Modeling and verifying security protocols using PAT approach

Luu Anh Tuan
School of Computing,
National University of Singapore
tuanluu@comp.nus.edu.sg

***Abstract*—Security protocols play more and more important role nowadays, ranging from banking to electronic commerce systems. They are designed to provide properties such as authentication, key exchanges, key distribution, non-repudiation, proof of origin, integrity, confidentiality and anonymity, for users who wish to exchange messages over a medium over which they have little control. These properties are often difficult to characterize formally (or even informally). The protocols themselves often contain a great deal of combinatorial complexity, making their verification extremely difficult and prone to error. To overcome these obstacles, many different approaches are proposed such as using theorem provers or ranking systems. However, they are lack of automation, sufficiency in demand or time verification. In this paper, we will propose an approach using Real Time System (RTS) and an model checker PAT to deal with these problems.**

***Keywords*-Security protocol; PAT; verification; RTS;**

I. Introduction

With the explosion of the Internet, electronic transactions have become more and more common. The security for these transaction is very crucial to many applications, e.g. electronic commerce, digital contract signing, electronic voting, and so on. However, these large open networks where trusted and untrusted parties coexist and where messages transit through potentially dangerous environment pose new challenges to the designers of communication protocols. Properties such as authenticity, confidentiality, proof of identity, proof of delivery, or receipt are difficult to assure in this scenario.

Security protocols, communications protocols with an essential use of cryptographic primitives, aim at solving this problem [16]. By a suitable use of shared and public key cryptography, random numbers, hash functions, encrypted and plain messages, a security protocol may assure an agent that the invisible responder at the other side of the network really is who he claims to be.

Not surprisingly, security protocols are very difficult to design by hand, as errors may creep in by combining protocols actions in ways not foreseen by the designer [9]. Thus, the formal verification of security protocols became the subject of intense research in recent. For instance, methods based on beliefs logics [9] [24], theorem proving with induction [14][1], and state exploration methods [5] have been successfully used to verify and debug security protocols. However, it became apparent that the formalization process itself was a serious bottleneck in the design process. At first, formalizing a protocol was hardly doable by somebody different from the proposer of the formal method itself. Second, the ambiguity of the goals of the protocol made it possible to find ”attacks” with formal methods that security analysts will never regard as such. Indeed, a security analyst can easily define a ”security violation” in terms of sent, received, and missing messages, while the language gap between the analyst and the formal method makes it difficult to formally and exactly capture what is needed. Moreover, the verification in these methods has a little subtle: some methods can use theorem prover as PVS for semi-automated supporting, but mainly proving by hand, experience and tricky. Therefore, it is very hard to prove for the large-scale protocol.

In this paper, we introduce an approach to automatically verify the security protocols using Real Time System (RTS). By transforming the user specification into RTS model, we use the model checker named PAT to verify the security properties. We also introduce a new security language and a transformation tool to help the user more convenient and easier in specifying the security protocols precisely.

Outline The rest of the paper is organized as follows. Section II discusses related works on others formal models and verifications of security protocols. Section III and IV present some background knowledge about security protocols, *RTS* model and PAT model checker. Section V details the *RTS* models of security protocols. Security goals and results of experiments of verifying are presented in Section VI. SEVE language and translation tool are introduced in Section VII and Section VIII finally concludes the whole paper.

II. Related Works

In recent years, method for analyzing security protocols using the process algebra CSP [4] has been developed [18][7]. An advantage of using process algebra for modeling security protocols is that the model is easily extended. This technique has proved successful, and has been used to discover a number of attacks upon protocols [17][18][20][19].

However; the technique has required producing a CSP description of the protocol by hand; this has proved tedious and error-prone. The verification is also mostly base on manual proofs or theorem proving, not suitable for large system.

ProVerif [2] is an automatic cryptographic protocol verifier, in the formal model (so called Dolev-Yao model). This protocol verifier is based on a representation of the protocol by Horn clauses. It can handle many different cryptographic primitives and an unbounded number of sessions of the protocol and message space. However, it does not support timed language specification and focus mainly on authentication and secrecy only. The NRL protocol analyzer [15] , based on narrowing in rewriting systems, can verify correspondences defined in a rich language of logical formulae. It is sound and complete, but does not always terminate.

