := c::acc; false) => !!(0);
_ => return(acc)
}
```

In transformation `n_tuple`, parameters `acc` and `n` are mutables whose definition are local to the transformation. They are set at the first call of the transformation. Applied to a collection C, pattern of the first rule `(_* as X) / size(X) == n)` matches a subcollection c of C of size n such that all elements of c are neighbors (with respect to the topology induced by C). Once c is found, predicate `(acc := c::acc; false)` is calculated: collection c is added to the accumulator (`::` is the concatenation of a value to a collection) and the value `false` is returned. Since the predicate does not hold, the right hand side of the rule is not evaluated (the expression `!!(0)` aborts the program) and the rule is tried against another instance, storing each time the solution of the matching into the accumulator. Once all the possibilities have been tried and failed, the second rule is tried. That rule succeeds in matching anything and returns the value of the accumulator. Transformation `n_tuple[acc=set:(), n=2]((3,4,5,6,set:()));;` computes all the pairs

```
((3, 4):'seq, (3, 5):'seq, (3, 6):'seq, (4, 3):'seq,
(4, 5):'seq, (4, 6):'seq, (5, 3):'seq, (5, 4):'seq,
(5, 6):'seq, (6, 3):'seq, (6, 4):'seq, (6, 5):'seq):'set
```

where `(3, 4):'seq` is a pair holding the two integers value.

4 The Needham-Schroeder Public-Key Protocol

4.1 Description of the Protocol

The Needham-Schroeder public key protocol [14] (NSPK for short) is the favorite case study for formal methods applied to the verification of cryptographic protocols. This popularity certainly comes from one of the most famous success story in this domain, which is the discover in 1994 by G. Lowe [26] of a replay attack in this protocol 16 years after its publication. In [26], G. Lowe models the protocol in the CSP process algebra and uses the model checker FDR to explore the state space.

The Needham-Schroeder public key protocol involves two participants Alice (A), Bob (B) which are willing to authenticate reciprocally with three messages using public keys. The original protocol of [14] involves also a server distributing the public keys to A and B with three additional messages. We omit the server and its three messages here, assuming that A and B both initially know each other's public key, since they are not necessary in Lowe's attack. The messages are described in Figure 1, in the traditional notation.

In the first line (message labeled `REQ`), Alice generates a random number (nonce) N_a, appends it to her name A (the append operator is $\_,\_$) encrypts the results with Bob's public key $K(B)$ (public key encryption is denoted with the binary operator $\{\_\}_\_$) and sends the result to the network. When Bob receives a message of the form of `REQ`, he deciphers it and retrieves the identity A of Alice and the nonce N_a. Then he generates a

$$
\begin{array}{llcll}
\texttt{REQ} & A & \rightarrow & B: & \{A, N_a\}_{K(B)} \\
\texttt{CHAL} & B & \rightarrow & A: & \{N_a, N_b\}_{K(A)} \\
\texttt{AUTH} & A & \rightarrow & B: & \{N_b\}_{K(B)}
\end{array}
$$

Figure 1: Needham-Schroeder public key protocol

second random number N_b, appends it to N_a and sends the result encrypted with Alice's public key $K(A)$ (message `CHAL` for a challenge). Alice, receiving message `CHAL`, can decipher it and check whether the first component corresponds to the name she sent in message `AUTH`. Then, she resend Bob's nonce N_b encrypted with Bob's public key (message `AUTH`). Bob can check that the message `AUTH` contains the nonce N_b he has generated at second step (`CHAL`).

4.2 A Replay attack

Receiving the message `AUTH` ensures Bob that Alice has really received the message `CHAL` and answered, because Alice is the only one able to decipher this message. We assume indeed that each agent, as well as the intruder (let us call him Charly, C), knows only its own private key, and that this key is necessary to decipher a message encrypted with the corresponding public key.

