Using tccp for the Specification and Verification of Communication Protocols [1]

Alexei Lescaylle[a,2] Alicia Villanueva[a,3]

[a] *Dept. de Sistemas Informáticos y Computación*
Technical University of Valencia
Valencia, Spain

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. Some declarative approaches based on CLP, Haskell or Maude have been proposed for the specification and analysis of communication protocols. In this paper, we propose the Timed Concurrent Constraint Language (tccp in short) as the specification language for protocols. tccp is a declarative concurrent programming language which allows one to intuitively model concurrent and reactive systems. Cryptographic protocols can be specified in a very compact way thanks to certain features of the language such as the non-deterministic behavior and concurrency. We show how to specify the basic actions that are usually performed during a protocol run. Thereafter, we show how to specify the participants in the protocol. Finally, we specify the intruder model by means of a hostile environment in which principals run the protocol. The model considered for the intruder is the popular model of Dolev-Yao. We use along the paper the Needham-Schroeder public key authentication protocol to illustrate the approach. The basic actions and the intruder implementation can be reused for the specification of many different protocols.

Keywords: Communication Protocols, Timed Concurrent Constraint Language, Automatic Verification.

1 Introduction

Cryptographic (communication) protocols are protocols that use cryptography primitives to authenticate principals over a network. The network is usually assumed to be a hostile environment controlled by an intruder who can read, modify and delete messages sent by *honest* principals. Session establishment protocols aim to establish a secure connection among two or more principals in such a way that each principal is sure to be communicating to the principal he intends to. Dishonest participants are able to modify the protocol behavior, for example making a principal believe to be communicating with a principal when actually is communicating with a different one. The huge cost of designing, analyzing or verifying protocols does not depend

[1] This work has been partially supported by the EU (FEDER) and the Spanish MEC under grants TIN2004-7943-C04, HA2006-0007, and Valencian Government under Grant GV06/285

[2] Email: alescaylle@dsic.upv.es

[3] Email: villanue@dsic.upv.es

only on the protocol specification, but also on the environment in which the protocol is run. Formal methods have been shown to be effective tools when specifying, analyzing and verifying protocols thanks to their capability to detect *non intuitive* attacks, impossible to find otherwise. In that context, declarative techniques have also come up as a very convenient approach.

Logic programming languages have been applied to the problem of specification and analysis of communication protocols in different ways. For example, [19] was a pioneer work, where a reachability analysis was implemented in Prolog. In [1], a logic programming language was used to specify in a declarative way knowledge, protocol rules and intruder capabilities. Prolog was again used in [7] for the specification of protocol abstractions for proving the absence of attacks. Other declarative languages have also been shown suitable for the protocol specification and analysis problems. In [15], the specification language Maude was shown as an alternative approach. Later, in [11], an approach based on multiset-rewriting was presented. Another work based on multiset-rewriting was presented and implemented in OCAML [12,13]. Haskell has also been considered for the specification of communication protocols: in [5] we find a comparison (both qualitative and comparative analysis is done) between the specification of protocols in Maude and in Haskell.

The above mentioned literature shows how a formal concurrent language allows the designer to write unambiguous, clear, and concise specification. In this paper, we propose the Timed Concurrent Constraint Language (tccp in short, [9]) as the specification language. Its declarative and concurrent nature, together to some additional features such as the non-determinism, makes this language suitable for our purpose. On the one hand, the non-determinism behavior of tccp allows one to obtain compact specifications for systems. On the other hand, the agent-based model provides an intuitive way to specify principals. Finally, the notion of time allows us to intuitively model the interaction among principals, and to use the LTL logic to check temporal properties over protocol specifications.

Synchronous languages such as ESTEREL [6] or LUSTRE [18] have also been shown to be suitable for the specification and analysis of protocols. For example, in [23] the language ESTEREL is used to analyze the TMN protocol [24]. There, principals of the protocol (initiator, responder, server and intruder) are specified, and some security properties are verified.

