

Using tccp for the Specification of Communication Protocols*

Alexei Lescaylle Alicia Villanueva

alescaylle@dsic.upv.es villanue@dsic.upv.es

DSIC, UPV

DSIC, UPV

Abstract

The automatic analysis of cryptographic protocols by using formal methods on concurrent languages is a subject widely treated in the literature. From its beginning in the decade of the 70s, the field has been gaining maturity and consolidation. The Timed Concurrent Constraint Language (tccp in short) is a declarative concurrent programming language which, like other concurrent languages, allows us to intuitively model concurrent and reactive systems. In particular, cryptographic protocols can be specified in a very compact way thanks to certain features such as the non-determinism and concurrency, which are necessary to model such kind of systems. In this work, we show a method to specify cryptographic protocols using tccp. We have defined specifications for the actions that principals usually do during a protocol run. Moreover, we have defined a hostile environment in which agents run the protocol. The environment is actually the representation of the popular intruder model of Dolev-Yao. We use the Needham-Schroeder public key authentication protocol to illustrate the approach.

1 Introduction

A cryptographic protocol is a protocol that uses cryptography primitives to authenticate agents (principals) over a network. The network can be assumed as a hostile environment controlled by an intruder who can read, mod-

ify and delete the messages sent by honest principals. Thus, when a protocol is executed, dishonest intruders can interfere and modify the intended behavior. Usually, *security of protocols* must be ensured, but the term *security* is often too vague. It can refer to privacy, data integrity, etc.

The automatic analysis of cryptographic protocols by using formal methods on concurrent languages is a subject widely treated in the literature. From its beginning in the decade of the 70s, the field has been gaining much maturity and consolidation. A concurrent language can enable the designer to write unambiguous, clear, and concise specification which can result in an executable specification. Moreover, formal languages can be used to analyze systems thanks to its formal semantics, that can be handled by formal methods. In fact, formal methods are suitable techniques for solving the problem of verifying the correct behavior of cryptographic protocols. The Timed Concurrent Constraint Language (tccp in short, [8]) is a declarative concurrent programming language which allows us to intuitively model concurrent and reactive systems. In particular, cryptographic protocols can be specified in a very compact way thanks to certain features such as the non-deterministic behavior and concurrency, which are necessary to model such kind of systems.

Some techniques used in the automatic analysis of protocols (not only cryptographic ones) are mentioned below:

- State Machine Models [16, 17],
- approaches and models based on the π -Calculus [2],
- methods based on the belief logic [10], or

*This work has been partially supported by the EU (FEDER) and the Spanish MEC under grants TIN2004-7943-C04-02, HA2006-2007, and by Valencian Government under grant GV06/285.

- methods based on theorem proving [15, 6, 21, 9, 24].

Other techniques such as Rewriting [12], Typing [1] or Abstract Interpretation [7, 14, 18] are also applied to the verification of protocols.

In [23], Shyamasundar shows his approach by using the language ESTEREL [5] to analyze the TMN protocol. ESTEREL is used both, to specify principals (initiator, responder, server and intruder attacks), and to verify security properties.

In this paper we specify, by using *tccp*, the actions that both, honest principals and intruders can perform. Moreover, we use as running example the Needham-Schroeder public key authentication protocol [19], and the popular intruder model of Dolev-Yao [13].

The remainder of this paper is organized as follow. In Section 2, we show an introduction to cryptographic protocols, explaining the Needham-Schroeder public key authentication protocol. An overview of *tccp* and a new agent for the language are given in Section 3. Sections 4 and 5 are devoted to the specification of principals and the intruder. Finally, in Section 6 we conclude and discuss future work.

2 Cryptographic Protocols

A protocol can be seen as a recipe that enables the connection, communication and data exchange through messages among two or more entities (*principals*), which interact to achieve some goal. The principals can be users, hosts or processes [11] behaving as initiators or responders in the protocol. Therefore, the protocol describes how messages can be sent from one principal to other, and how the participants react when receiving a message during the protocol run. Messages consist of atoms (principals' names, nonces, or other data) and they may be encrypted with a key in order to guarantee confidentiality. A nonce is a value randomly generated by a principal which is usually sent to other principal in order to ensure the freshness of messages.