Developed originally by Gavin Lowe, the Casper/FDR tool set as described in [7] automatically produces the CSP description from a more abstract description, thus greatly simplifying the modeling and analysis process. The user specifies the protocol using a more abstract notation, similar to the notation appearing in the academic literature, and Casper compiles this into CSP code, suitable for checking using FDR. However, Casper just supply some forms of specification for protocols, mostly focus on authentication and secrecy, not for other security properties such as integrity, fairness, non-repudiation, anonymity ... Casper also support for security protocols involved time, but this time is just the simulation captured by variables, not by the semantic of the language, so it is difficult to describe the essence and semantic of the time in timed protocols.

Our approach is using Real Time System (RTS), an dialect of Timed CSP, to model the security protocols. Therefore, we can specify and verify precisely the time related security model. Moreover, we also support many kind of security properties such as secrecy, agents and data authentication, data integrity, non repudiation and LTL assertion checking. In addition, to ease the difficulty in specify the protocol in RTS , which is very easy to get error and only suitable for those master in this language, we introduce a new security specify language and a transformation tool from this language to RTSwhich is suitable input for PAT model checker.

III. Introduction About Security Protocols

Security protocols describe the messages sent between honest participants during a session. A session is a single run of the protocol. Most protocols allow multiple concurrent sessions. Participant of the session are called agents and are usually denoted A (for Alice) and B (for Bob). A third participant I (for Intruder) represents the adversary who tries to break the protocol (for example by getting some secret information). In some cases. we also have a trusted third party named server, who intervenes in case of disputes or who provides non repudiable evidence that certain transactions took place.

The goals of the particular security protocol that Alice and Bob run are application dependent, but may be loosely classified as follows:

- Secrecy is obtained when during the protocol, the information which Alice or Bob want to keep secret is not leaked.
- Authenticity means that whenever Alice follows the protocol and gets a message allegedly from Bob, then Bob actually sent the message (possibly to Alice, depending on how strong we want authentication to be).
- Confidentiality is obtained when Alice gets some secret information from Bob that is only shared by her, Bob, and possibly by some other entity trusted by her.
- Freshness (or timeliness) means that Alice is assured that the message was sent recently, after she has done a certain action or after a given time.
- Proof of identity is obtained when Alice is convinced that Bob is really the entity she is communicating with.
- Proof of delivery means that Alice gets some message that convinces her that Bob received (and read) some crucial information from her.
- Non repudiation usually refers to a mixture of the above in which the evidence in the hands of Alice must be sufficient to convince somebody else (usually server).

To make things rather more concrete, let us consider an example: the Needham-Schroeder protocol [13]. This protocol involves two agents A and B.

$$Message1 : A \rightarrow B : \{A, N_A\}_{pk_B}$$
$$Message2 : B \rightarrow A : \{N_A, N_B\}_{pk_A}$$
$$Message3 : A \rightarrow B : \{N_B\}_{pk_B}$$

These lines describe a correct execution of one session of the protocol. Each line of the protocol corresponds to the emission of a message by an agent (A for the first line) and a reception of this message by another agent (B for the first line).

In line 1, the agent A is the initiator of the session. Agent B is the responder. Agent A sends to B her identity and a freshly generated nonce N_A, both encrypted using the public key of B, pk_B. Agent B receives the message, decrypts it using his secret key to obtain the identity of the initiator and the nonce N_A.

In line 2, B sends back to A a message containing the nonce N_A that B just received and a freshly generated nonce N_B. Both are encrypted using the public key of A, pk_A. The initiator A receives the message and decrypts it, A verifies that the first nonce corresponds to the nonce she sent to B in line 1 and obtains nonce N_B.

In line 3, A sends to B the nonce N_B she just received encrypted with the public key of B. B receives the message

and decrypts it. Then B checks that the received nonce corresponds to N_B.