We propose the tccp language for the specification of communication protocols due to its nature. tccp shares some concurrency features with synchronous languages. Moreover, it shares the formal nature and non-determinism with logic-based approaches. We intend to take advantage of both views, and use the formal verification tools defined for tccp ([2,3,17]) in order to verify protocols' properties.

The main contributions of this work are discussed in the following. We first show how we can specify the basic actions that principals can perform. Afterwards, we show how to model the participants of a protocol. We illustrate our proposal by using the Needham-Schroeder public key authentication protocol [20]. We use a general notation that can be used to specify other protocols, for example protocols in which messages can be partially encrypted, etc. Regarding the intruder, let us mention that we do not model the intruder in an explicit way, but we follow the approach of specifying a (hostile) environment which controls the messages

flaw. In other words, the Dolev-Yao intruder [16] is implicitly specified by a non-deterministic choice among (correct or incorrect) *actions* of the environment, such as losing a message, modifying a message, or even modifying the destination principal of a message. The specification of the basic actions as well as the specification of the intruder are defined in such a way that they can be reused for the specification of different protocols.

The remainder of this paper is organized as follow. In Section 2 we show an introduction to cryptographic protocols, and in particular to the Needham-Schroeder public key authentication protocol that will be our running example in the rest of the paper. An overview of `tccp` and an extension which makes specifications clearer is given in Section 3. Sections 4 and 5 present the method to model principals and the Dolev-Yao intruder. Section 6 is devoted to the verification of the protocol and discusses some related work. Finally, Section 7 concludes and discusses future work.

2 Cryptographic Protocols

A protocol can be seen as a recipe that enables the connection and data exchange between two or more entities through messages. We usually call these entities *principals*. In its simplest form, a protocol can be defined as the rules governing the syntax, semantics and synchronization of communication. Protocols are executed to achieve some specific goal. For instance, a typical goal is to establish a *secure* channel that principals would use later for exchanging confidential data. The principals can be users, hosts or processes [14] usually playing the role of initiator, responder or server. A protocol describes how messages must be sent from one principal to another, and how these principals react when receiving a message. Messages consist of atoms (principals' names, nonces, etc.) and, to guarantee confidentiality, they may be encrypted with a given key that only the target principal can decrypt. A nonce is a value (ideally) randomly generated by a principal which is usually sent to other principals in order to ensure the freshness of messages. As occurs typically in the literature, we assume that nonces cannot be guessed by intruders.

Cryptographic protocols are an important subclass of protocols where (perfect [4]) cryptographic techniques are used. In the following, we illustrate the notation that is normally used to represent a protocol.

- A and B are 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 [20] based on public-key cryptography without using trusted key servers. Some implicit assumptions are commonly made:

- any principal knows the public key of the principals involved in the protocol, and

[4] Perfect cryptography assumes that one cannot decrypt an encrypted message without the decryption key.

- only the owner of a secret key knows his secret key.

The second assumption implies that intruders cannot decrypt messages sent to other principals. Note that assumptions can be the source of many problems in the design and verification of protocols since make more difficult to find subtle problems unless an automatic verification tool were used.

1. $A \rightarrow B : \{A, N_A\}K_{BP}$
2. $B \rightarrow A : \{N_A, N_B\}K_{AP}$
3. $A \rightarrow B : \{N_B\}K_{BP}$

Fig. 1. Needham-Schroeder public key authentication protocol

As we can observe, this protocol establishes the order and contents of messages between two principals, A and B, that play the role of *initiator* and *responder*, respectively. Once the protocol has been completed, both principals are convinced about the identity of their partners. From that time instant, they can share critical information. In other words, the protocol ensures A to be communicating with B, and vice-versa. Let us describe each of the three steps of the protocol:

(i) A sends to B a message encrypted with B's public key (K_{BP}). The message contains A's name, and a fresh nonce generated by A (N_A). B decrypts the message by using his private key K_{BS}, thus B *knows* the content of the message.

(ii) B sends to A, encrypted with the public key of A, a message which contains the received nonce N_A, and a newly generated nonce N_B. A decrypts the message by using her secret key K_{AS}. Since the message contains the nonce N_A, A can deduce that the message has recently been sent by B.