In the following, we illustrate the classical notation to specify a protocol:

- A and B are names of principals (Alice and Bob in the literature).
- K_{AP} and K_{AS} denote A 's public and secret keys, respectively.
- K_{BP} and K_{BS} denote B 's public and secret keys, respectively.
- N_A and N_B are nonces generated by A and B , respectively.
- $A \rightarrow B : M$ means that A sends to B the message M .

We show in Figure 1 the specification of the Needham-Schroeder public key authentication protocol [19] based on public-key cryptography. We assume that each principal knows the public key of the rest of principals involved in the protocol. We use this protocol along the paper as a running example due to its wide use in the literature, and also to its simplicity.

- | |
|---|
| <ol style="list-style-type: none"> 1. $A \rightarrow B : \{A, N_A\}K_{BP}$ 2. $B \rightarrow A : \{N_A, N_B\}K_{AP}$ 3. $A \rightarrow B : \{N_B\}K_{BP}$ |
|---|

Figure 1: Needham-Schroeder public key authentication protocol.

The protocol involves two principals playing the roles of *initiator* and *responder* (A and B). It consists of three messages:

First Step. A sends to B a message encrypted with B 's public key (K_{BP}) containing (1) her name, and (2) the nonce N_A generated for the transaction. Once received, B decrypts the message by using his private key K_{BS} , thus B *knows* the contents of the message.

Second Step. B sends (1) N_A back to A , and (2) a newly generated nonce (N_B) encrypted with K_{AP} . When A receives the message, she decrypts it by using her secret key K_{AS} . A then detects that N_A is a part of the received message, thus she assumes that the message has been sent by B in response to the previous message.

Third Step. A returns N_B to B encrypted with the B 's public key K_{BP} to confirm that she is really A . Once received, B decrypts the message with his secret key.

The aim of the protocol is to provide mutual entity authentication between two principals by using the exchanged nonces (N_A and N_B) as shared secrets for key establishment. This means that, when A gets the message $\{N_A, N_B\}K_{AP}$, she assumes that, since she had previously sent N_A to B , then only B can know N_A , thus only B can have sent back to her N_A . The same occurs when B receives the message $\{N_B\}K_{BP}$. B knows that only A can know N_B , and so A must have sent such message. At the end of the protocol, both A and B are convinced about the identity of the other participant in the protocol.

3 Introduction to the tccp Language

The Timed Concurrent Constraint Language (tccp) is a declarative language defined in [8] as an extension of the Concurrent Constraint Programming language ccp [22]. In the ccp paradigm, the notion of *store as valuation* is replaced by the notion of *store as constraint*. The computational model is based on a global store where constraints are accumulated, and on a set of agents that interact with the store. The model is parametric w.r.t. a cylindrical constraint system \mathcal{C} defined as follows.

Definition 1 (Constraint System[8]) *Let $\langle \mathcal{C}, \leq, \sqcup, true, false \rangle$ be a complete algebraic lattice where \sqcup is the lub operation, and true and false are the least and the greatest elements of \mathcal{C} , respectively. Assume that Var is a denumerable set of variables, and for each $x \in Var$, there exists a function $\exists_x : \mathcal{C} \rightarrow \mathcal{C}$ such that, for each $u, v \in \mathcal{C}$:*

1. $\exists_x u \leq u$
2. $u \leq v$ then $\exists_x u \leq \exists_x v$
3. $\exists_x(u \sqcup \exists_x v) = \exists_x u \sqcup \exists_x v$
4. $\exists_x(\exists_y u) = \exists_y(\exists_x u)$

Then, $\langle \mathcal{C}, \leq, \sqcup, true, false, Var, \exists \rangle$ is a cylindrical constraint system.

Like in [3], we use the entailment relation \vdash in place of its inverse relation \leq . Formally, given $u, v \in \mathcal{C}$, $u \leq v \iff v \vdash u$, see [22, 8] for more details on the constraint system.

