Available online at www.sciencedirect.com
SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

ELSEVIER Electronic Notes in Theoretical Computer Science 117 (2005) 209–227
www.elsevier.com/locate/entcs

Rule-based Programming in Java For Protocol Verification

Horatiu Cirstea, Pierre-Etienne Moreau, Antoine Reilles

LORIA & University Nancy II & INRIA & CNRS
Campus Scientifique, BP 239, 54506 Vandoeuvre-lès-Nancy Cedex France

Abstract

This paper presents an approach for the development of model-checkers in a framework, called TOM, merging declarative and imperative features. We illustrate our method by specifying in TOM the Needham-Schroeder public-key protocol that aims to establish a mutual authentication between an initiator and a responder that communicate via an insecure network. We describe the behavior of the agents exchanging messages as well as the intruders and the security invariants the protocol should verify using the rewrite rules of TOM. The (depth-first or breadth-first) exploration of the search space is described using the imperative features of the language. We propose several optimizations and we compare our results to existing approaches.

Keywords: Model-checkers, protocol verification, Java, TOM.

1 Introduction

Programming with rewrite rules and strategies has been already used for describing several computational logics and transition systems. In rewrite based languages like ELAN [2] or MAUDE [4] the transitions can be described by conditional rewrite rules and the way these rules are applied is defined using strategies. For example, one can define a strategy that applies exhaustively all the rules in all the possible ways and thus verify if, starting from an initial state, a certain state can be reached (by successive transitions). It is then natural to use such rewrite based frameworks in order to model-check transition systems and, in particular, cryptographic protocols like the Needham-Schroeder public-key protocol.

doi:10.1016/j.entcs.2004.06.022

The Needham-Schroeder public-key protocol [14] has been already analyzed using several methodologies from model-checkers like FDR [15] to approaches based on theorem proving like NRL [10]. Although this protocol is described only by a few rules it has been proved insecure only in 1995 by G. Lowe [8]. After the discovery of the security problem and the correctness proof of a modified version [9], several other approaches have been used to exhibit the attack and obtain correct versions [10,12,5].

The final goal of the protocol is to provide mutual authentication between agents communicating via an insecure network. The agents use public keys distributed by a key server in order to encrypt their messages. We consider here a simplified version proposed by G. Lowe in [8] where we assume that each agent knows the public keys of all the other agents but it does not have access to their private keys.

In this paper we analyze the Needham-Schroeder public-key protocol using an approach where rewrite rules are combined with an imperative style of programming leading to a clear and flexible specification that can be executed efficiently. More precisely, we explain how this protocol can be specified in TOM [13], an extension of C and Java which adds pattern matching facilities to these languages.

The protocol is described by defining the messages exchanged between the participants and their consequent change of state. For example, an agent that wants to initiate a new communication session sends a message (in the network) and goes into a new state in which it expects a confirmation message. The protocol can thus be seen as a sequence of states of the agents (including possible intruders) and of the communication network. Therefore it is 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 state transitions of the agents and of the intruder as well as the invariants the protocol should satisfy are described by TOM conditional rewrite rules. The specification is both natural and concise, the rewrite rules describing the protocol being directly obtained from classical presentations [8]. In order to improve the efficiency, the imperative features of the language are used.

The strategy guiding these rules and thus exploring the search space is completely defined using the imperative part of TOM. We obtain a very flexible and efficient specification of depth-first and breadth-first strategies. The execution of the specification allows one, on the one hand, to describe attacks and, on the other hand, to certify the corrected version by exploring all the possible behaviors.

Section 2 briefly presents the TOM environment. The Needham-Schroeder

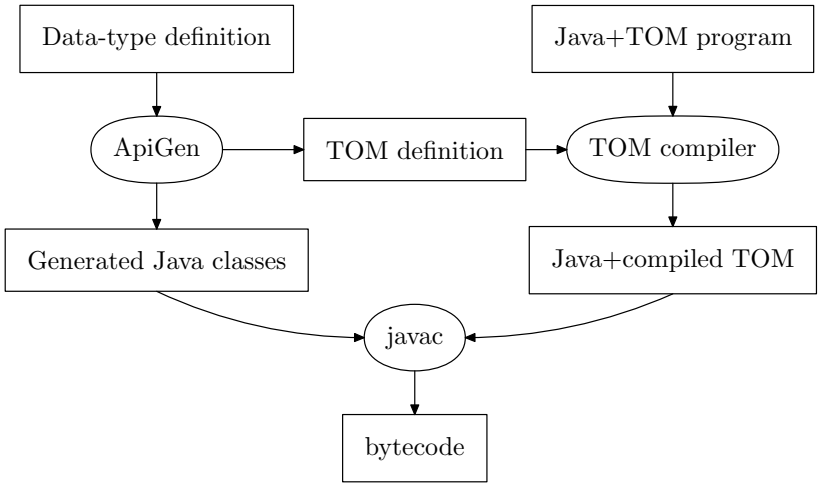

Fig. 1. The interaction between TOM and ApiGen

public-key protocol is described in Section 3 together with an attack and a corrected version. Section 4 presents the encoding of the protocol and compares our results with other approaches. The last section concludes the paper and presents perspectives to this work.