(iii) A sends the nonce N_B to B, encrypted with B's public key K_{BP}. Then, B can infer that he is communicating to A since he had previously sent to A such nonce.

We use an asynchronous model for the specification of messages exchange. This means that each step in the protocol is modeled as the execution of two asynchronous actions: a principal sending a message, and a second principal receiving the message.

3 Introduction to the tccp Language

The Timed Concurrent Constraint Language (tccp) is a concurrent declarative language defined in [9] as an extension of the Concurrent Constraint Programming paradigm (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. Intuitively, the execution of a tccp program evolves by asking and telling information to the store. It is a simple but powerful model in which partial information can easily be handled. The model is parametric w.r.t. a cylindric constraint system $\mathcal{C}$ defined as follows:

Definition 3.1 [Constraint System [9]] Let $\langle \mathcal{C}, \leq, \sqcup, true, false \rangle$ be a complete algebraic lattice where $\sqcup$ is the lub operation, and *true*, *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, $\langle \mathcal{C}, \leq, \sqcup, \mathit{true}, \mathit{false}, \mathit{Var}, \exists \rangle$ is a *cylindric constraint system.*

Like in [3], we use the entailment relation $\vdash$ instead of its inverse relation $\leq$. Formally, given $u, v \in \mathcal{C}, u \leq v \iff v \vdash u$. The interested reader can find in [9,22] more details about the constraint system.

In tccp, it is possible to model behaviors typical from reactive or embedded systems. In these systems, the absence of information can cause the execution of an action. tccp introduces the conditional agent (now c then A else A) in order to model such behaviors. Let us briefly recall the syntax of the language:

$$A ::= \mathsf{stop} \mid \mathsf{tell}(c) \mid \textstyle\sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i \mid \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, A \mid A || A \mid \exists x\, A \mid \mathsf{p}(\overline{x})$$

where c, c_i are *finite constraints* (i.e., atomic propositions) of $\mathcal{C}$. A tccp program P is an object of the form $D.A$, where D is a set of procedure declarations of the form $\mathsf{p}(\overline{x})$:$-A$, and A is an agent. Intuitively, the stop agent finishes the execution of the program. $\mathsf{tell}(c)$ adds the constraint c to the store, but c will be available only in the subsequent time instant. The choice agent $\sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i$ checks whether the store satisfies the guards, and non-deterministically executes (in the following time instant) one of the agents A_i, provided its guard c_i were satisfied. In case no condition c_i is entailed, the choice agent *suspends*. The conditional agent (now c then $A1$ else $A2$) executes agent $A1$ if the store satisfies c, otherwise executes $A2$. $A1||A2$ executes the two agents $A1$ and $A2$ in parallel (the concurrent model used is *maximal parallelism*). The $\exists x\ A$ agent is used to define variables local to the process A. Finally, $\mathsf{p}(\overline{x})$ is the procedure call agent.

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 in the original tccp model can be seen as a blackboard where information is continuously written and never canceled. Stores grow monotonically, thus it is not possible to change the value of a given variable. In order to model the evolution of variable values along the time, we use *streams*: A stream allows one to handle imperative-style variables in the same way as logical lists work for concurrent logic languages. We write $X = [Y|Z]$ for denoting a stream X recording (i) the current value Y of the (imperative-style) variable, and (ii) the stream Z of future values of such variable.

The tccp model presents the problem that we cannot retrieve in a simple way the order in which the information has been added to the store, or the information added at a specific time instant. In [3], a new computational model was proposed in which a new notion of store, called *structured store*, was defined. This new

computational model allows us to recover the order in which the information is added during a computation. In this paper, we consider that framework instead of the original one since, as we will show later, it makes simpler some tasks such as the nonce generation.

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

Definition 3.2 [Structured Store [3]] A *structured store* is an infinite indexed sequence of stores where the ith component of the sequence st is denoted as st_i, and it represents the store at time i.