Intuitively, the execution of a tccp program evolves by asking and telling information to

the store. Let us briefly recall the syntax of the language:

$$A ::= \text{stop} \mid \text{tell}(c) \mid \sum_{i=1}^n \text{ask}(c_i) \rightarrow A_i \mid \\ \text{now } c \text{ then } A \text{ else } A \mid A \parallel A \mid \exists x A \mid \rho(x)$$

where c, c_i are *finite constraints* (i.e., atomic propositions) of \mathcal{C} . A *tccp process* P is an object of the form $D.A$, where D is a set of procedure declarations of the form $\rho(x) :- A$, and A is an agent. The stop agent finishes the execution of the program; $\text{tell}(c)$ adds the constraint c to the store, however c will be available only in the subsequent time instant. The agent $\sum_{i=1}^n \text{ask}(c_i) \rightarrow A_i$ consults the store and non-deterministically executes A_i in the following time instant, provided the store satisfies the condition c_i ; otherwise, if no condition c_i is entailed by the store, the choice agent *suspends*. When executing the conditional agent ($\text{now } c \text{ then } A1 \text{ else } A2$), if the store satisfies c , then the agent $A1$ is executed; otherwise $A2$. $A1 \parallel A2$ executes the two agents $A1$ and $A2$ in parallel. The $\exists x A$ agent is used to make variables local to some process. Finally, $\rho(x)$ is the procedure call agent.

In tccp, thanks to the conditional agent ($\text{now } c \text{ then } A \text{ else } A$), it is possible to model behaviors, typical from reactive systems, where the absence of information can cause the execution of a specific action. The notion of time is introduced by defining a global clock which synchronizes all agents. In the semantics of tccp, the only agents that consume time are the *tell*, *choice* and *procedure call* agents. The store grows monotonically, thus it is not possible to change the value of a given variable. If we want to model the evolution of variable values along the time, we have to deal with *streams*. A *stream* allows us to handle imperative variables in the same way as logical lists are used in concurrent logic languages. We write $X = [Y|Z]$ for denoting a stream X recording the current value Y of the variable X . The stream Z represents the future values of the same variable.

The store can be viewed as a blackboard where information is continuously written and never canceled. This definition presents the problem that the order in which information

is added to the store is lost. Therefore, we cannot recover the information added to the store at a given time instant. In [3], it was proposed a new computational model in which a new notion of store (*structured store*) was defined. This new computational model allows us to specify protocols in a simpler way.

A structured store consists of a timed sequence of stores where each store only contains the information added at a given time instant.

Definition 2 (Structured Store [3])

A structured store is an infinite indexed sequence of stores where the i^{th} component of the sequence st is denoted as st_i , and it represents the store at time i .

The current value of a *stream* is the last value added to the stream, formalized in the following way:

Definition 3 (Current value [3]) Given a stream S , a structured store st , and $t \in \mathbb{N}$. Then, A is the value of S in st at instant t , denoted by $st \models_t S = [A|As]$ or $S \doteq_t A$, iff $\exists m > 0$ such that:

$$st \models_t \exists A_1 \dots \exists A_{m-1} \exists As$$

such that $S = [A_1, \dots, A_{m-1}, A|As]$ and

$$st \not\models_t \exists A' \exists As'. As = [A'|As']$$

Under these conditions, the length of stream S in the store st at time t is m , denoted by the term $len(S, st, t) = m$.

We refer to [3] for details regarding the redefinition of both the notion of entailment relation, and the mechanism for updating the store. Following the notation described in [3], $st \vdash_t c$ is used to check if the information stored in the elements up to the t th position of st entails c . Moreover, $st \sqcup_t \{c\}$ is used to add the constraint c to the t th element of st .

In Figure 2, we show the transition relation \longrightarrow of the operational semantics where $\longrightarrow \in (A \times \text{STORE} \times \mathbb{N})^2$, A is the set of tccp agents, and \mathbb{N} is the domain of natural numbers. In the figure, the symbol $\not\longrightarrow$ is used to indicate that it is not possible a step using relation \longrightarrow , i.e., the agent suspends.

Given a tccp program P , an agent A_0 , and an initial structured store $st^0 = st_0^0 \cdot true^\omega \in$

STORE , where st_0^0 represents the first component of the structured store st^0 , the *timed operational semantics* of P w.r.t. the initial configuration $\langle A_0, st^0 \rangle$, is

$$\mathcal{O}_T(P)[\langle A_0, st^0 \rangle] = \{st = st_0^0 \cdot st_1^1 \cdot \dots \in \text{STORE} \mid \langle A_i, st^i \rangle_i \longrightarrow \langle A_{i+1}, st^{i+1} \rangle_{i+1} \text{ for } i \geq 0\}$$

Thus, for each $st^i \in \text{STORE}$ incrementally built during the execution, the semantics only records its i^{th} component st_i^i , which corresponds to the constraints added at the time instant i . We assume that each trace in $\mathcal{O}_T(P)[\langle A_0, st^0 \rangle]$ is infinite (the last configuration is repeated indefinitely if necessary).