The goal of this protocol is to provide authentication of A and B. When the session ends, agent A is sure that she was talking to B and agent B is sure that he was talking to A. To ensure this property, when A decodes the second message, she verifies that the person she is talking to correctly put N_A in it. As N_A was encrypted by the public key of B in the first message, only B could infer the value of N_A. When B decodes the third message, he verifies that the nonce is N_B. As N_B only circulated encrypted by the public key of A, A is the only agent that could have deduced N_B. For these two reasons, A thinks that she was talking to B and B thinks that she was talking to A.

The protocol seems very secure. However, there are still many attacks which we will consider in later sections.

IV. Introduction About RTS And PAT model checker

RTS (Real-time System) is presented in [23], as an dialect of *Timed CSP* with extensions like variables, event operations, and so on. Interested readers are strongly recommended to refer to [23] for its operational semantics. An *RTS* process is a timed process defined as the following BNF, where P and Q range over processes, $e \in \Sigma$ is an observable event, b is a Boolean expression on global variables or process parameters and d is an integer constant.

$P = Stop \mid Skip$	– primitives
$\mid action \rightarrow P$	– data-operation prefixing
$\mid if\ b\ then\ P\ else\ Q$	– if-then-else
$\mid P \Box Q$	– general choice
$\mid P \parallel Q$	– parallel composition
$\mid P;\ Q$	– sequential composition
$\mid P \setminus X$	– hiding
$\mid P \mathrel{\widehat{=}} Q$	– process referencing
$\mid Wait[d_0, d_1]$	– delay
$\mid P\ timeout[d]\ Q$	– timeout
$\mid P\ interrupt[d]\ Q$	– timed interrupt
$\mid P\ within[d_0, d_1]$	– react within some time
$\mid P\ waituntil[d]$	– wait until
$\mid P\ deadline[d]$	– deadline

Process *Stop* does nothing but idling. Process *Skip* terminates (possibly after some idling). Process $e \rightarrow P$ engages in event e first and then behaves as P. Notice that e may be an abstract event or a data operation, e.g. written in the form of $e\{x = 5;\ y = 3;\ \}$ or an external C# program. The data operation may update data variables (and is assumed to be executed atomically). A guarded process is written as *if b then P else Q*. If b is true, then it behaves as P, else it behaves as Q. Parallel composition of two processes is written as $P \parallel Q$, where P and Q may communicate via multi-party event synchronization or shared variables. Process $P;\ Q$ behaves as P until P terminates and then behaves as Q immediately. Given a channel ch with predefined buffer size, process $ch!exp \rightarrow P$ evaluates the expression exp (with the current valuation of the variables) and puts the value into the respective buffer and then behaves as P. Process $ch?x \rightarrow P$ gets the first element in the respective buffer, assigns it to variable x and then behaves as P.

Timed process constructs are used to capture common real-time system behavior patterns. Process $Wait[d]$ idles for exactly d time units. In process $P\ timeout[d]\ Q$, the first observable event of P shall occur before d time units elapse (since the process starts). Otherwise, Q takes over control after exactly d time units elapse. Process $P\ interrupt[d]\ Q$ behaves exactly as P (which may engage in multiple observable events) until d time units elapse, and then Q takes over control. Process $P\ within[d]$behaves as P but will not terminate before d time unit elapse. Process $P\ deadline[d]$ constrains P to terminate before d time units.

Our home-built model checker *PAT*[1] (Process Analysis Toolkit) is designed to apply state-of-the-art model checking techniques for system analysis. PAT [22][21] supports a wide range of modeling languages including *CSP*♯ (short for communicating sequential programs), which shares similar design principle with integrated specification languages like TCOZ [10][11]. The verification features of *PAT* are abundant in that on-the-fly refinement checking algorithm is used to implement Linear Temporal Logic (LTL) based verification, partial order reduction is used to improve verification efficiency, and *LTL* based verification supports both event and state checking. Furthermore, *PAT* has been enhanced to accept *RTS* models, for verifying properties such as deadlock freeness, divergence freeness, timed refinement, temporal behaviors, etc [23].