The new model makes possible to retrieve from the store in a simple way the current value of a stream, i.e., the last value added to the stream:

Definition 3.3 [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:

$$st \models_t \exists A_1 \cdots \exists A_{m-1} \exists As \text{ such that } S = [A_1, \cdots, A_{m-1}, A|As] \text{ and } st \not\models_t \exists A' \exists As'.As = [A'|As']$$

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

Following the notation described in [3], $st \vdash_t c$ is used to check if the information stored up to the tth element of st entails c. Moreover, $st \sqcup_t \{c\}$ is used to add the constraint c to the tth 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 a 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 to evolve by using relation $\longrightarrow$, i.e., the execution suspends.

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

Fig. 2. Structured operational semantics.

Given a tccp program P, an agent A_0, and an initial structured store $st^0 = st^0_0 \cdot true^\omega$ where st^0_0 represents the first component of st^0, the *timed operational semantics* of P w.r.t. the initial configuration $\langle A_0, st^0\rangle$ is defined as

$$\mathcal{O}_T(P)[\langle A_0, st^0\rangle] = \{st = st^0_0 \cdot st^1_1 \cdot \ldots \mid \langle A_i, st^i\rangle_i \longrightarrow \langle A_{i+1}, st^{i+1}\rangle_{i+1} \text{ for } i \geq 0\}$$

*3.1 The **ask-tell** agent*

During the protocol specification process, an important issue is the generation of nonces that can be done by following different approaches. For instance, many approaches generate the new nonce by applying a (more or less complicated) function to the last generated nonce. In this paper, nonces are generated by recovering from a stream in the store the last generated nonce. Then, a new value is computed and stored. In tccp, this process must be done in two steps, so we define the ask-tell agent which mechanizes and makes clearer that task.

The new ask-tell agent instantiates a given variable with the current value of a given stream. In order to motivate the definition of this agent, let us first show some examples that illustrate the different behavior of some tccp agents. Let A and B be generic agents of the language:

(i) The agent now $X = 5$ then A else B executes A if the store satisfies the constraint $X = 5$; Otherwise B is executed. This agent **does not** force the binding of variable X to the value 5, i.e., the store does not change.

(ii) The agent ask $(X = 5) \rightarrow A$, which is a non-deterministic choice in which only one possible branch has been defined, executes (at the following time instant) the agent A provided the store satisfies $X = 5$; Otherwise the agent suspends. Again, this agent **does not** modify the store.

(iii) The agent tell$(X = 5)$ adds the constraint $X = 5$ to the store. The added information is available at the following time instant. This agent **may modify** the store by instantiating the variable X to 5.

As one can observe, consults do never add information to the store. They must be combined with tell agents in order to add any kind of information to the store. The ask-tell agent consults the store and updates it by instantiating a fresh variable.

Definition 3.4 Let S be a stream and A a fresh variable (i.e., a non-instantiated variable). The ask-tell(S, A) agent recovers in A the current value of stream S. Moreover, it instantiates A to such value, adding the corresponding information to the store. The new information (the value of A) will be available at the following time instant, similarly to the tell agent.

Figure 3 shows the operational semantic for the new agent where st is a structured store.

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

Fig. 3. Operational semantics for the ask-tell agent

4 Protocol Specification using tccp

This section presents how to encode protocols in tccp. We use the Needham-Schroeder protocol as a running example. The system consists of a set of principals modeled by procedure declarations, and a controller which models the interaction among principals. The controller can be seen as a synchronization process which actually models both, the *honest behavior* of principals, and the *intruder behavior*. We assume perfect cryptographic primitives. In particular, we assume that each principal can encrypt a message with the public key of any principal, and one principal can decrypt a message only if it has the corresponding secret key. Our notation allows one to specify protocols which use symmetric and shared keys instead of pairs of public and secret keys.

4.1 Facts and procedure declarations

In the following, we first show the set of terms that are handled by processes, i.e., terms that agents add/consult to the store. We also show some procedure declarations that implement basic actions executed by principals during a protocol run. For example, the generation of nonces, the decryption of a message, or the process of sending messages. All these definitions can be reused for the specification of protocols different from the Needham-Schroeder protocol.