3.1 The new ask-tell agent

When we specify a protocol, an important issue is the generation of nonces. In order to implement the generation of nonces, the last nonce generated could be useful in order to (maybe randomly) generate the following one. Therefore, we would need to recover the last information or value added to the store associated to the list of nonces generated by a principal. In order to ease this kind of definitions, we have defined a new agent, called ask-tell, which instantiates a variable with the current value of a given stream.

The following examples show the behavior of some tccp agents, and motivate the definition of the new agent.

1. The agent $\text{now } X=5 \text{ then } A \text{ else } B$ executes A if the store satisfies $X=5$; Otherwise B is executed. This agent does not instantiate variable X with value 5, simply consults whether the constraint holds.
2. The agent $\text{ask}(X=5) \rightarrow A$, similarly to the previous case, executes A (at the following time instant) if the store satisfies $X=5$; Otherwise the agent suspends. Again, this agent does not modify the store.
3. The agent $\text{tell}(X=5)$ adds the constraint $X=5$ to the store. Note that this agent does modify the store.

The syntax of the agent is the following: $\text{ask-tell}(S, A)$, where S is a stream and A is a

R1	$\langle \text{tell}(c), st \rangle_t \longrightarrow \langle \text{stop}, st \sqcup_{t+1} c \rangle_{t+1}$	
R2	$\langle \sum_{i=0}^n \text{ask}(c_i) \rightarrow A_i, st \rangle_t \longrightarrow \langle A_j, st \rangle_{t+1}$	if $0 \leq j \leq n$ and $st \vdash_t c_j$
R3	$\frac{\langle A, st \rangle_t \longrightarrow \langle A', st' \rangle_{t+1}}{\langle \text{now } c \text{ then } A \text{ else } B, st \rangle_t \longrightarrow \langle A', st' \rangle_{t+1}}$	if $st \vdash_t c$
R4	$\frac{\langle B, st \rangle_t \longrightarrow \langle B', st' \rangle_{t+1}}{\langle \text{now } c \text{ then } A \text{ else } B, st \rangle_t \longrightarrow \langle B', st' \rangle_{t+1}}$	if $st \not\vdash_t c$
R5	$\frac{\langle A, st \rangle_t \not\longrightarrow}{\langle \text{now } c \text{ then } A \text{ else } B, st \rangle_t \longrightarrow \langle A, st \rangle_{t+1}}$	if $st \vdash_t c$
R6	$\frac{\langle B, st \rangle_t \not\longrightarrow}{\langle \text{now } c \text{ then } A \text{ else } B, st \rangle_t \longrightarrow \langle B, st \rangle_{t+1}}$	if $st \not\vdash_t c$
R7	$\frac{\langle A, st \rangle_t \longrightarrow \langle A', st' \rangle_{t+1} \text{ and } \langle B, st \rangle_t \longrightarrow \langle B', st'' \rangle_{t+1}}{\langle A \parallel B, st \rangle_t \longrightarrow \langle A' \parallel B', st' \sqcup st'' \rangle_{t+1}}$	
R8	$\frac{\langle A, st \rangle_t \longrightarrow \langle A', st' \rangle_{t+1} \text{ and } \langle B, st \rangle_t \not\longrightarrow}{\langle A \parallel B, st \rangle_t \longrightarrow \langle A' \parallel B, st' \rangle_{t+1}}$	
R9	$\frac{\langle A, st_1 \sqcup \exists x st_2 \rangle_t \longrightarrow \langle A', st' \rangle_{t+1}}{\langle \exists^{st_1} x A, st_2 \rangle_t \longrightarrow \langle \exists^{st'} x A', st_2 \sqcup \exists x st' \rangle_{t+1}}$	
R10	$\langle p(x), st \rangle_t \longrightarrow \langle A, st \rangle_{t+1}$	if $p(x) : -A \in D$

Figure 2: Augmented operational semantics of the language

R11	$\langle \text{ask-tell}(S, A), st \rangle_t \longrightarrow \langle \text{stop}, st \sqcup_{t+1} A = Q \rangle_{t+1}$	if $S \doteq Q$, $\text{free}(A)$, and $\text{len}(S, st, t) > 0$
------------	--	--

Figure 3: Operational semantics for the ask-tell agent

free variable. The agent recovers in A the current value of S , thus instantiating A to such value. The new information (the value of A) will be available to other agents at the following time instant. In Figure 3, we show the operational semantics for the new agent where st is a structured store.