2 The TOM Environment

TOM is a language extension which adds new matching primitives to existing imperative languages. This is particularly well-suited when describing various transformations of structured entities like, for example, trees/terms and XML documents. The main originality of this system is its language and data-structure independence. From an implementation point of view, it is a compiler which accepts different *native languages* like C or Java and whose compilation process consists in translating the matching constructs into the underlying native language. Its design follows our experiences on the efficient compilation of rule-based systems [7]. For an interested reader, design and implementation issues related to TOM are presented in [13].

The language provides support for matching modulo sophisticated theories. For example, we can specify a matching modulo associativity and neutral element (also known as list-matching) that is particularly useful to model the exploration of a search space and to perform list or XML based transformations.

The TOM tool is used in conjunction with the ApiGen system which is a generator of abstract syntax tree implementations [16] (see Figure 1). Starting

from an input datatype definition, ApiGen generates a datatype in Java and a definition of this datatype as input for TOM. The users can then write code using the match construct on the generated Abstract Syntax Tree classes and TOM compiles this to normal Java. The generated code is characterized by strong typing combined with a generic interface and by maximal sub-term sharing for memory efficiency and fast equality checking.

For expository reasons, we assume that TOM only adds two new constructs: `%match` and *build* (‘). The first construct is similar to the `match` primitive of ML and related languages: given a term (called subject) and a list of pairs pattern-action, the `match` primitive selects a pattern that matches the subject and performs the associated action. This construct may thus be seen as an extension of the classical `switch/case` construct. The second construct is a mechanism that allows one to easily build ground terms over a defined signature. This operator, called *build*, is followed by a well-formed term built over constructors, variables and function calls. This term is written in prefix notation.

In order to give a better understanding of TOM's features, let us consider a simple symbolic predicate (`greaterThan`) defined on Peano integers built using the `zero` and `suc` symbols. When using Java as the native language, the comparison of two integers can be described in the following way:

```
boolean greaterThan(Nat t1, Nat t2) {
  %match(Nat t1, Nat t2) {
    x,x           -> { return true; }
    suc(_),zero() -> { return true; }
    zero(),suc(_) -> { return false; }
    suc(x),suc(y) -> { return greaterThan(x,y); }
  }
}
```

This example should be read as follows: given two terms `t1` and `t2` (that represent Peano integers), the evaluation of `greaterThan` returns `true` if `t1` is greater or equal to `t2`. This is implemented by (non-linear) pattern matching (first pattern) and anonymous variables (second and third patterns). The reader should note that variables do not need to be declared: their type is automatically inferred from the definitions of the operators using them. To distinguish a constant from a variable (*e.g.* the constant `zero`), empty braces could be used.

As mentioned previously, an important feature of TOM is to support list matching, also known as associative matching with neutral element. Let us consider the associative operator `conc` used for building lists of naturals (`NatList`). In TOM, an associative operator is variadic and each sub-term could be either of sort *element* or *list* (respectively `Nat` or `NatList` in our case). To illustrate the expressiveness of associative matching, we can define a sorting algorithm as follows:

```
public NatList sort(NatList l) {
  %match(NatList l) {
    conc(X1*,x,X2*,y,X3*) -> {
```

```
      if(greaterThan(x,y)) { return ‘sort(conc(X1*,y,X2*,x,X3*)); }
    }
    _ -> { return l; }
  }
}
```

In this example, one can remark the use of *list variables*, annotated by a '`*`': such a variable should be instantiated by a (possibly empty) list. Given a partially sorted list, the `sort` function tries to find two elements `x` and `y` such that `x` is greater than `y`. If two such elements exist, they are swapped and the `sort` function is recursively applied. When the list is sorted this pattern cannot by found in the list and the next pattern is tried. In fact, this pattern imposes no restrictions on its subject and thus the corresponding action is triggered and the sorted list `l` is returned.

3 The Needham-Schroeder Public-Key Protocol

The Needham-Schroeder public-key protocol [14] 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, i.e. by A.

In this paper we only consider the simplified version proposed in [8] assuming that each agent knows at the beginning the public keys of all the other agents.

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

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 agent A by N_A.