Similarly, when receiving the message `CHAL`, Alice is ensured that it really comes from B (and is not a fake message from Charly), as proven by the presence of N_a because the knowledge of Bob's private key is necessary for the extraction of N_a from message `REQ`. Hence, N_a and N_b are used as *authenticators* in this protocols. Therefore, they must remain secret. However, the attack of [14], described below, shows that it is not the case, even with the above hypotheses concerning the private keys.

$$
\begin{array}{llcll}
\texttt{REQ} & A & \rightarrow & C: & \{A, N_a\}_{K(C)} \\
\texttt{REQ}' & C(A) & \rightarrow & B: & \{A, N_a\}_{K(B)} \\
\texttt{CHAL}' & B & \rightarrow & C(A): & \{N_a, N_b\}_{K(A)} \\
\texttt{CHAL} & C & \rightarrow & A: & \{N_a, N_b\}_{K(A)} \\
\texttt{AUTH} & A & \rightarrow & C: & \{N_b\}_{K(C)} \\
\texttt{AUTH}' & C(A) & \rightarrow & B: & \{N_b\}_{K(B)}
\end{array}
$$

This attack involves two sessions in parallel. In the first session, Alice enters in communication with Charly (without knowing that he is an intruder). Since the message `REQ` is encrypted with Charly's public key $K(C)$, Charly can retrieve A, N_a and encrypt it with Bob's public key $K(B)$, and send this message as the first message `REQ'` of a second session between A and B. In this step `REQ'`, Charly impersonates A, which is denoted $C(A)$. Bob answers to `REQ'` and Charly diverts this message `CHAL'` (it is by denoted $C(A)$). Then Charly, with two messages `CHAL` and `AUTH` of the first session uses A as an oracle in order to obtain the Bob's nonce N_b.

5 Two Different Models to Find an Attack on the NSPK in MGS

We shall describe here the specification of the Needham-Schroeder public key protocol and the implantation of an attack-search procedure in a `MGS` program.

The description of the protocol in MGS involves two different kind of components: *entities* and *evolution rules*. The entities are MGS records and evolution rules are given by rewrite rules specified as MGS transformations. A *system state*, we shall also write *solution*, is a finite collection of entities which are of three kind: agents, messages transmitted trough the network and messages components memorized by the intruder. Several entities in a state shall react, firing an evolution rule which transforms a system state into a successor state.

The MGS model is organized into the following parts, detailed in the next sections:

- record definitions are used to describe the three kind of entities (Section 5.1); recall that in MGS a record definition automatically defines a predicate, as described in Section 3.1.1.
- various predicates used to select, in the set of reacting entities, specific entity of a given kind (an agent, a message),
- transformations specifying the abilities of the intruder to collect all the messages that have been exchanged between agents and extracts pertinent informations (Section 5.2.1),
- transformation specifying the abilities of the intruder to produce fake messages from the informations gathered so far (Section 5.2.2),
- transformations (`alice_req`, `bob_chal`, `alice_auth`, `bob_finish`) specifying the reception and sending of messages by agents; such transformations are defined as reactions between an agent and a (received) message which fulfills some conditions (Sections 5.3 to 5.4).
- function implementing a state exploration procedure which halts with a predicate `broken` checking whether a bad state is reached, hence that the search of an attack was successful.

The functions of the above two last categories come in two versions, described respectively in Sections 5.3 and 5.4.

5.1 Representing Agents, Messages and Intruder Knowledge

The three different kinds of entities (unstructured informations) found in the system states (solutions) are represented using records.

Agents. We shall distinguish the *roles*, Alice and Bob in our example, which are programs, from the *agents* executing the programs, characterized by an identifier (agent's name), a role and a bounded memory. In particular, there can be several agents for one role.
One set of records is used to define *agents*. An agent consists in:

- an *identity* (its name, note that several agents may have the same identity),
- two *stores* to memorize the session-specific values of the nonces N_a and N_b,
- a *program counter* (*pc*), which can take the value described below.

and is defined as