`a(X)`. The term `a/1` identifies the principal `X` as a participant of the protocol.

`kp(X)`. The term `kp/1` identifies the public key of principal `X`. This information allows us to indicate which key has been used to encrypt a message.

`ks(X)`. The term `ks/1` identifies the secret key of principal `X`. This information allows one to check whether a given principal can decrypt a message.

`sk(X,Y)`. The term `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` (nonces, encrypted messages, principal names, etc.). Note that each element in the list may be encrypted with a different key, or non 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) impersonatig `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 now 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 applied 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)) ||
                        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 the principal `X`. Then, the new `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 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,Y,T) else stop).
```

Finally, the predicate `send/3`, models the action when Principal `S` sends a message `Msg` to principal `R` :

```
send(S,R,Msg) :- now (a(S) ∧ a(R) ∧ knows(S,kp(R))) then
                   ∃ Proc tell(deliver(S,S,R,Msg,Proc)).
```

A 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`. It also stores the fact that `S` is the sender. The environment would decide whether the message is actually delivered or not. The asynchronous model for the communication becomes clear at this point. Note that `send` predicates are invoked by honest principals, thus they always impersonate themselves.

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.

Next we show the behavior of the protocol initiator in the Needham-Schroeder example, i.e., the A principal in Figure 1.

```
initiator(X,Y) :- tell(a(X)) ||
 ∃ N_A (now (know(X,kp(Y))) then
            (generator(X,N_A) ||
             ask(inst(N_A))→send(X,Y,[enc(kp(Y),[X,N_A])]) ||
             ask(know(X,[N_A,_])) →
                     ∃ I_1(tell(know(X,[N_A,I_1])) ||
                          ask(true)→send(X,Y,[enc(kp(Y),[I_1])])))
        else stop).
```

First of all, the initiator stores that she is a 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. [5] 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.

The model for the responder principal (principal B in Figure 1) is defined as:

```
responder(X,Y) :- tell(a(X)) ||
   ∃ N_B(now(know(X,kp(Y))) then
            ask(know(X,[Y,_]))→
               ∃ R_1(tell(know(X,[Y,R_1]))) ||
                    generator(X,N_B) ||
                    ask(inst(N_B))→ send(X,Y,[enc(kp(Y),[R_1,N_B])]) ||
        ask(know(X,[N_B])) → stop).
```

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 the environment is defined to model the Dolev-Yao intruder.

5 Modeling the Intruder

The design of protocols turns out 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 [16]. A man-in-the-middle attack (assuming the Dolev-Yao model) was detected for the Needham-Schroeder public key authentication protocol. We show the attack in Figure 4 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

[5] We assume the constraint system can check whether a variable is non-free: inst(N_A).

A or B, thus any principal could initiate a protocol run to communicate with him.

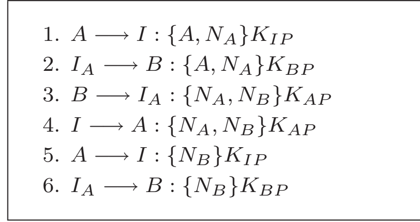

Fig. 4. Lowe's attack on Needham-Schroeder public key authentication protocol

In the attack, A initiates a protocol run with I, who (impersonating A) starts a second run of the protocol with B. In other words, the intruder asks B to initiate a communication session saying that he is A. 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. We assume that the intruder I, as any other principal, has a key pair K_{IP} and K_{IS} corresponding to his public and secret key, respectively.

The sequence of steps in the attack is described below:

(i) A initiates a protocol run to communicate with the principal I.

(ii) I_A initiates a second protocol run to communicate with B playing the role of A. Therefore, the intruder I_A sends to B the nonce N_A just received from A.

(iii) 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 the public key of A. Note that the intruder cannot decrypt the received message since it has been encrypted by using the key of A.

(iv) I sends to A the message just received from B.

(v) A, once received the message from I, responds to it by sending the nonce just received. Note that A assumes is communicating to I, thus the message will be encrypted by using the public key of I. This means that I will be able to recover the nonce N_B.

(vi) I, playing the role of A, completes the protocol by sending the nonce N_B to B, thus B believes that A has correctly established a 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 5 we show the intruder role in `tccp`. Note that, when the environment receives a message, the message can non-deterministically be forwarded to the destination principal (modeling the correct behavior), be canceled (ignored), be sent to a different principal (without modification), or some principal can impersonate another one changing the sender information stored in the `deliver` term.

Let us describe the actions the environment can do. We have labeled the code in Figure 5 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 (actually activating a prin-

cipal playing a specific role in the protocol). Note how the honest participants can interact with dishonest ones as Z in this example.

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 (ignored). $1b$ models the case when the message is correctly delivered, thus the destination participant decrypts the contents of the message. The third and fifth options represent the case when the intruder impersonates the first participant, whereas in the rest of the options, the intruder impersonates the second participant. The environment processes each message only once. The last argument in the `deliver` term controls such restriction.