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 together with its identity and this message is encrypted with its key $K(B)$. When such a message is received by B, the agent 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 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 the message and assumes that a session has been established with B. The agent A sends the nonce N_B back to B and when receiving this last message B assumes that a session

has been established 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 a user of the communication network and so, it can initiate standard sessions with the other agents and it can respond to messages sent by the other agents. The intruder can intercept any message 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 (fake) messages. The intercepted messages that can not be decrypted by the intruder are replayed as they are.

An attack on the protocol is presented in [8] where the intruder impersonates an 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 $I.n$, $II.n$ represent steps in the first and second session respectively and $I(A)$ represents the intruder impersonating the agent A:

$$
\begin{array}{lll}
\text{I.1.} & A \rightarrow I & : \{N_A, A\}_{K(I)} \\
\text{II.1.} & I(A) \rightarrow B & : \{N_A, A\}_{K(B)} \\
\text{II.2.} & B \rightarrow I(A) & : \{N_A, N_B\}_{K(A)} \\
\text{I.2.} & I \rightarrow A & : \{N_A, N_B\}_{K(A)} \\
\text{I.3.} & A \rightarrow I & : \{N_B\}_{K(I)} \\
\text{II.3.} & I(A) \rightarrow B & : \{N_B\}_{K(B)}
\end{array}
$$

The agent A tries to establish a session with the intruder I by sending 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 it 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 a session has been established with A while this session has in fact been established with the intruder.

In the correction shown sound in [9] the responder introduces its identity in the encrypted part of the message and the initiator checks if this identity corresponds to the agent it has started the session with. The second step of the protocol is then modified and the corrected protocol becomes:

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

4 Encoding the Needham-Schroeder Protocol in TOM

In this section we give a description of the protocol in TOM. The TOM rewrite rules correspond to transitions of agents from one state to another after sending and/or receiving messages. The strategies guiding the application of these rewrite rules describe a form of model-checking in which all the possible behaviors are explored. This exploration can be done in a depth-first or breadth-first manner. The former approach is more efficient for detecting an attack when it exists and thus when the exploration of the search space is not complete, while the latter approach is more efficient when a corrected version is analyzed and thus all the possible states are eventually explored. The specification technique we have proposed in this paper allows us to explore a finite state space and thus, it is well suited for detecting the attacks against the protocol and not for proving its correctness in an unbounded model.

We present first the encoding of the different entities taking part in the protocol such as the agents exchanging messages, the intruder(s) intercepting these messages and the network used as communication support. As we will see later on, the number of agents (initiators and responders) is not hard-coded in the implementation but this number should be specified when checking the specification. This allows us, on the one hand, to show the expressiveness of a TOM specification and, on the other hand, to compare the results, in terms of efficiency, with other approaches like ELAN [3] and Murφ [6].

Each agent is characterized by a state that is modified when it sends or receives a message. The transitions of the agents and, more generally, of the global environment, from one state to another are specified by TOM rewrite rules. All the possible behaviors are then explored by trying to apply each of the defined rules on the current state.

As mentioned previously, the datatypes are defined abstractly using many-sorted signatures and, more precisely, we use ApiGen to generate a Java implementation which ensures maximal sub-term sharing. A complete specification of the agents, messages and the network is described in section 4.1. In section 4.2, we show how TOM can be used in order to describe concisely the behavior of the agents (using transition rules) and the exploration of the search space (using Java native code).

4.1 Data Structures

The agents (that can be either initiators or responders) are defined by three *attributes*: the identity `id`, the current state `state` and a nonce `nonce` they have created. Objects of type `Agent` can be built using the constructor `agent` as specified by the following ApiGen definition:

```
Agent ::= agent(id:AgentId, state:AgentState, nonce:Nonce)
```

The identity of an agent is either a name defined explicitly or a generic sender or receiver:

```
AgentId ::= alice | bob | dai
          | sender(number:int)
          | receiver(number:int)
```

There are three possible values for `AgentState` states. An agent is in the state `SLEEP` if it has neither 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. All these states are defined explicitly:

```
AgentState ::= SLEEP | WAIT | COMMIT
```

The nonces created by the agents are normally random numbers that identify a session but in our encoding we consider nonces that keep track of the agents using them:

```
Nonce ::= N(id1:AgentId, id2:AgentId)
```

When using this kind of encoding, if the agent that generated a nonce memorizes it then, the agent knows at each moment who is the agent with whom it is establishing a session. For example, the agent `alice` generates and memorizes the nonce `N(alice,bob)` when trying to establish a session with `bob`. A dummy nonce `DN` is defined as `N(dai,dai)` where `dai` is a dummy agent identity.