4 Protocol Specification using tccp

In this section, we present the notation that we use to specify the Needham-Schroeder public key authentication protocol in tccp. The system consists of a set of procedure declarations modeling principals, and a controller which models the interaction among principals. The controller also models the intruder, which is who decides whether a message reaches its target or not, whether messages are modified, etc. We assume perfect cryptographic primitives.

In particular, one principal can decrypt a message only if it has the appropriate secret key.

4.1 Facts and procedure declarations

In the following we, show the set of facts and procedure declarations used to represent the data regarding principals, the information known by each principal, and the interaction in the protocol.

a(X). The term $\mathbf{a}/1$ identifies the principal X as a participant of the protocol.

kp(X). The term $\mathbf{kp}/1$ identifies the public key of principal X .

ks(X). The term $\mathbf{ks}/1$ identifies the secret key of principal X .

sk(X, Y). The term $\mathbf{sk}/2$ specifies a key which is shared by two principals: X and Y . Only these two principals know the shared key.

enc(K,MsgItem). The term **enc**/2 represents a list of data elements **MsgItem** encrypted with the key **K**. **K** can be a public key **kp**(X), or a shared key **sk**(X,Y).

msg(MsgCnt). The term **msg**/1 represents a message containing the list of items **MsgCnt** (each item being a nonce, an encrypted message, a principal name, etc.). Each item may be encrypted with a different key, or not encrypted at all.

nonce(X,NonceSt). The term **nonce**/2 stores in the stream **NonceSt** the nonces generated by principal **X**.

know(X,Info). The term **know**/2 stores the fact that principal **X** knows **Info**. **Info** can be instantiated to terms such as keys, names of principals, messages, etc.

deliver(R,X,Y,M,P). The term **deliver**/5, when present in the store, represents the fact that principal **R** has sent the message **M** to **Y**, (maybe) impersonating **X**. Moreover, if the variable **P** is instantiated to **ok**, then the message has already been processed (traveled the network); Otherwise, it must still be delivered.

Let us show some basic actions that are commonly part of protocols. An important action that is commonly part of protocol specifications is the generation of nonces. The generation process implemented in this paper uses the value of the last generated nonce to calculate the new nonce.

```
generator(X,N) :-
  ∃ S,St(tell(nonce(X,S)) ||
    ask(true) → tell(S=[_|St]) ||
    ask(true) →
      ask-tell(S,Q) ||
      ask(true) →
        tell(N is Q+1) ||
        ∃ Aux(tell(St=[N|Aux]))).
```

The predicate **generator**/2 generates and stores a new nonce **N**. The first **tell** agent instantiates **S** to the stream containing the nonces generated by **X**. Then, the **ask-tell** agent instantiates **Q** to the current value of such stream. Since the information is only available in the following time instant, we force the

execution to let pass one time instant by executing an **ask**(**true**) agent. At that point, the new nonce **N** can be generated and the stream of nonces of **X** is consistently updated.

The predicate **dec**/2 models the process of a principal decrypting a message. The message consists of a list of elements that are recursively processed. These elements may be encrypted, thus the principal **X** will know the contents of the message only if he knows the appropriate encryption key.

```
dec(X,Msg) :- ∃ E,T(
  tell(Msg=[E|T]) ||
  ask(true) →
    now(E=enc(_,_)) then ∃ K,L(
      tell(E=enc(K,L)) ||
      ask(true) →
        now((K=kp(X) ∧ know(X,[ks(X)])) ∨
          K=sk(X,_)) ∨ K=sk(_,X))
        then tell(know(X,L))
        else tell(know(X,E))
      else tell(know(X,E)) ||
      now(T≠[]) then dec(X,T) else stop).
```

Finally, the predicate **send**/3, models principal **S** sending a message **Msg** to principal **R**:

```
send(S,R,Msg) :- now(a(S) ∧ a(R)) then
  ∃ Proc tell(deliVer(S,S,R,Msg,Proc))
  else stop.
```

The condition for the action to be taken is that both, **S** and **R**, participate in the protocol. The process tells to the store that the message **Msg** must be delivered to principal **R**. The environment would decide whether the message is actually delivered or not.