```
record agent = { id, ni, nr, pc};;
record alice = { dest } + agent;;
record bob = agent;;
```

All agents with either role Alice or Bob shall create a nonce and receive another one during the execution of the protocol of Section 4.1. The fields `ni` and `nr` store these two values, for Alice, `ni` stores N_a and `nr` stores N_b, and reciprocally for Bob (`ni` stands for *nonce initial*, because we can assume that each agent initially creates the nonces before starting a session of the prococol, and `nr` stands for *nonce received*).

The program counter `pc` of an agent can take the following values, according to the role:

Alice	`'REQ`	`'AUTH`	`'FINISHED`
Bob	`'CHAL`	`'WAIT`	`'FINISHED`

For Alice, `pc` = `'REQ` means that the agent is about to send the message with the corresponding label in Figure 1, and similarly for `pc` = `'AUTH` (role Alice) and `'CHAL` (Bob). For an agent playing the role of Bob, `pc` = `'WAIT` means that he is waiting for the answer of Alice to his challenge `CHAL`, and `pc` = `'FINISHED` means that the agent has completed his session of the protocol. Some records for the the various possible agent `pc` are defined as follows:

```
record req = { pc = 'REQ };;
record chal = { pc = 'CHAL };;
record auth = { pc = 'AUTH };;
record wait = { pc = 'WAIT };;
record finished = { pc = 'FINISHED };;
```

Messages. Three different kind of messages are exchanged between Alice and Bob during the protocol. The messages are characterized by the kind of information that they hold. We define a predicate for each kind of message:

```
record messageReq = { na, a, kb };;
record messageChal = { na, nb, ka };;
record messageAuth = { nb, kb };;
```

In `message_req`, `na`, `a` represent the content of the message `REQ`, and `kb` is the public key used for encryption. For the sake of simplicity, in our program, every public key or private key is represented by the identity of the owner.

Intruder Knowledge. Finally, we define a predicate for each kind of information that the intruder will be able to reveal from the whole history of exchanged messages:

```
record info_name = { name };;
record info_nonce = { nonce };;
record info_pub = { pub };;
record info_priv = { priv };;
```

Predicates are defined for each kind of message to determine the presence of a message of a given kind in the solution:

```
fun messageReqCond(a, m) = messageReq(m) & (m.kb == a.id);;
fun messageChalCond(a, m) = messageChal(m) & (m.ka == a.id)
            & (m.na == a.ni);;
fun messageAuthCond(a, m) = messageAuth(m) & (m.kb == a.id)
            & (m.nb == a.ni);;
fun PmessageReq(b, all) = exists(messageReqCond(b), all);;
fun PmessageChal(a, all) = exists(messageChalCond(a), all);;
fun PmessageAuth(a, all) = exists(messageChalCond(a), all);;
```

5.2 The Intruder Transformation Rules

Since the network is common to all agents and the intruder, he's able to read and produce new messages. This behavior is implemented by the transformations presented in the two following sections.

5.2.1 Reading and Analyzing Messages

In our approach, the existing messages are read by the intruder from the current state and they are put back unchanged. Moreover, the encrypted contents of a message are added as new known information to the state if decryption is possible. More precisely, the intruder can learn a plaintext encrypted with a public key (for instance the nonce `nb` encrypted with `kb` in `messageAuth`) only if he knows the corresponding private key.

The following transformation rules define the evolution of the knowledge of the intruder, according to the messages present in the network. There is exactly one rule for each kind of message. They will actually not generate all the informations that the intruder can extract from collected message. However, these transformations are sufficient to extract all the information needed to built messages with the rules of Section 5.2.2. For instance, if a message `m` present in the solution has type `REQ` (*i.e.* `messageReq(m)` is true), and the intruder knows the private key associated to `m.kb`, then he learns the components `m.na` and `m.a` of `m`. Theoretically, he also learns the pair `m.na, m.a` but storing such an information is useless since we assume that the intruder is able to build pairs arbitrarily.

```
trans intruder = {
m / messageReq(m) & exists((\k.(info_priv(k) &
          (k.priv == m.kb))), neighbors(m))
=> m, {nonce = m.na}, {name = m.a};
m / messageChal(m) & exists((\k.(info_priv(k) &
          (k.priv == m.ka))), neighbors(m))
=> m, {nonce = m.na}, {nonce = m.nb};
m / messageAuth(m) & exists((\k.(info_priv(k) &
          (k.priv == m.kb))), neighbors(m))
=> m, {nonce = m.nb}
};;
```

The function `neighbors` used in the transformation is a special form that returns all the neighbors of the element denoted by a pattern variable.

5.2.2 Forging Some New Messages

In the previous section, we have describe the `intruder` transformation which only reveals information according to already known messages an keys. The following transforma-

tion *produces* new fake messages from know informations in the solution. There is one transformation for each kind of message:

```
trans forge_req[acc = set:()] =
{
((k:info_pub), (n:info_name), (m:info_nonce)) as X
/ acc := {na = m.nonce, a = n.name, kb = k.pub},acc; false
=> !!(0);
_ => return(acc)
};;

trans forge_chal[acc = set:()] =
{
((k:info_pub), (n:info_nonce), (m:info_nonce)) as X
/ acc := {na=m.nonce, nb=n.nonce, ka=k.pub},
               {nb=m.nonce, na=n.nonce, ka=k.pub},acc; false
=> !!(0);
_ => return(acc)
};;


trans forge_auth[acc = set:()] =
{
((k:info_pub), (m:info_nonce)) as X
/ acc := {nb=m.nonce, kb=k.pub}, acc; false
=> !!(0);
_ => return(acc)
};;

fun forge(s) =
s, forge_req[acc=set:()](s), forge_chal[acc=set:()](s),
            forge_auth[acc=set:()](s);;