The messages exchanged between agents contain the identities of the sender (`src`) and of the receiver (`dst`) of the message, the encryption key (`key`), two nonces (`nonce1` and `nonce2`) and the explicit address of the sender:

```
Message ::= msg(src:AgentId, dst:AgentId, key:Key,
                nonce1:Nonce, nonce2:Nonce, adr:Address)
```

The address of an agent as well as its private key depend only on its identity:

```
Address ::= A(id:AgentId)
Key     ::= K(id:AgentId)
```

The header of the message containing the source and destination addresses are not encrypted (and thus can be faked by an intruder) while the body of the message containing all the other information is encrypted with the given key and can be decrypted only by the owner of the private key.

The communication network can be seen as a list of messages and therefore we introduce, using the list constructor of ApiGen, the type `ListMessage` as a list of objects of type `Message`. Similarly, we can define lists of agents (`ListAgent`) and of nonces (`ListNonce`).

The intruder is a special agent that does not only participate to normal communications but can as well intercept and create (fake) messages. An intruder stores (in `nonces`) the nonces of the intercepted messages that are encrypted with its key and (in `messages`) the messages it cannot decrypt.

```
Intruder ::= intruder(id:AgentId, nonces:ListNonce,
                      messages:ListMessage)
```

The rewrite rules presented in the next section 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. There are also two particular states representing a configuration obtained after a successful attack on the current session and an error during the message exchange respectively:

```
State ::= state(senders:ListAgent, receivers:ListAgent,
                intruder:Intruder, network:ListMessage)
        | ATTACK | ERROR
```

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.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 `x` is in the state `SLEEP`, it can initiate a session with a (randomly chosen) responder `y` by sending a message containing a nonce and its address and encoded with the public key of `y`. We can describe the global state change by the following rule:

$$\begin{array}{ll} \text{senders :} & S_1^* + \texttt{agent(x,SLEEP,\_)} + S_2^* \\ \text{receivers :} & R_1^* + \texttt{agent(y,\_,\_)} + R_2^* \\ \text{intruder :} & I \\ \text{network :} & M^* \end{array} \Longrightarrow \begin{array}{l} S_1^* + \texttt{agent(x,WAIT,N(x,y))} + S_2^* \\ R_1^* + \texttt{agent(y,\_,\_)} + R_2^* \\ I \\ M^* + \texttt{msg(x,y,K(y),N(x,y),DN(),A(x))} \end{array}$$

$$\texttt{if sizeMessage}(M^*) \texttt{ < maxMessagesInNetwork}$$

where the symbol "_" represents an insignificant value *w.r.t* the current transformation and symbols of the form S_1^* are place-holders for (possibly empty) lists of values. For example, when writing $S_1^* + \texttt{agent(x,SLEEP,\_)} + S_2^*$ an agent in the state `SLEEP` is non-deterministically selected from the list of senders and its identity is stored in the variable `x`. For expository reasons, we use an abuse of notation in the rule above: "_" is used in the right-hand side of the rule to denote the fact that the corresponding value is not modified by the rule. To be rigorous, we should have introduced named variables in the left-hand side and re-used them in the right-hand side of the rule.

The initiator sends a nonce `N(x,y)` and its address (identity) encrypted with the public key of the responder. 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, *i.e.* inserted in the list of messages representing the network, only if the capacity of the network (`maxMessagesInNetwork`) is not exceeded. As we will see in what follows all these messages can be intercepted, stored and forwarded (in a random but systematic way) by the intruder that listens the network. Consequently, considering only a network of size one leads to equivalent results but this extension allows us to show the extensibility of the approach and to provide more benchmarks.

Starting from this definition, the TOM rule follows immediately:

```
state(concAgent(S1*,agent(x,SLEEP(),_),S2*),
      dst@concAgent(_*,agent(y,_,_),_*), I, M*) -> {
 if(sizeMessage(‘M*) < maxMessagesInNetwork) {
   State state = ‘state(concAgent(S1*,agent(x,WAIT(),N(x,y)),S2*),
                  dst,I,
                  concMessage(msg(x,y,K(y),N(x,y),DN(),A(x)),M*));
   set.add(state);
  }
}
```

In the above rewrite rule `x` and `y` are variables of type `AgentId` representing the identity of the initiator and the identity of the responder respectively. Since `concAgent` is a list constructor (of `AgentList`), an associative matching is employed when dealing with this operator and thus the identities of the two agents, `x` and `y`, are selected non-deterministically from the two lists

of initiators and receivers respectively. In fact, all possible solutions of the corresponding matching will be considered when exploring exhaustively the search space. We use in this rule several syntactic features provided by TOM: the "`dst@`" annotation allows us to give a name to a matched sub-term. As any other variable, this can can be re-used in the action part to build a new term. The notations '`_`' and '`_*`' are used for anonymous variables and list-variables respectively. The latter one can be instantiated by the empty list.