4.2 The Participants

Once the basic terms and the basic actions have been modeled, let us show how the two participants in the protocol are modeled. Whereas the above described actions can be reused for modeling other protocols, the code for principals should be redefined for each case.

In Figure 4, we show the behavior of the protocol initiator in the Needham-Schroeder example, i.e., the **A** principal in Figure 1. First of all, the initiator stores that she is a participant

```

initiator(X,Y) :- tell(a(X)) ||
  ∃ NA(now (know(X, [kp(Y)]))) then
    (generator(X,NA) ||
      ask(inst(NA) → send(X,Y, msg([enc(kp(Y), [X,NA])))) ||
      ask(know(X, [NA, _])) →
        ∃ I1(tell(know(X, [NA, I1])) ||
          ask(true) → send(X,Y, msg([enc(kp(Y), [I1])))))
    else stop).

```

Figure 4: Procedure modeling the initiator participant

of the protocol. Then, she checks whether she knows the public key of Y . In that case, sends to Y a message with her name and a new generated nonce.¹ In parallel, she waits until knowing a message containing the generated nonce. The process won't proceed until knowing (receiving) such message. Once received, the initiator sends the confirmation message to Y .

Figure 5 models the responder principal (principal B in Figure 1). The responder waits until knowing (receiving) the initial message of the protocol. Once received, he generates a new nonce and sends the corresponding message to the principal Y . Finally, he waits for the last message of the protocol.

Up to this point, we have modeled the basic actions that are commonly performed during a protocol execution, and the principals participating in the Needham-Schroeder protocol. In the following section, we show how to model the Dolev-Yao intruder.

5 Modeling the Intruder

The design of protocols turns out to be problematic even assuming perfect cryptography. The problem is mainly due to the fact that principals communicate over a network controlled by an intruder who can intercept, analyze, and modify messages, being thus able to carry out malevolent actions. These capabilities correspond to the Dolev-Yao attacker [13]. A man-in-the-middle attack (assuming the Dolev-Yao model) was detected for the Needham-Schroeder public key authentication

¹We assume the constraint system can check whether a variable is non-free: $\text{inst}(N_A)$.

protocol. We show the attack in Figure 6 in order to illustrate which kind of actions an intruder is able to do. I is, in principle, another participant of the protocol, just as A or B , thus any principal could initiate a protocol run to communicate with him.

```

1.  $A \longrightarrow I : \{A, N_A\}K_{IP}$ 
2.  $I_A \longrightarrow B : \{A, N_A\}K_{BP}$ 
3.  $B \longrightarrow I_A : \{N_A, N_B\}K_{AP}$ 
4.  $I \longrightarrow A : \{N_A, N_B\}K_{AP}$ 
5.  $A \longrightarrow I : \{N_B\}K_{IP}$ 
6.  $I_A \longrightarrow B : \{N_B\}K_{BP}$ 

```

Figure 6: Lowe's attack on Needham-Schroeder public key authentication protocol

In the attack, A initiates a protocol run with I , who (impersonating A) starts a second protocol run with B . At the end of the attack, B thinks he is communicating to A , which is false. We use the notation I_X to denote that the intruder I is playing the role of the principal X , and I for denoting the intruder acting as himself. The intruder, as any other principal, has a key pair K_{IP} and K_{IS} corresponding to his public and secret key, respectively.

The steps of the attack are described below:

1. A initiates a protocol run to communicate with the principal I .
2. I_A initiates a second protocol run to communicate with B playing the role of A . The intruder I_A sends to B the nonce N_A just received from A .
3. B assumes that the message has been sent by A , thus it responds to I_A by sending the nonces N_B and N_A encrypted with

```

responder(X,Y) :- tell(a(X)) ||
  ∃ NB(now(know(X, [kp(Y)]))) then
    ask(know(X, [Y, _])) →
      ∃ R1(tell(know(X, [Y, R1]))) ||
        generator(X, NB) ||
        ask(inst(NB)) → send(X, Y, msg([enc(kp(Y), [R1, NB])))) ||
    ask(know(X, [NB])) → stop
  else stop).

```

Figure 5: Procedure modeling the responder participant

A's public key. Note that the intruder cannot decrypt the received message.

4. *I* sends to *A* the message just received from *B*.
5. *A*, once received the message from *I*, responds by sending the nonce just received. Note that *A* assumes she is communicating to *I*, thus she encrypts the message by using the public key of *I* making *I* able to know the nonce N_B .
6. *I*, playing the role of *A*, completes the protocol by sending the nonce N_B to *B*, thus *B* believes that *A* has established a communication session with him.