V. Modeling Security Protocols In PAT

The typical security protocol involves some agents and perhaps a server that performs some service such as key generation, translation and certification. For illustration, we reconsider the Needham-Schroeder protocol as described in section I.

Modeling the sender and receiver

A trustworthy agent can take one of two roles, namely initiator (the sender of message 1, designated A above), or responder (the sender of message 2, designated B). It can be supposed that all agents can take either of these roles, and it may be the case that we allow an agent to be able to run several different instances of the protocol at once. However, for simple illustration, we only build agents for single run of the initiator and responder roles.

Each of the three messages in Needham-Schroeder protocol above are sent by one process and received

[1] http://www.patroot.com

by another. The view that each process has the running protocol is the sequences of sent and received messages are following:

A's view (as initiator):

- Message 1: A sends to B: $\{A, N_A\}_{pk_B}$
- Message 2: A gets from B: $\{N_A, N_B\}_{pk_A}$
- Message 3: A sends to B: $\{N_B\}_{pk_B}$

B's view (as responder):

- Message 1: B gets from A: $\{A, N_A\}_{pk_B}$
- Message 2: B sends to A: $\{N_A, N_B\}_{pk_A}$
- Message 3: B gets from A: $\{N_B\}_{pk_B}$

First of all, we consider the semantic of initiator A. At the beginning, A sends the message $\{A, N_A\}$ encrypted with public key B. After that, A gets a message from the responder which is encrypted by public key A. After decrypting this message using A's private key, if A get the same value N_A which sent to responder before, A continues sending the message $\{N_B\}_{pk_B}$ to responder to complete the protocol. If A cannot get the value N_A from responder's message, he only needs to stop the running.

We will represent the initiator who communicates with the responder with identity "receiver" by the RTS process Initiator(receiver). We use a channel, named *environment*, to capture the environment in which the messages are sent and receive. The process can be defined by:

$$\begin{aligned} Initiator(receiver) \quad = \quad & message1\_From\_A\_To\_receiver \rightarrow \\ & environment!key.receiver.Na.A \rightarrow \\ & environment?key.A.Na.nx \rightarrow \\ & message3\_From\_A\_To\_receiver \rightarrow \\ & environment!key.receiver.nx \rightarrow \\ & Skip; \end{aligned}$$

The responder can be defined similarly:

$$\begin{aligned} Responder(sender) \quad = \quad & environment?key.B.nx.sender \rightarrow \\ & message2\_From\_B\_To\_sender \rightarrow \\ & message2.B.sender \rightarrow \\ & environment!key.sender.nx.Nb \rightarrow \\ & environment?key.B.Nb \rightarrow \\ & Resp\_commit.sender.B \rightarrow \\ & Skip; \end{aligned}$$

The behaviors of intruder

If the sender and receiver are in a world where their message are transmitted reliably and where there is no other entity generating messages to put into the communication medium, then it seems most unlikely that anything could go wrong. However, this world, of course, is never true. The way in which this risky world of communication is modeled is by adding an intruder process into the network, who is given special powers to tamper with the messages that pass around. The only limitations to what the intruder can do are that his source of knowledge (aside from things he knows initially or invents for himself) is what he observes the communication, and that he is constrained by the rules of encryption. In other words, he cannot read the mind of another agent to know some secret, and can only decrypt an encrypted message if he has the appropriate key.

We want to model the intruder as a process that can perform any attack that we would expect a real-world intruder to be able to perform. Thus the intruder should be able to:

- Overhead and intercept any messages being passed in the system, and he can
- Decrypt messages that are encrypted with his public key so as to learn new knowledge, or
- Introduce new messages into the systems, using his knowledge he knows, or
- Replay any message he has seen (possibly changing plain-text parts), even if he does not understand the contents of the encrypted part.