The sender having sent the message switches to the state `WAIT` where it waits for a response. The list of responders `dst` is left unchanged (since only used for selecting a possible destination).

A second similar rule considers the case where the destination of the initial message is the intruder instead of a responder.

$$\left\|\begin{array}{ll}\text{senders}: & S_1^* + \texttt{agent(x,SLEEP,\_)} + S_2^* \\ \text{receivers}: & R^* \\ \text{intruder}: & \texttt{intruder(w,\_,\_)} \\ \text{network}: & M^*\end{array}\right\| \Longrightarrow \left\|\begin{array}{l} S_1^* + \texttt{agent(x,WAIT,N(x,w))} + S_2^* \\ R^* \\ \texttt{intruder(w,\_,\_)} \\ M^* + \texttt{msg(x,w,K(w),N(x,w),DN(),A(x))}\end{array}\right\|$$

$$\texttt{if sizeMessage}(M^*) < \texttt{maxMessagesInNetwork}$$

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 encrypted with its key. In this latter case, the responder sends the second message of the protocol to the initiator and goes in the state `WAIT` where it waits for the final acknowledgment. For efficiency reasons we can suppose that all messages are intercepted by the intruder and only afterwards forwarded to the concerned agents. We implement this optimization proposed in [11] and we consider that the initiator and the responder listen only to messages coming from the intruder:

$$\left\|\begin{array}{l} S^* \\ R_1^* + \texttt{agent(y,SLEEP,\_)} + R_2^* \\ \texttt{intruder(w,\_,\_)} \\ M_1^* + \texttt{msg(w,y,K(y)},N_1\texttt{,\_,A(z))} + M_2^*\end{array}\right\| \Longrightarrow \left\|\begin{array}{l} S^* \\ R_1^* + \texttt{agent(y,WAIT,N(y,z))} + R_2^* \\ \texttt{intruder(w,\_,\_)} \\ M_1^* + \texttt{msg(y,z,K(z)},N_1\texttt{,N(y,z),A(y))} + M_2^*\end{array}\right\|$$

Once again, due to the associative matching used for the message lists, the position of the message in the network (*i.e.* the moment when the message was sent) is not important. Notice that the message should not only have `y` as destination but it should be also encrypted with its public key. The encrypted identity of the sender of the message will be used in order to build the response to this message.

Two other rewrite rules describe the other steps of the protocol. For these rules as well we consider only messages that have been intercepted and forwarded by the intruder. The specification of the original version of the protocol can be easily adapted in order to obtain the corrected one. Since the agents

always send their identity, all we have to do is to add a test checking that the correct identity have been provided in the rule corresponding to the second step of the protocol.

When an initiator `x` and a responder `y` have reached the state `COMMIT` at the end of a correct session, 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 should be considered as a normal agent that can not only participate in normal sessions but that also tries to break the security of the protocol by obtaining information that are supposed to be confidential.

For doing so, the intruder eavesdrops the network that serves as communication support and thus, common and accessible to all the agents. Hence, the intruder observes and intercepts any message in the network, decrypts messages encrypted with its key and stores the obtained information, replays intercepted messages and generates fake messages starting from the information it has gained.

The intruder intercepts all the messages in the network except for the messages generated by itself and stores or decrypts them. We just present here the rule corresponding to a message the intruder can decrypt:

$$\begin{array}{l}\mathsf{senders}: \\ \mathsf{receivers}: \\ \mathsf{intruder}: \\ \mathsf{network}: \\ \qquad \texttt{if x} \neq \texttt{y}\end{array} \left\| \begin{array}{l} S^* \\ R^* \\ \texttt{intruder(x,}L^*\texttt{,\_)} \\ M_1^* + \texttt{msg(y,\_,K(x),}N_1\texttt{,}N_2\texttt{,\_)} + M_2^* \end{array} \right\| \Longrightarrow \left\| \begin{array}{l} S^* \\ R^* \\ \texttt{intruder(x,}L^* + N_1 + N_2\texttt{,\_)} \\ M_1^* + M_2^* \end{array} \right\|$$

We have to check that the source of the messages intercepted by the intruder is not the intruder itself since otherwise the intruder can engage in an infinite loop where it generates and intercepts the same message forever.

The case where the intruder cannot decode the message and thus just stores it, is handled in a similar and simpler way. These (encrypted) 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. The TOM rule describing this behavior is similar to the previous ones: the message to be sent and its destination are selected non-deterministically from the list of stored messages and from the list of senders/receivers respectively.