As we have said before, we model the intruder by implicitly implementing it in the environment. This means that the environment decides whether a message reaches its destination, whether it is modified, or even whether protocol runs are mixed. In Figure 7, we partially show the intruder. When the environment processes a message, the message can non-deterministically be forwarded to the destination principal (modeling the correct behavior), be canceled (ignored), some principal can impersonate another one changing the sender information in the `deliver` term, etc.

Let us describe the actions the environment can do. We have labeled the code in Figure 7 to make clearer the description. First of all, it must be said that the environment is defined as a cycle where, during each iteration, one of the specified actions labeled 1 to 11 is non-deterministically executed. The actions 2 to 11 correspond to the environment starting a

protocol run. Note how honest participants can interact with dishonest ones.

The first possible choice (labeled 1) models the case when someone has sent a message that must be delivered. The environment detects that there is a message to process, so non-deterministically decides what to do with such message. First of all, the environment could do nothing (1a) modeling the case when the message is lost. 1b models the case when the message is correctly delivered, thus the destination participant decrypts the contents of the message. The third option represents the case when the intruder impersonates the first participant, whereas in the last shown option, the intruder impersonates the second participant.

Next we show the system initialization:

```

run :- tell(know(alice, [kp(alice)])) ||
  tell(know(alice, [kp(bob)])) ||
  tell(know(alice, [kp(intruder)])) ||
  tell(know(bob, [kp(alice)])) ||
  [...]
  tell(know(alice, [ks(alice)])) ||
  [...]
  environment(alice, bob, intruder).

```

6 Conclusions

We have shown how `tccp` is a suitable specification language for communication protocols. The language was defined as a simple model to specify reactive and embedded systems. Features such as concurrency or non-determinism natural in `tccp` marry perfectly these systems. In this paper, we have seen that these characteristics are also useful when specifying communication protocols. We have

```

environment(X,Y,Z) :-  $\exists N,M,Ms,Proc,Proc',R($ 
1   (ask(deliver(_,_,-,-,-))  $\rightarrow$ 
    tell(deliver(R,X,Y,M,Proc)) ||
    tell(M=msg(Ms)) ||
    ask(true)  $\rightarrow$ 
    now(Proc/=ok) then
1a   ask(true)  $\rightarrow$  stop || tell(Proc = ok) +
1b   ask(true)  $\rightarrow$  dec(Y,Ms) || tell(Proc = ok) +
1c   ask(a(Z)  $\wedge$  Z $\neq$ Y  $\wedge$  Z $\neq$ X)  $\rightarrow$ 
    tell(a(Z)) || dec(Z,Ms) ||
    ask(true)  $\rightarrow$  now(know(Z,[X,N])) then
        tell(deliver(Z,X,Y,msg(kp(Y),[X,N])),Proc') ||
        tell(Proc = ok)
        else stop +
1d   ask(a(Z)  $\wedge$  Z $\neq$ Y  $\wedge$  Z $\neq$ X)  $\rightarrow$ 
    tell(a(Z)) || dec(Z,Ms) ||
    ask(true)  $\rightarrow$  now(know(Z,[Y,N])) then
        tell(deliver(Z,Y,X,msg(kp(X),[Y,N])),Proc') ||
        tell(Proc = ok)
        else stop +
    [...]
    else stop
+
2,3  ask(true)  $\rightarrow$  initiator(X,Y) + ask(true)  $\rightarrow$  responder(Y,X) +
    [...]
10,11 ask(true)  $\rightarrow$  initiator(Y,Z) + ask(true)  $\rightarrow$  responder(Z,Y) ||
environment(X,Y,Z).

```

Figure 7: Procedure modeling the attacks that an intruder carries out in the hostile environment

defined a compact translation from the informal specification of protocols to the formal one (in *tccp*). To improve the compactness and clarity of models, we have defined a new agent for *tccp*. Many of the components in the defined model can be reused for the specification of other protocols. In this paper we have chosen the Needham-Schroeder protocol due to its simplicity and wide use in the literature, but we can also model other protocols such as the Otway-Rees protocol [20].