We assume that the intruder is a user of the computer network, so can take part in normal runs of the protocol, and other agents may initiate runs of the protocol with him. We will define the most general intruder who can act as above. We consider an intruder with identity I, with public key K_i who initially knows a nonce N_i. We also keep track the knowledge of intruder during the transaction by using some variable: kNa (kNb) is knowledge of intruder with N_A (N_B). The RTS process represents the intruder in Needham-Schroeder protocol is:

$$\begin{aligned} Intruder() = \ & environment?key.x.Na.A \rightarrow \\ & \{if \ (x == I) \\ & \quad decript\ the\ message\ and\ learn\ knowledge \\ & \quad \{kNa = true; \ \} \\ & else\ if \ (x == B) \\ & \quad //forward\ the\ message \\ & \quad environment!key.B.Na.A \\ & \} \\ & \rightarrow Intruder() \\ & \Box \\ & ...\ //similar\ for\ message\ 2\ and\ 3 \\ & \Box \\ & [kNa == true]//fakemessage1 \\ & environment!key.B.Na.A \rightarrow Intruder() \\ & \Box \\ & ...\ //similar\ for\ message\ 2\ and\ 3 \\ & \Box \\ & //\ Act\ as\ an\ agent \\ & message1\_From\_I\_To\_B \rightarrow \\ & environment!key.B.Ni.I \rightarrow Intruder(); \\ & ...\ //other\ behaviors \end{aligned}$$

Network together

We now have the idea of how reliable the sender, receiver and intruder operate. In this part, we will see how these are put together into a network that can be used to test the resilience of the protocol. The approach taken is to provide a CSP description of a generalization of the Dolev-Yao model [3] as in Figure 1. Here it is assumed that the communications medium is entirely under the control of the enemy, which can block, readdress, duplicate, and fake messages.

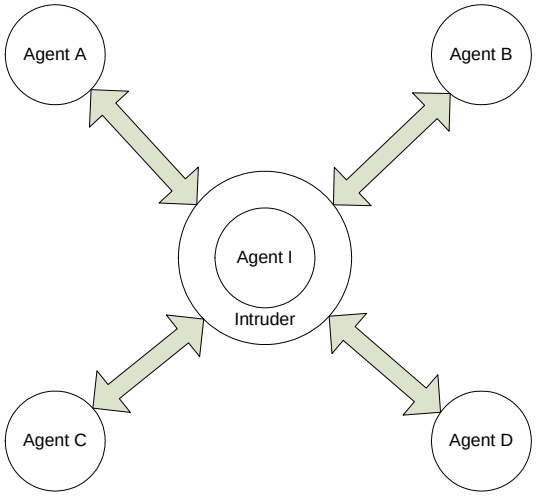

Figure 1. Dolev-Yao model

The result system is given by: $System \quad =||_{Agent_X} [Initiator(Agent_X) \;||\; Responder(Agent_X)] \;||\; Intruder()$

VI. Expressing protocol goals

Cryptographic protocols are used to ensure some security properties. Hence validation of cryptographic protocols is a deeply investigated domain. The objective of this research field is to prove formally that a given protocol verifies a given property. Let us first present which properties can be of interest for a protocol.

A. Secrecy

The secrecy property concerns a message used by the protocol. This message is typically a nonce or a secret key that should not become public at the end of the protocol.

As in section III, we can check the secrecy of some secret information by checking the leak of the information captured by some variable at the end of the run. For example, in the above Needham-Schroeder model, we want to keep the secrecy of the nonce of A and B which is captured by the variable kNa and kNb (initially are false, meaning that the intruder doesn't know NA and NB) and define the behavior of the intruder as in above. We can check this secrecy by adding the assertion in PAT:

$$\#define \quad secrecy \quad (kNa == true \;\; || \;\; kNb == true);$$
$$\#assert \quad Protocol \;\; |= \;\; \Box \; !(secrecy);$$

The symbol $\Box$ means that "always". The result of the assertion is not valid. That is, there exists an execution path where this secrecy property is not satisfied. PAT found a counter example:

$$message1\_From\_A\_To\_I \rightarrow environment!key.I.Na.A$$
$$\rightarrow environment?key.A.Na.Ni$$

The counter example shows that: the Intruder can act as an agent and start the session with Alice and get the N_A during the run.