The intruder also tries to impersonate all the agents by sending fake messages built using the nonces previously obtained. This time both the destination of the message and the fake address in the encrypted part of the message are generated non-deterministically:

$$\left\|\begin{array}{ll}\text{senders}: & S^* \\ \text{receivers}: & R^* \\ \text{intruder}: & \texttt{intruder(w,}L^*\texttt{,\_)} \\ \text{network}: & M^*\end{array}\right\| \Longrightarrow \left\|\begin{array}{l} S^* \\ R^* \\ \texttt{intruder(w,}L^*\texttt{,\_)} \\ M^* + \texttt{msg(w,y,K(y),resp,init,A(y'))}\end{array}\right\|$$

$$\texttt{if sizeMessage(}M^*\texttt{) < maxMessagesInNetwork}$$
$$\texttt{if msg(w,y,K(y),resp,init,A(y'))} \notin M^*$$
$$\texttt{where (agent(y,\_,\_),agent(y',\_,\_))} \in ((S^* + R^*) \times (S^* + R^*))$$
$$\texttt{where (resp, init)} \in (L^* \times L^*)$$

The fake source `y'` and destination `y` of the generated message are selected non-deterministically from the list obtained as the concatenation of the lists of senders and receivers. The two nonces of the message are obtained similarly from the list of nonces the intruder has stored. The generated message is sent only if it has not already been done previously.

The conditions involving cartesian products can be encoded easily in TOM by list-matching the needed (agent) expressions from the same list $(S^* + R^*)$. The two lists are matched separately obtaining thus all the possible combinations source/destination. In the following rule we use a syntactic extension of TOM which allows us to omit the name of the list operator when there is no ambiguity and to abbreviate `concAgent(S*)`, `concMessage(S*)` and `concNonce(S*)` by `(S*)`:

```
state(S*, R*, intru@intruder(w,nonces,ll), M*) -> {
  if(sizeMessage(‘M*) < maxMessagesInNetwork) {
    ListAgent SR = ‘(S*,R*);
    %match(ListAgent SR, ListAgent SR,
           ListNonce nonces, ListNonce nonces) {
      (_*,agent(y,_,_),_*), (_*,agent(yp,_,_),_*),
      (_*,init,_*), (_*,resp,_*)  -> {
        Message message = ‘msg(w,y,K(y),resp,init,A(yp));
        if(!existMessage(message,‘M*)) {
          State state = ‘state(S*,R*,intru,(message,M*));
          set.add(state);
} } } } }
```

Due to the associative matching used for selecting nonces and agents, every time this rule is applied either a different message is generated or the message is sent to a different destination. This way the intruder generates all the possible messages and sends them (if not already sent) to all the possible agents. A similar rule describes the construction of messages containing only one nonce.

4.2.3 The Invariants

The invariants used to specify the correctness conditions of the protocol are expressed as well by rewrite rules. We have to check the authenticity of the responder, that is, to check if the agent that replied to the initiator is indeed the correct one (and not the intruder, for example). Similarly, we should verify that the communication was established with the agent that initiated it, that is, the authenticity of the initiator.

Instead of specifying the conditions that should be respected by the agents having established a communication we define their negation that can be seen as a violation of the authenticity of the protocol.

The rewrite rule describing the negation of the first invariant and thus, the possibility of an attack checks if an initiator has concluded the protocol (*i.e.* is in the state `COMMIT`) while the corresponding responder (that is not an intruder) has neither committed nor sent an appropriate response (and thus ready for committing). More precisely, an attack is possible if there exists an initiator in the state `COMMIT` such that the corresponding responder is neither in the state `COMMIT` nor in the state `WAIT` (waiting for an acknowledgment from the initiator).

$$\left\|\begin{array}{ll} \text{senders}: & S_1^* + \texttt{agent(x,COMMIT,N(x,y))} + S_2^* \\ \text{receivers}: & R^* \\ \text{intruder}: & \texttt{intruder(w,\_,\_)} \\ \text{network}: & M^* \end{array}\right\| \Longrightarrow \texttt{ATTACK}$$

$$\texttt{if } \texttt{y} \neq \texttt{w}$$
$$\texttt{if agent(y,WAIT,N(y,x))} \notin R^*$$
$$\texttt{if agent(y,COMMIT,N(y,x))} \notin R^*$$

For the authenticity of the initiator we proceed similarly and we verify if a responder is in the state `COMMIT` while the corresponding initiator has not reached this state yet:

$$\left\|\begin{array}{ll} \text{senders}: & S^* \\ \text{receivers}: & R_1^* + \texttt{agent(y,COMMIT,N(y,x))} + R_2^* \\ \text{intruder}: & \texttt{intruder(w,\_,\_)} \\ \text{network}: & M^* \end{array}\right\| \Longrightarrow \texttt{ATTACK}$$

$$\texttt{if } \texttt{x} \neq \texttt{w}$$
$$\texttt{if agent(x,COMMIT,N(x,y))} \notin S^*$$

If one of 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 first rewrite rule can be applied we can conclude that the responder has been impersonated (by the intruder) and if the second rewrite rule can be applied we can conclude that the initiator has been impersonated (by the intruder).