```
      environment(X,Y,Z) :- ∃N,M,Ms,Proc,Proc',R(
1        (ask (deliver(_,_,_,_,_)) →
              tell(deliver(R,X,Y,M,Proc)) ||
              tell(M=msg(Ms)) ||
              ask(true) →
                 now (Proc/=ok) then
1a                    ask(true) → stop ||
                                 tell(Proc = ok) +
1b                    ask(true) → dec(Y,Ms)
                                 tell(Proc = ok) +
1c                    ask(a(Z) ∧ Z≠Y ∧ Z≠X) →
                         tell(a(Z)) ||
                         dec(Z,Ms) ||
                         now(know(Z,[X,N])) then
                             ask(true)→tell(deliver(Z,X,Y,msg(kp(Y),[X,N])),Proc')) ||
                                        tell(Proc = ok)
                         else stop +
1d                    ask (a(Z) ∧ Z≠Y ∧ Z≠X) →
                         tell(a(Z)) ||
                         dec(Z,Ms) ||
                         now(know(Z,[Y,N])) then
                             ask(true)→tell(deliver(Z,Y,X,msg(kp(X),[Y,N])),Proc')) ||
                                        tell(Proc = ok)
                         else stop +
1e                    ask(a(Z) ∧ Z≠Y ∧ Z≠X) →
                         tell(a(Z)) ||
                         dec(Z,Ms) ||
                         now(know(Z,[N])) then
                             ask(true)→tell(deliver(Z,X,Y,msg(kp(Y),[N])),Proc')) ||
                                        tell(Proc = ok)
                         else stop +
1f                    ask (a(Z) ∧ Z≠Y ∧ Z≠X) →
                         tell(a(Z)) ||
                         dec(Z,Ms) ||
                         now(know(Z,[N])) then
                             ask(true)→tell(deliver(Z,Y,X,msg(kp(X),[N])),Proc')) ||
                                        tell(Proc = ok)
                         else stop +
                 else stop
         +
2,3      ask(true) → initiator(X,Y) + ask(true)→ responder(Y,X) +
4,5      ask(true) → initiator(Z,X) + ask(true)→ responder(X,Z) +
6,7      ask(true) → initiator(Z,Y) + ask(true)→ responder(Y,Z) +
8,9      ask(true) → initiator(X,Z) + ask(true)→ responder(Z,X) +
10,11    ask(true) → initiator(Y,Z) + ask(true)→ responder(Z,Y)) ||
        environment(X,Y,Z).
```

Fig. 5. Procedure modeling the attacks that an intruder carries out in the hostile environment

We have shown here the fragment of the intruder that can affect the protocol run (the form of messages determines the fragment). To conclude, we show the initialization of the system:

```
run :- tell(know(alice,kp(alice))) ∥ tell(know(alice,kp(bob))) ∥
       tell(know(alice,kp(intruder))) ∥ tell(know(bob,kp(alice))) ∥
       tell(know(bob,kp(bob))) ∥ tell(know(bob,kp(intruder))) ∥
       tell(know(intruder,kp(alice))) ∥ tell(know(intruder,kp(bob))) ∥
       tell(know(intruder,kp(intruder))) ∥ tell(know(alice,ks(alice))) ∥
       tell(know(bob,ks(bob))) ∥ tell(know(intruder,ks(intruder))) ∥
       environment(alice,bob,intruder).
```

6 Verification of the specified protocol

In this section we show how it is possible to verify properties for the protocol. Recall that we want to use the model checker defined for the `tccp` language. The most important point, independently from the verification or validation technique used, is to identify which properties we must check, and how can we specify them. Note that depending on the protocol considered, some properties have no sense, whereas others are the ones characterizing the well-behavior of the system.

In our case, an interesting property is whether, nonces are always known by at most two principals. The situation when more than two principals know a given nonce characterizes a protocol run where a man-in-the-middle attack has happened. If we try to check the property by using a model checker and it is not satisfied, then the model checker shows a counterexample, i.e., a system execution where a nonce is known by too many people.

We use the linear temporal logic (LTL) presented in [8] for dealing with `tccp` programs in order to specify the property. Then, we use the `tccp` model checker in [17] to verify the property. In the LTL logic, a formula must be satisfied by all the possible executions in order to be valid. We can use the classical temporal operators *always* $\Box\phi$, *eventually* $\Diamond\phi$, *until* $\phi\,\mathcal{U}\,\psi$, *next* $\bigcirc\phi$, as well as an existential quantificator over variables and classical logic operators.

Let us assume that we can check whether a principal knows a given nonce by using the predicate `know-nce/2` (we can avoid this assumption by expanding the predicate). `know-nce(X,N)` is true whenever the store satisfies the following formula: `know(X,[N])` $\vee$ `know(X,[N,_])` $\vee$ `know(X,[_,N])`. Then, the following temporal formula must be satisfied by the system:

$\neg\Diamond(\exists$ `N(know-nce(alice,N)` $\wedge$ `know-nce(bob,N)` $\wedge$ `know-nce(intruder,N))`$)$

The problem of this protocol comes up when a participant uses a nonce generated by a different participant in order to start a new protocol run. The intruder takes advantage making the responder to construct a message that he later will use in order to ask to the actual owner of the nonce.

Note that to specify a good property, i.e., a property that characterizes the well or bad behavior of a system, the user must know the goal of the protocol. Otherwise he could be proving properties that have no sense in a real environment.

We could also specify more sophisticated properties, for example to make explicit the amount of time that a principal waits for the completion of the protocol by another principal. For example, we could check whether the second confirmation comes up no later than 10 time units from the addition of the first one to the store. This kind of properties are especially interesting in the protocol verification context. A traditional LTL logic is not expressive enough to model such properties, thus we need a real-time logic as the one presented in [3].

In the introduction of this paper, we have already mentioned some related work where other declarative languages and different logics have been proposed for the specification of protocols in order to take advantage of the available analysis tools. Some of these languages share some good feature with tccp, but only tccp joins three of these features. First of all, it is a concurrent language in whose semantics time is modeled. Therefore, the specification of interactive or reactive systems is very intuitive. Moreover, it handles constrains, thus providing the capability to reason about partial information. CLP is a constraint based language that was proposed as an alternative. Constraints provide an important feature: to deal with partial information, and in particular with infinite-state systems. The idea of the CLP-based approaches is to transform the verification problem into a CLP model, and then to check the satisfiability of the generated model. In this paper, we use the constraint system to handle information and specify properties in a more compact way. We do not transform the verification process since we model the system and later we specify the property and use the model checker. Another difference between the two languages is that tccp is an agent-based model with time, which simplifies the specification of reactive systems.