B. Authentication

Entity authentication is concerned with verification of an entity's claimed identity. An authentication protocol provides an agent B with a mechanism to achieve this: an exchange of messages that establishes that the other party A has also been involved in the protocol run. This provides authentication of A to B: an assurance is provided to B that some communication has occurred with A.

Authentication of initiator by responder
We introduce signal Resp_commit_A_B into the description of B's run of the protocol to mark the point at which authentication of initiator A to responder B is taken to have been achieved. The value "true" of Resp_commit_A_B in B's protocol means simply that: "Responder B has completed a protocol run apparently with Initiator A".

The signal Init_running_A_B in A's run of the protocol are introduced to mark the point that should have been reached by the time the responder B performs the Resp_commit_A_B. The value "true" of Init_running_A_B meaning in A's protocol run means simply that: "Initiator A is following a protocol run apparently with responder B".

If an Init_running_A_B signal always have occurred by the time the Resp_commit_A_B signal is performed, then authentication is achieved. This is authentication of initiator by responder and is illustrated in Figure 2.

Adding these events into the RTS description of Needham-Schroeder protocol, we can check the authentication of initiator by responder by following assertion:

$$\#assert \quad Protocol() \;|=$$
$$\Box(Resp\_commit\_A\_B \rightarrow Init\_running\_A\_B);$$

The result of the assertion is that the formula is not valid. This means that there exists an execution path where this LTL formula is not satisfied. As expected, PAT discovers that the protocol does not satisfy the above property. It found a counter example which we can explain as:

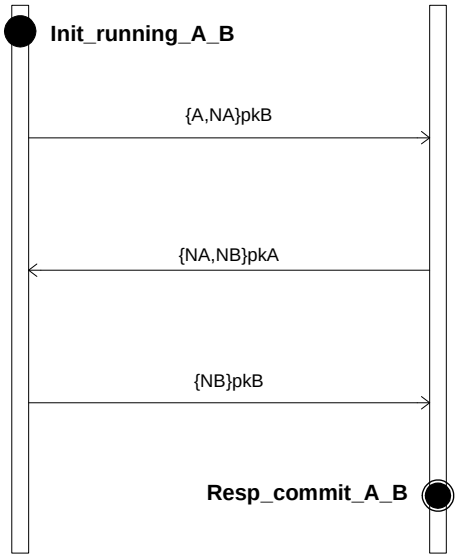

Figure 2. Authentication of initiator by responder

$$
\begin{array}{lcl}
\textit{Message } a1 : A \rightarrow I & : & \{A, N_A\}_{Pk_I} \\
\textit{Message } b1 : I(A) \rightarrow B & : & \{A, N_A\}_{Pk_B} \\
\textit{Message } b2 : B \rightarrow I(A) & : & \{N_A, N_B\}_{Pk_A} \\
\textit{Message } a2 : I \rightarrow A & : & \{N_A, N_B\}_{Pk_A} \\
\textit{Message } a3 : A \rightarrow I & : & \{N_B\}_{Pk_I} \\
\textit{Message } b3 : I(A) \rightarrow B & : & \{N_B\}_{Pk_B}
\end{array}
$$

Note that in this attack, intruder I is actually a recognized user, that is he is known to the other users and has a certificated public key. Alice starts a protocol run with the intruder I, thinking that he is a trust user. However, the intruder does not respond to Alice as the expected way. He used Alice's nonce to initiate another run with Bob but inserting Alice's name instead of his own. The notation I(A) denotes I generating the message, but pretending that it comes from A. Bob responds with his nonce N_B but he will encrypt it with Alice's public key as he thinks that he is contacting with Alice. This is exactly Alice is expecting from Intruder and she proceeds the next step: she decrypts it and send a message back to I containing N_B encrypted by I's public key. I now can decrypt this message and get N_B. Intruder I then construct the final message of the run he initiated with Bob: encrypt N_B under Bob's public key.