4.3 Exploring the Search Space

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.

In a previous implementation written in ELAN, the backtracking mechanism was used to explore the search space: when a message does not lead to an attack, a backtrack is performed and a new destination and/or address are selected. 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 (if not already sent) to all the possible agents. One difficulty, when using a backtracking approach, is to model a strategy other than the depth-first one. In our case, since several transitions can lead to the same state, it is interesting to explore the search space with a breadth-first strategy and to delete the doubles reducing this way the number of states to explore.

When using TOM, contrary to systems like ELAN or Prolog, the search space has to be handled explicitly. On one side this leads to more complex implementations, but on the other side, this gives more flexibility for the description of search strategies which are thus specific and optimized. In the current implementation, we start with a set of states which contains the initial state. Then, for each state of the set, we compute the set of successors: all the states that can be reached by applying a single transition step. This process is repeated until either an attack is found in the set of states or a fix-point is reached: no more transition can be performed.

Assuming that `start` is the initial state where all the agents are in `SLEEP` state, that `set1` and `set2` are Java collections and that `computeSuccessors` is a function that takes a state as input and computes a set of successors, the presented strategy can be implemented as follows:

```
public boolean breadthSearch(State start) {
  set1.add(start);
  while(!set1.isEmpty()) {
    Iterator it = set1.iterator();
    while(it.hasNext()) { computeSuccessors(it.next(),set2); }
    if(set2.contains(ATTACK)) { return true; }
    set1 = set2;
  }
  return false;
}
```

4.4 Comparing the TOM *Implementation to Existing Approaches*

We compare the TOM specification to a similar one defined in the model-checker Murφ [11], and to another one described in the rewrite based language ELAN [3]. We have chosen the latter one essentially for historical reasons and the former one due to the indirect comparisons it provides. These approaches are very similar to the TOM one in the way they describe the protocol, making thus the comparison of the efficiency of these implementations quite reliable. Of course, several other implementations of the protocol are available (*e.g.* in Spin or Maude) but a direct comparison have not been realized due to significant differences in the corresponding specifications.

In ELAN the rules describing the protocol and the strategy guiding these rules are defined at the same level. We obtain thus a uniform specification but, although the strategy language is fairly powerful and the definition of a depth-first search method is natural and simple, the description of breadth-first strategies is complex and needs a modification of the existing rules. Several operators like, for example, the ones used for building the list of messages, are declared as associative-commutative avoiding thus some implicit manipulation like sorting the messages.

The rules used in the Murφ implementation are similar to the ones used in the ELAN and TOM specification but the number of interacting agents is built-in. There are two possible built-in verification strategies (depth-first and breadth-first) that are specified at execution time. Several optimizations like, for example, a reduction by symmetry that allows one to cut (significantly) the search space, are available.

First, we can compare the ease of modeling of these approaches. For this, we need to use some kind of metric like the one used in [1]. As the metric reflects only part of the story, care must be taken in interpreting the results. We will compare the number of lines in the different parts of the specification. This is not problem free, as the number of lines is influenced by layout conventions, and there must be compromise between code size, clarity of the specification and development time.

Lines	*Total*	*Stripped*	*Data*	*Rules*	*Control*
Murφ	449	323	47	255	Built-in
ELAN	439	250	49	154	35
TOM	563	381	30	258	75

Total measures the lines in the entire specification, including comments,

Number of		Network size	Time (seconds)		
send	rcv		Murφ	ELAN	TOM
1	1	1	0.11	0.01	0.08
1	1	2	1.47	0.11	0.34
1	1	3	38.90	1.20	0.66
2	1	1	2.76	0.49	0.51
2	1	2	77.01	-	7.88
2	2	1	69.50	39.04	3.52
3	1	1	13.69	34.90	2.62
3	2	1	703.76	-	69.59
2	2	2	6456.53	-	234.54

Fig. 2. Comparison of the Murφ, ELAN and TOM implementations.

while *Stripped* is the size of the specification code, without comments. The other columns compare the size of the declarations for the data structures, of the model definition, and of the control procedures.

Note that the main difference for this metric is the size of the control procedures. As mentioned before, in Murφ the exploration strategy is built-in and the rules are applied repeatedly in a depth-first or breadth-first manner. In ELAN user-defined rewriting strategies are used while in TOM the control is expressed using the native language Java. This leads to a fine grained and very flexible control in TOM, but the counterpart is the length of the control procedures.