fun attack(s) = intruder(forge(s));;
```

Consider the first transformation: one should remark that, since the record made of `info_pub`, `info_name` and `info_nonce` might not be unique, we have to use the same kind of procedure described in section 3.3 to produce *all* matching triple. This way, we produce all possible fake messages knowing public keys, names of agents involved in the sessions and revealed nonces.

Function `forge`, applied to the solution `s` adds to the original solution the result of the application of the three `forge` transformations.

An attack, described by the `attack` consists in the revealing of all possibles informations by the `intruder` after having forged all possible fake messages. Actually, we'll see in the following that a *real* attack always consists in the fixpoint of the `attack` function.

5.3 First Version: Nested Multiset Rewriting

The first idea to implement the logical analysis of NSPK is to aggregate all the entities involved into the protocol in a single set acting as a chemical solution contaning the agents,

the messages and the revealed informations. The agents and the intruder will react with messages to augment the solution with new informations. All informations are in the solution at the same level. An attack on the NSPK protocol consists here in finding an interleaving of the agents actions described below such that Bob's nonce is revealed.

This approach suffers from the following problem: let S be a solution, a an agent in a state where he might reply to two different messages m_1 and m_2. The two following scenarii could happen:

1. The agent replies to both messages: to m_1 to give m_1' and to m_2 to give m_2'. Here, after the agent action, S is equal to $S \cup m_1' \cup m_2'$. In the future of the protocol, another agent may react to *both* m_1' *and* m_2' leading to an incorrect situation, even where the intruder may break the protocol and reveal the nonce.

2. The agent replies to only one of the two message: to m_i to produce m_i'. In that case, an attack might not be found because the case where the reply should have concerned the other message has not been considered. The protocol analysis is therefore too weak.

The consequence is that we have to take into account the different evolutions of the protocol that might happen when an agent receives more than one message. To model such a situation, we while make use of several sets (membranes) to localize the computation and to avoid the (possible) interference. The initial state consists in a *slice* of sets. Each set in the slice is a possible state in the protocol.

A new collection type is defined: `slice` which derives from the collection type `seq` (`slice` is then just a sequence with a different name). The empty collection of that kind is `():slice`.

```
collection slice = seq;;
```

5.3.1 The Agents

The behavior of each agent, at each possible `pc`, is described by a transformation. Since they all are (almost) the same, we only describe the behavior of `alice` at `pc 'AUTH`:

- the rule matches if there is in the solution one instance of Alice being at `pc 'AUTH` *and* with some messages addressed to her,
- in that case, all the messages for Alice are stored in the `all_messages` variable and, for each message for Alice, an answer is produced.

The transformations describing the behavior of each agent are described below:

```
trans alice_req = {
x / (req(x) & alice(x)) => (x + {pc = 'AUTH}),
     {kb = x.dest, na = x.ni, a = x.id}
};;

trans bob_chal = {
y / bob(y) & chal(y) & PmessageReq(y, neighbors(y))
=> let all_messages = filter(messageReqCond(y), neighbors(y))
in return(map((\m.((y + {pc = 'WAIT, nr = m.na}),
{ka = m.a, na = m.na, nb = y.ni}, setify(neighbors(y)))),
```

```
 all_messages))
 };;

 trans alice_auth = {
 x / auth(x) & alice(x) & PmessageChal(x, neighbors(x))
 => let all_messages = filter(messageChalCond(x), neighbors(x))
 in return(map((\m.((x + {pc = 'FINISHED}), {kb = x.dest, nb = m.nb},
 setify(neighbors(x)))),
 all_messages))
 };;

 trans bob_finish = {
 y / bob(y) & wait(y) & PmessageAuth(y, neighbors(y))
 => let all_messages = filter(messageAuthCond(y), neighbors(y))
 in return(map((\m.((y + {pc = 'FINISHED}), setify(neighbors(y)))),
all_messages))
 };;
```

Notice that the messages addressed to Alice are not removed from the solution. Since they do not appear in the pattern part of the rule, they are not matched and therefore not "consumed" from the solution.

Care has been taken in the previous transformations to generate the correct slice structure (this is the `setify(neighbors(y))` argument in the `map` of the r.h.s. of each transformation; `setify` computes the set of elements of its collection argument and the function `neighbors` returns all the neighbors of the element denoted by a pattern variable, that is, all the other values in the set).

5.3.2 Revealing a Successful Attack

A successful attack is to find in the chemical solution the nonce of Bob revealed. Since we have a slice of sets, revealing a successful attack consists in looking in each set if the nonce is revealed:

```
 fun isbroken(x) = member({nonce = 1}, x);;
 fun broken(x) = exists(isbroken, x);;
```

5.3.3 The Initial State

The initial state for the attack search consists in:

- the two agents, Alice and Bob initialized (with their respective identity, the destination of the message for Alice, initial nonces to arbitrary integer values, program counter),
- intruder knowledge (public keys for all participants and its own private key).

The initial state is a slice of sets with only one set:

```
 initial := ({id = "alice", ni = 0, nr, pc = 'REQ, dest = "charly",},
 {id = "bob", ni = 1, nr, pc = 'CHAL},
 {priv = "charly"}, {pub = "charly"}, {pub = "alice"},
{pub = "bob"}, set:()
 ):: slice:();;
```

Remark that the `nr` field is not set in the definitions: in this case, it is defined with an undefined value (and will later be set to a relevant value once a message is received).

5.3.4 Looking for an Attack

In our definition of the initial state, the number of agents is fixed and remains such. Therefore, the number of execution steps is bounded accordingly. The problem consists in finding the correct interleaving of Alice and Bob actions leading to a successful attack.

Transformation `breaks` succeeds if such an interleaving does exist. It is applied on `functions` which is the set of the transformations describing the agents behavior. The `MGS` pattern expression `(_*) as F` will match all possible permutations of the elements of `functions`. For the sake of explanation, let `F` be the sequence $[f_1, ..., f_n]$ of *one possible permutation*. The guard checks whether `broken` holds for an attack on the state $\texttt{attack}^* \circ f_1 \circ ... \circ \texttt{attack}^* \circ f_n(\texttt{initial})$.

As for the search of an attack, we still look for an interleaving leading to revealing the nonce. We now have to `map` and `flatten` the attack that follows an action of one of the agents:

```
fun fmap(f, e) = flatten(map(f, e));;

trans break = {
(_*) as F / broken(fold((\fn.\s.(fmap(attack[*], fmap(fn,s)))),
initial, F)) => return(true)
};;

functions := alice_req, alice_auth, bob_chal, bob_finish, set:();;

successful := break(functions);;
```

The search for an attack succeeds in less than a second on a AMD-1.4Ghz/LINUX computer, and reveals that the correct interleaving of functions is, as expected, $\texttt{bob\_finish} \circ \texttt{alice\_auth} \circ \texttt{bob\_chal} \circ \texttt{alice\_req}$.

5.3.5 Conclusion

In this first version, we are searching for the correct interleaving of the agents actions leading to a possible attack. We handle correctly the fact that an agent may have to react to more than one message leading to more than one evolution of the state.

Nevertheless, this method is tailored for the search of an interleaving of agents actions leading to the revelation of the nonce. This is possible because we actually know that such an interleaving *will* lead to a successful attack. We propose in the next section a more general approach where a full state space search is done.

5.4 Second Version: Full Exploration of the State Space of the Protocol

In this second version of the model, we use a more systematic approach to search for an attack on the NSPK protocol: we will use only sets (our proposal works also by replacing sets with multisets) and we do not rely on the assumption that the number of interleaving is bounded.

Given an initial state s_1 of the protocol, to produce the first evolution of the solution, our strategy is to apply the transformation of each agent to s_1 giving the four new states

$s_1^1 =$ `alice_req`(s_1), $s_1^2 =$ `bob_chal`(s_1), $s_1^3 =$ `alice_auth`(s_1), $s_1^4 =$ `bob_finish`(s_1). The application of the intruder's transformations to the $s_1^i, 1 \leq i \leq 4$ gives $s_1^{i'}, 1 \leq i \leq 4$. In the process, state s_1 is removed and replaced by the four new states $s_1^{i'}$.

The second evolution of the system starts will all four states $s_1^{i'}$, and for each $s_1^{i'}, 1 \leq i \leq 4$, produces the four new states $s_2^{ij}, 1 \leq j \leq 4$ by the application of `alice_req`, ..., `bob_finish`. The application of the intruder's transformations to those new states leads to the 16 states $s_2^{ij'}, 1 \leq i, j \leq 4$. The process continues as long as an attack is not found.

5.4.1 The Agents

The agents description is very similar to the previous one, with the difference that the `map` operations applies to `setify(all_messages)` in order to produce the correct result in term of sets. This is a minor change, since the overall design remains the same.