At the end of this, we have two interleaved runs of the protocol with Intruder sitting in the middle. Alice thinks that she and Intruder share knowledge of N_A and N_B. Bob thinks that he is running the protocol with Alice. Thus, the Intruder has created the mismatch in Alice and Bob's perception. The above section gives an idea about the variety and subtlety of the attacks to which protocols may be vulnerable.

Authentication of responder by initiator

This property can be check similarly as authentication of initiator by responder. The only difference is that we add the signal Init_commit_A_B at the beginning description of A's and the signal Resp_running_A_B at the end of B's run of the protocol. For shortly, we do not go into the details.

C. Non-repudiation

Non-repudiation ensures that the author of a message cannot later claim not to be the author. There is a proof that the sender sent the message. This is an indispensable property for the electronic commerce protocols, the seller needing to prove to the bank that the client has really paid. Non-repudiation can be checked easily using LTL assertion. An example about non-repudiation can be found at the Fair Non-repudiation protocol test case in [25].

D. Anonymity

Anonymity ensures that the identity of an agent is protected with respect to the message that he sent. For example, in a voting protocol the vote must not be linked back to the voter who cast it. In this case, the messages themselves do not have to be secret, only their association with a particular agent. This property can be consider as the inversion of non-repudiation.

E. Integrity

Integrity is usually meant that data cannot be corrupted, or at least that any such corruption will always be detected. In this sense, integrity can be considered as the corollary of the authentication property. This follows from the face that in the authentication, we required that the contents of the output message match that of the input message. If it were possible for a corrupted message to be accepted then this would lead to a violation of the authentication property and we would think the protocol to be flawed. This property can be checked in the same manner of secrecy property. We can refer to the Protocol for cetified email test case in [25] for further details.

VII. SEVE GRAMMAR AND PAT TRANSLATION

The RTS method has proved remarkably successful, and has been applied to find attacks upon a number of protocols [6] [8]. However, the task of producing the RTS description of the system is very time-consuming, and only possible for people well practiced in RTS-and even the experts will often make mistakes that prove hard to spot.

We create a language named SEVE and make a translation SEVEtrans to automate this process. The user specifies the protocol using a more abstract notation, similar to the notation appearing in the academic literature, and SEVEtrans compiles this into RTS code, suitable for checking using PAT. The most interesting thing in the translation is that the user do not need to specify the behavior of the intruder, which is very complicated. By observing the general intruder behavior as described in section V, the translator will

automatically generate all the possible attack of the intruder. This paper does not aim to give a tutorial in the use of SEVE, merely to give the reader an overview of what it does. Those interested in learning more are referred to the manual [25].

In this section we illustrate the input format by describing an input file suitable for analyzing the three message version of the Needham-Schroeder Public Key Protocol as described in section I.

The protocol description consists of five parts: a message list, defining the sequence of messages that constitutes a normal protocol run; a declaration of the types of the variables appearing in the message list; a declaration of the initial knowledge of the agents ; a declaration of the number and name of actual agents and intruder taking part in the protocol; and a specification of what the protocol is supposed to achieve.

For the Needham-Schroeder Public Key Protocol, the exchange of messages may be represented as follows:

#Protocol description
$a \rightarrow b : \{a, na\}kb;$
$b \rightarrow a : \{na, nb\}ka;$
$a \rightarrow b : \{nb\}kb;$

The protocol description part is as close as to the actual description of the protocol, so we do not need to explain more. The declaration of types and initial knowledge parts are also self-explained:

#Variables
Agents: $a, b;$
Nonces: $na, nb;$

#Initial
a **knows** $na, ka;$
b **knows** $nb, kb;$

In the real system, we can declare the number and name of the actual agents, servers and intruders playing in this protocol:

#System
Initiator: *Alice*;
Responder: *Bob*;

#Intruder
Intruder: *Jeeve*;
Intruder knowledge: $\{ni\};$

The last thing we concern is the verification part, which we support many kinds of verification properties: secrecy, agent authentication, data authentication, integrity, non repudiation and so on.