This fine grained control over the execution of the model leads to an optimized execution of the model. Tests are performed on a 1,4 GHz Xeon workstation with 512MB memory and 256KB cache. For these tests, we consider the fixed version of the protocol, as described in [9]. We explore thus the entire search space, showing that there are no attacks in the corrected version of the protocol when variable numbers of agents and messages in the network are considered.

The results presented in Figure 2 clearly show the interest of the TOM approach. On small problems, the efficiency is comparable to Murφ and ELAN. On bigger examples (2 senders, 2 receivers and 1 message in the network for example), the experimental results show that TOM is 10 to 20 times faster than ELAN and Murφ.

Let us remind that the presented implementation is written in Java, compared to the Murφ and ELAN versions which are respectively written in C++ and C. Therefore, the gain does not come from low level implementation details: the performance improvement comes from the way the search space is explored. In ELAN, due to backtracking, a depth-first search strategy is applied. In Murφ, we used a breadth-first search strategy combined with symmetry reduction optimizations similar to our sorting procedures. In TOM, we used the very simple breadth-first search strategy described in Section 4.3 combined with a maximally-shared representation of states (obtained implicitly thanks to ApiGen). This allows us to efficiently compare and eliminate redundant states and thus to reduce the search space. Since TOM does not support associative-commutative matching, two equivalent states could be represented differently (the order in which messages and nonces are stored is important). To improve the sharing of such states, we use sorted lists of messages and nonces in such a way that we manipulate canonical representations. Thus, the comparison of two states can be performed in constant time, by comparing pointers.

The results presented in Table 2 also show that for some parameters (3 senders and 2 receivers for example) the ELAN version cannot verify the correctness of the protocol within 512MB of memory.

5 Conclusion

We have proposed an approach merging declarative and imperative paradigms for the verification of the Needham-Schroeder public-key protocol. The rules describing the protocol are easily represented by TOM conditional rewrite rules. The built-in list-matching available in TOM allows us to express and easily handle the random selection of agents from a set of agents or of a message from a set of messages.

The rewrite rules of the implementation describe the behavior of the agents and of the intruder as well as the invariants to be verified. The number of senders and receivers as well as the maximum number of messages in the network is not fixed but can be given at verification time.

We obtain this way a concise specification which is both flexible and efficient. The performances of our specification are better than those obtained with other tools.

An obvious continuation of this work would be, on the one hand, the improvement of our implementation methods like, for example, better associative or associative-commutative matching algorithms. On the other hand, more complex protocols should be specified and verified with such a hybrid

approach for modeling strategies. The comparison of our technique in terms of expressiveness and efficiency should be extended to other tools like Spin, for example.

References

[1] D. Basin and G. Denker. Maude versus haskell: an experimental comparison in security protocol analysis. In K. Futatsugi, editor, *Electronic Notes in Theoretical Computer Science*, volume 36. Elsevier Science Publishers, 2001.

[2] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of ELAN. In *2nd International Workshop on Rewriting Logic and its Applications*, volume 15, Pont-à-Mousson (France), September 1998. Electronic Notes in Theoretical Computer Science.

[3] H. Cirstea. Specifying Authentication Protocols Using Rewriting and Strategies. In *Third International Symposium on Practical Aspects of Declarative Languages*, volume 1990 of *LNCS*, pages 138–153, Las Vegas, USA, March 2001.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The maude 2.0 system. In R. Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, number 2706 in Lecture Notes in Computer Science, pages 76–87. Springer-Verlag, June 2003.

[5] G. Denker. From rewrite theories to temporal logic theories. In *2nd International Workshop on Rewriting Logic and its Applications*, volume 15, Pont-à-Mousson (France), September 1998. Electronic Notes in Theoretical Computer Science.

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

[7] H. Kirchner and P.-E. Moreau. Promoting rewriting to a programming language: A compiler for non-deterministic rewrite programs in associative-commutative theories. *Journal of Functional Programming*, 11(2):207–251, 2001.

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

[9] 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.

[10] C. 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.

[11] 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.

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

[13] P.-E. Moreau, C. Ringeissen, and M. Vittek. A Pattern Matching Compiler for Multiple Target Languages. In G. Hedin, editor, *12th Conference on Compiler Construction, Warsaw (Poland)*, volume 2622 of *LNCS*, pages 61–76. Springer-Verlag, May 2003.

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

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

[16] M. van den Brand, P.-E. Moreau, and J. Vinju. A generator of efficient strongly typed abstract syntax trees in java. Technical report SEN-E0306, ISSN 1386-369X, CWI, Amsterdam (Holland), November 2003.