```
 trans alice_req = {
 x / (req(x) & alice(x)) => (x + {pc = 'AUTH}), {kb = x.dest,
     na = x.ni, a = x.id}
 };;
 trans bob_chal = {
 y / bob(y) & chal(y) & PmessageReq(y, neighbors(y))
 => let all_messages = filter(messageReqCond(y), neighbors(y))
 in return(map((\m.((y + {pc='WAIT, nr=m.na}),
 {ka=m.a, na=m.na, nb=y.ni}, setify(neighbors(y)))),
 setify(all_messages)))
 };;
 trans alice_auth = {
 x / auth(x) & alice(x) & PmessageChal(x, neighbors(x))
 => let all_messages = filter(messageChalCond(x), neighbors(x))
 in return(map((\m.((x + {pc='FINISHED}), {kb=x.dest, nb=m.nb},
setify(neighbors(x)))),
 setify(all_messages)))
 };;
 trans bob_finish = {
 y / bob(y) & wait(y) & PmessageAuth(y, neighbors(y))
 => let all_messages = filter(messageAuthCond(y), neighbors(y))
 in return(map((\m.((y + {pc='FINISHED}), setify(neighbors(y)))),
setify(all_messages)))
 };;
```

5.4.2 Revealing a Successful Attack

Since the code is the same, we do not detail it here.

5.4.3 The Initial State

The solution is now a set of sets. This is described in the initial state of the model:

```
 initial := ({id = "alice", dest = "charly", ni = 0, nr, pc = 'REQ},
 {id = "bob" , ni = 1, nr, pc = 'CHAL},
 {priv = "charly"}, {pub = "charly"}, {pub = "alice"},
{pub = "bob"}, set:()
 )::set:();;
```

5.4.4 Looking for an Attack

This time, rather than searching for an interleaving, we exhaustively apply the agents transformations to the elements of the solution. This is done in the `evolve` transformation which afterwards applies `attack`. Functions `nested` and `flat` are here to ensure that the solution, after a transition step, is correctly nested:

```
fun nested(x) = set(x) && set(hd(x));;

trans flat = { x:nested => sequify(x) };;

trans evolve = {
x => map(attack, flat(alice_req(x)::alice_auth(x)::bob_chal(x)
        ::bob_finish(x)::set:()))
};;

successful := evolve['fixpoint=broken](initial);;
```

An attack is found as the fixpoint of the evolution of the system function. The option `'fixpoint = broken` of the `evolve` function is used to stop the iterates of `evolve` once a fixpoint is reached (in the usual sense of `evolve`$(x) = x$) or `broken`(x) holds.

5.4.5 Conclusion

Here, we should face the usual combinatorial explosion of these kind of strategies. By chance, most agent transformations do not apply to the $s_i^{j^{k'}}$. Due to `MGS`'s policy on rule application, if one rule does not apply on its argument, the argument is returned unchanged. Since all the states are in a set, only a single instance of all non-matching arguments of a kind will remain. This drastically reduces the effective complexity of this approach. At the end of the process, an attack is found (in less than a second) with only 6 states in the solution, rather than the 256 awaited ones.

6 Conclusion: From `MGS` to Standard P System

In this paper, we have used the `MGS` language to describe and analyze in two different ways the NSPK protocol. It has been shown that the well-know security hole of [27] is easily (in less than one second) discovered by our state exploration procedure. The `MGS` program has been written for this special protocol, however, the principles of our modeling are general enough to envision a systematic way to derive a program for searching attacks from an abstract description of the messages of a protocol given with the notations of Section 4.1, following [11].

Section 5 shows that the programming of the NSPK analysis is particularly simple and readable in `MGS`. Moreover, it is also easy to evolve the initial analysis to more sophisticated ones. However, we want to stress that we believe that a translation of these `MGS` programs to the fundamental core mechanisms of P systems seems possible. This opinion is supported by the following remarks.

- Records are used to represent entities like agents, messages or the intruder knowledges (cf. section 5.1). A record is used here because we want to aggregate several informations into a single entity and the record data structure makes simple the

examination of this aggregate. However, each information is taken in a finite set of symbols. An alternative and more basic approach would be to use a multiset to aggregate each information part of an entity.

- We have used sets instead of multisets. It is simply more elegant and efficient (the generation of some duplicate case is spared). One can replace each set by a multiset without changing the results.

- The rule in transformation `intruder` of section 5.2.1 selects an `m` that satisfies a local property (depending only of `m`) when some global property (depending on `m` and the entire multiset) holds (namely: it exists an element k in the multiset that has some relationship with `m`). The formulation chosen to check the global property allows the generation of all the new informations in only one step. However, if we accept to generate the new informations in several steps, it is easy to check the global property using only basic multiset rewriting. The idea is to make `m` to react with k and becomes inert after that. We only have to wait until a fixpoint is reached.

- The transformations `forge_xxx` in section 5.2.2 are variations on the transformation `n_tuple` presented in section 3.3. This transformation is "non standard": the first rule updates a global variable and the second rule relies on the `return` statement. Again, this transformation can be turned into a standard multiset rewriting system by some coding. For instance, transformation:

```
trans couple = {
u:int, v:int => {fst = u, snd = v}, {fst = v, snd = u}, u, v;
u:int => {fst = u, snd = u}, u
}
```

  iterated until a fixpoint computes the set of couples of a set of integers (couples are represented by records and the integers can then be easily eliminated). The computation of the multiset of couples of a multiset of integers is slightly more difficult because the repetition of couples must be eliminated using an additional rule. However, one may notice that the rule application strategy has a big impact and, even if reaching a fixpoint has some similarities with reaching the steady state of a chemical solution, it is not completely clear how to determine that a fixpoint has been reached (which is needed for the sequencing of operations).

- Several rules make use of the `flatten` and `map` operator. A map consist in applying a function to each element in a multiset. This can be done by iterating a rewriting rule that applies the transformation and tags the results to prevent a second application. The `flatten` operator transforms a multiset of multisets into a simple multiset. It corresponds to the use of the membrane dissolving operator.

- A rule that uses the `return` construct corresponds to a non standard exit. It can be emulated using the membrane dissolving operator: when a rule with a `return` applies, then the r.h.s of the rule can be generated together with the dissolution of the enclosing membrane. The problem is to cancel all other elements of the membrane before the dissolution. This can be done using two nested membranes. In the top membrane, a rule cancels all elements, except the element to be returned (which can be tagged). A second rule is used for its dissolution. The enclosed membrane is the initial one.

- The `broken` function in section 5.3.2 is an application of the member predicate. Thus, it corresponds to the process of finding if a membrane contains an element satisfying a property, which can be easily coded.
- The `break` transformation in section 5.3.4 is also simple to translate despite its apparent complexity: the idea is to make each element to react with the other and to generate all the possible interleaving of actions. Then this interleaving is applied to the initial state and the boolean `true` value is returned if an attack is found. Again, this can be done using a fixpoint (and a bound to the number of interleaved actions can also be handled). The style used here favors the single application of a transformation.
- Other transformations correspond to very simple rules for multiset rewriting.
- All others functions are simple ones: their definitions are not recursive and most of the times they are simple predicate used to check if some data (*e.g.* a record) holds some property.

The previous remarks are indications that the `MGS` programs presented here can be translated into the P system formalism. A complete P systems solution of the problem would be very nice (as an indication of the class of P system suited to this problem, we may note that the approach taken in this paper relies crucially on the possibility to create and dissolve membranes). However, it may require a considerable amount of work to "desugar" all the available high-level `MGS` constructs. A possible outcome of such translation would be a systematic construction for the sequencing of several fixpoints computations and it is leaved to future work.

Acknowledgments

All programs in this paper have been processed with the current `MGS` version. It is available at the following URL `http://mgs.lami.univ-evry.fr`

The authors would like to thanks the reviewers for their suggestions. They are also grateful to Jean-Louis Giavitto, Julien Cohen and Antoine Spicher at LaMI for stimulating discussions and thoughtful remarks. This research is supported in part by the CNRS, the GDR ALP, the University of Évry, Genopole©, INRIA and ENS Cachan.

References

[1] Gh. Paun, *Membrane Computing. An Introduction*, Springer-Verlag, 2002.

[2] J.-L. Giavitto, Topological Collections, Transformations and Their Application to the Modeling and the Simulation of Dynamical Systems, *Rewriting Technics and Applications (RTA'03)*, LNCS 2706, Springer 2003, 208–233.

[3] J.-L. Giavitto, G. Malcolm, O. Michel, Rewriting systems and the modelling of biological systems, *Comparative and Functional Genomics*, 5, 1 (2004), 95–99.