The design of communication protocols sometimes seems an easy, even trivial task: usually protocols are composed by few simple steps. However, nowadays everyone knows that, mainly due to the (hostile and powerful) nature of the environment in which a protocol is executed, formal methods are essential to guarantee the correctness and well behavior of these systems. By specifying protocols in *tccp*, we can use its model checker to ver-

ify, not only LTL properties, but also real-time temporal properties.

We plan to extend this work in several ways. First of all, we plan to compare our approach to others previously defined, comparing not only the clarity and compactness in the protocol specification, but also the kind of properties able to check. We also plan to move to a more specialized verification method to optimize the search algorithm, thus the verification time cost. Finally, we plan to study the impact of the nonce generation method, first by improving its implementation (maybe using the functional extension of *tccp* in [4]), and then modeling the capability of guessing nonces by the intruder (thus removing the assumption of secure nonces).

References

- [1] M. Abadi. Secrecy by typing in security protocols. In *Proc. Theoretical Aspects of*

- Computer Software*, pp. 611–638, 1997.
- [2] M. Abadi and A.D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. In *ACM Conf. of Computer and Comm. Security*, pp. 36–47. ACM, 1997.
- [3] M. Alpuente, M. M. Gallardo, Ernesto Pimentel, and A. Villanueva. Verifying Real-Time Properties of tcp Programs. *JUCS*, 12(11):1551–1573, 2006.
- [4] M. Alpuente, B. Gramlich, and A. Villanueva. A Framework for Timed Concurrent Constraint Programming with External Functions. *ENTCS*, to appear, 2007.
- [5] Gérard Berry et al. ESTEREL: a formal method applied to avionic software development. *Science of Computer Programming*, 36(1):5–25, 2000.
- [6] B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *Proc. 14th IEEE Computer Security Foundations Workshop*, 2001.
- [7] C. Bodei. *Security Issues in Process Calculi*. PhD thesis, Università di Pisa, 2000.
- [8] F. S. de Boer, M. Gabbriellini, and M. C. Meo. A Timed Concurrent Constraint Language. *Information and Computation*, 161(1):45–83, 2000.
- [9] D. Bolignano. An Approach to the Formal Verification of Cryptographic Protocols. In *3rd ACM Conf. on Computer and Comm. Security*, pp. 106–118, 1996.
- [10] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, 1990.
- [11] J. A. Clark and J. L. Jacob. A survey of authentication protocol literature. TR, Defense Evaluation Research Agency, 1997.
- [12] G. Denker, J. Meseguer, and C. Talcott. Protocol Specification and Analysis in Maude. In *In Proc. Workshop on Formal Methods & Security Protocols*, 1998.
- [13] D. Dolev and A. C. Yao. On the Security of Public Key Protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, March 1983.
- [14] J. Goubault-Larrecq. A Method for Automatic Cryptographic Protocol Verification. In *Proc. of Workshops of 15th Int'l Parallel & Distributed Processing Symp.*, vol. 1800 of *LNCS*, pp. 977–984, 2000.
- [15] R. Kemmerer. Analyzing encryption protocols using formal verification techniques. vol. 7, pp. 448–457. ACM, 1989.
- [16] G. Lowe. Breaking and Fixing the Needham-Schroeder public-key protocol using FDR. In *Proc. TACAS*, pp. 147–166. Springer, 1996.
- [17] J.C. Mitchell, M. Mitchell, and U. Stern. Automated Analysis of Cryptographic Protocols Using Mur ϕ . In *Proc. of Conf. on Security and Privacy*, pp. 141–153. IEEE Press, 1997.
- [18] D. Monniaux. Abstracting Cryptographic Protocols with Tree Automata. In *Proc. SAS '99*, pp. 149–163, Springer, 1999.
- [19] R. Needham and M. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Communications of the ACM*, 21(12):993–999, 1978.
- [20] D. Otway and O. Rees. Efficient and timely mutual authentication. *Operating Systems Review*, 21(1):8–10, 1987.
- [21] L. C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *Journal of Computer Security*, 6:85–128, 1998.
- [22] V. A. Saraswat. *Concurrent Constraint Programming Languages*. The MIT Press, Cambridge, MA, 1993.
- [23] R. K. Shyamasundar. Analyzing Cryptographic Protocols in a Reactive Framework. In *Proc. VMCAI*, pp. 46–64, 2002.
- [24] C. Weidenbach. Towards an automatic analysis of security protocols in first-order logic. In *Proc. CADE-16*, vol. 1632 of *LNAI*, pp. 378–382. Springer, 1999.