7 Conclusions and Future Work

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 defined a compact and intuitive 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. We have also shown that 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 as a running example due to its simplicity and wide use in the literature but we can also model other protocols such as the Otway-Rees protocol [21].

The design of communication protocols sometimes seems an easy, even trivial task: usually protocols are composed by few simple steps. However, nowadays everyone know 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 verify, 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. We aim to compare, not only the clarity and compactness in the protocol specification, but also the kind of properties we are able to check. We also plan to move to a more specialized verification method in order 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] L.C. Aello and F. Massacci. Verifying security protocols as planning in logic programming. *Transactions on Computational Logic*, 2(4):542–580, 2001.

[2] M. Alpuente, M.M. Gallardo, E. Pimentel, and A. Villanueva. A Semantic Framework for the Abstract Model Checking of tccp Programs. *Theoretical Computer Science*, 346(1):58–95, 2005.

[3] M. Alpuente, M.M. Gallardo, E. Pimentel, and A. Villanueva. Verifying Real-Time Properties of tccp Programs. *Journal of Universal Computer Science*, 12(11):1551–1573, 2006.

[4] M. Alpuente, B. Gramlich, and A. Villanueva. A Framework for Timed Concurrent Constraint Programming with External Functions. *Electronic Notes in Theoretical Computer Science*, to appear, 2007.

[5] D. Basin and G. Denker. Maude versus Haskell: an Experimental Comparison in Security Protocol Analysis. *Electronic Notes in Theoretical Computer Science*, 36, 2001.

[6] G. Berry. *Proof, language, and interaction: essays in honour of Robin Milner*, chapter The foundations of Esterel, pages 425–454. MIT Press, Cambridge, MA, USA, 2000. ISBN 0-262-16188-5.

[7] B. Blanchet. An Efficient Cryptographic Protocol Verified Based on Prolog Rules. In S. Schneider, editor, *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, 2001.

[8] F. de Boer, M. Gabbrielli, and M.C. Meo. A Temporal Logic for reasoning about Timed Concurrent Constraint Programs. In *Proceedings of the 8th International Symposium on Temporal Representation and Reasoning (TIME'01)*, page 227. IEEE Computer Society, 2001.

[9] F. S. de Boer, M. Gabbrielli, and M. C. Meo. A Timed Concurrent Constraint Language. *Information and Computation*, 161(1):45–83, 2000.

[10] M. Bozzano and G. Delzanno. Automated Verification of Secrecy Properties for Linear Logic Specifications of Cryptographic Protocols. *Journal of Symbolic Computation*, 38(5):1375–1415, 2004.

[11] I. Cervesato, N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. A meta-otation for protocol analysis. In *Proceedings of the 12th Computer Security Foundations Workshop*, pages 55–69. IEEE Computer Society Press, 1999.

[12] Y. Chevalier, F. Jacquemard, M. Rusinowitch, M. Turuani, and L. Vigneron. CASRUL web site. `http://www.loria.fr/cassis/protheo/softwares/casrul`.

[13] Y. Chevalier and L. Vigneron. A tool for lazy verification of security protocols. 2001.

[14] J. A. Clark and J. L. Jacob. A survey of authentication protocol literature. Technical report, Defence Evaluation Research Agency, 1997.

[15] G. Denker, J. Meseguer, and C. Talcott. Protocol specification and analysis in Maude. In N. Heintze and J. Wing, editors, *Proceedings of Workshop on Formal Methods and Security Protocols*, 1998.

[16] D. Dolev and A. C. Yao. On the Security of Public Key Protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, March 1983.

[17] M. Falaschi and A. Villanueva. Automatic Verification of Timed Concurrent Constraint Programs. *Theory and Practice of Logic Programming*, 6(3):265–300, 2006.

[18] N. Halbwachs, P. Caspi, R. Raymond, and D Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[19] C. Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

[20] R. Needham and M. Schroeder. Using Encryption for Authentification in Large Networks of Computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[21] D. Otway and O. Rees. Efficient and timely mutual authentication. *Operating Systems Review*, 21(1):8–10, 1987.

[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 *VMCAI*, pages 46–64, 2002.

[24] M. Tatebayashi, N. Matsuzaki, and D. Neuman. Key distribution protocol for digital mobile communication systems. In *Proc. CRYPTO' 89*, number 90, pages 324–333. Springer Verlag, 1989.