[4] A. Atanasiu , Authentication of messages using P systems , *Membrane Computing: International Workshop, WMC-CdeA 2002, Curtea de Arges, Romania, August 19-23, 2002. Revised Papers.*, LNCS 2597, Springer, 2003, 33–42.

[5] P. Borovansky, C. Kirchner, H. Kirchner, P. E. Moreau, M. Vittek, ELAN: A Logical Framework Based on Computational Systems, *Proc. of the First Int. Workshop on Rewriting Logic, Electronic Notes in Theoretical Computer Science* 4, Elsevier, 1996.,

[6] I. Cervesato, N. Durgin, P.D. Lincoln, J.C. Mitchell, A. Scedrov, A Meta-Notation for Protocol Analysis, *12th IEEE Computer Security Foundations Workshop (CSFW'99)*, IEEE Computer Society Press, 1999, 55–69.

[7] H. Cirstea, Specifying Authentication Protocols Using ELAN, *Workshop on Modelling and Verification*, Besancon, France, 1999.

[8] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, J. F. Quesada, The Maude System, LNCS 1631, Springer 1999, 240–??.

[9] G. Denker, J. Meseguer, C. Talcott, Protocol Specification and Analysis in Maude, *Proc. of Workshop on Formal Methods and Security Protocols*, 1998.

[10] D. Dolev, A. Yao, On the Security of Pubic Key Protocols, *IEEE Transactions on Information Theory*, IT-29 (1983), 198–208.

[11] F. Jacquemard, M. Rusinowitch, L. Vigneron, Compiling and Verifying Security Protocols, *Logic for Programming and Automated Reasoning (LPAR'00)*, LNCS 1955, Springer, 2000.

[12] S.N. Krishna, R. Rama, Breaking DES using P systems, *Theoretical Computer Science*, 299, 1-2 (1003), 495–508.

[13] J.K. Millen, S.C. Clark, S.B. Freedman, The Interrogator: Protocol Security Analysis, *IEEE Transactions on Software Engineering*, SE-13, 2 (1987).

[14] R.M. Needham, M.D. Schroeder, Using Encryption for Authentication in Large Networks of Computers, *Communications of the ACM*, 21, 12 (1978), 993–999.

[15] A. Obtulowicz, Membrane Computing and One-Way Functions, *International Journal of Foundations of Computer Science*, 12, 4 (2001).

[16] M. Rusinowitch, M. Turuani, Protocol insecurity with finite number of sessions is NP-complete, *Proceedings of the 14th Computer Security Foundations Workshop (CSFW'01)*, IEEE Computer Society Press, 2001, 174–190.

[17] C. Weidenbach, Towards an Automatic Analysis of Security Protocols in First-Order Logic, *Proceedings of the 16th International Conference on Automated Deduction, CADE'99*, LNCS 1632, Springer, 1999, 378–382.

[18] J.-P. Banâtre, P. Fradet, D. Le Métayer, Gamma and the Chemical Reaction Model: Fifteen Years After, LNCS 2235, Springer 2001, 17–??.

[19] F. Jacquemard, M. Rusinowitch, L. Vigneron, Compiling and Verifying Security Protocols, book *7th International Conference on Logic for Programming and Automated Reasoning*, LNCS, Springer, 2000.

[20] D.L. Dill, A.J. Drexler, A.J. Hu, C.H. Yang, Protocol Verification as a Hardware Design Aid, *International Conference on Computer Design, VLSI in Computers and Processors* (ICCD '92), IEEE Computer Society Press, 1992, 522–525.

[21] J.L. Giavitto, O. Michel, The Topological Structures of Membrane Computing, *Fundamenta Informaticae*, 49 (2002), 107–129.

[22] J.L. Giavitto, O. Michel, Data Structure as Topological Spaces, *Proceedings of the 3nd International Conference on Unconventional Models of Computation UMC02*, LNCS 2509, Springer, 2002, 137–150.

[23] S. Peyton Jones, C. Hall, K. Hammond, W. Partain, P. Wadler, The Glasgow Haskell Compiler: A Technical Overview, *Joint Framework for Information Technology Technical Conference*, Keele, 1993.

[24] A. Huima, Efficient infinite-state analysis of security protocols, *Proceedings of FLOC'99 Workshop on Formal Methods and Security Protocols*, 1999.

[25] X. Leroy, *The Objective Caml system, release 3.07. Documentation and user's manual*, INRIA, 2004.

[26] G. Lowe, An Attack on the Needham-Schroeder Public Key Authentication Protocol, *Information Processing Letters*, 56, 3 (1995).

[27] G. Lowe, Breaking and fixing the Needham-Schroeder public-key protocol using FDR, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, LNCS, 1055, Springer 1996, 147–166.

[28] J. Mitchell, M. Mitchell, U. Stern, Automated analysis of cryptographic protocols using Murphi, *Proceedings of the IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, 1997, 141–151.

[29] C.A. Meadows, The NRL protocol analyzer: An overview, *Journal of Logic Programming*, 26, 2 (1995), 113–131.