#Verification
Data secrecy: $\{na$ **of** *Alice*$\}, \{nb$ **of** *Bob*$\};$
Agent authentication:
Bob **is authenticated with** *Alice* **using data** *nb*;
Alice **is authenticated with** *Bob* **using data** *na*;

Now we can use the SEVEtrans to translate this description into RTS model, and use PAT tool to check the protocol. At this point, we can see how simply we define a security protocol. In the SEVE language, we support many kind of cryptographic mechanisms such as symmetric and asymmetric keys, public and private keys, signature and hash keys. We also allow infinite number of agents attending protocol. Moreover, besides the common security properties as described above, we support manual user assertion such as checking one message will eventually or always come, a message will come before or after another message come, and so on. If using the key word "timestamp" and other RTS language feature such as "Wait", "Timeout", "Interrupt" in the declaration, the user can check the timed security protocols. The full description of SEVE language and other examples are available at [25].

VIII. Conclusion

In this paper we have given an overview of the RTS and PAT tool, for producing RTS descriptions of security protocols. We also give an overview of the SEVE language and SEVEtrans tool. The translation has revolutionized our approach to analyzing protocols. Previously, when we produced the RTS by hand, it would take about a day to code up a protocol; now it takes only a few minutes. In particular, making small changes to the protocol or the system to be checked typically requires only a couple of lines of the input file to be changed; when editing the RTS code by hand, the changes necessary were spread throughout the file, and it was hard to know whether you had remembered them all.

Also, it was easy to make mistakes when producing the RTS by hand, and these mistakes were hard to spot. When using SEVEtrans, errors are less common, most get caught by the compiler, and those errors that do get through are easier to spot because the file is so much shorter. The approach described in this paper has been applied to a number of other protocols, including the the Yahalom Protocol [9], A Fair Non-Repudiation Protocol [26], Computer-assisted verification of a protocol for certified email [12]. Some of these case studies are available via the [25]. The techniques seem to scale well to medium sized protocols, albeit with a reduction in the size of the system that can be studied; we have not yet looked at any large commercial protocols, although it would certainly be interesting to do so. In the future, we will enhance the tool to support more features of security properties and make the counter example more visualization to the user.

REFERENCES

[1] G. Bella and L. C. Paulson. Kerberos Version IV: Inductive Analysis of the Secrecy Goals. In *European Symposium on Research in Computer Security*, volume 1485, pages 361–375–107, 1999.

[2] B. Blanchet. Automatic verification of correspondences for security protocols. volume 19, pages 363–434. Journal of Computer Security, 2009.

[3] D. D and A. Yao. On the Security of Public Key Protocols. In *IEEE Transactions on Information Theory*, volume 29, pages 198–208, 1983.

[4] C. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.

[5] U. S. John C. Mitchell, Mark Mitchell. Automated analysis of cryptographic protocols using Murphi. In *IEEE Symposium on Security and Privacy*, pages 141–151, 1997.

[6] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proceedings of TACAS*, pages 147–166, 1996.

[7] G. Lowe. Casper : A Compiler for the Analysis of Security Protocols. In *Journal of Computer Security*, volume 6, pages 53–84, 1998.

[8] J. Lowe and A. W. Roscoe. Using CSP to detect errors in the TMN protocol. In *Technical Report, Department of Mathematics and Computer Science, University of Leicester*, 1996.

[9] B. M., A. M., and N. R. A logic for authentication. volume 1 of *ACM Trans. Comput.Syst.*, pages 18–36, 1999.

[10] B. Mahony and J. Dong. Blending Object-Z and Timed CSP: An introduction to TCOZ. In *In Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, pages 95–104, 1998.

[11] B. Mahony and J. Dong. Timed Communicating Object Z. In *IEEE Transactions on Software Engineering*, volume 26, pages 150–177, 2000.

[12] B. B. Martn Abadi. Computer-assisted verification of a protocol for certified email. In *Static Analysis, 10th International Symposium*, pages 316–335, 2005.

[13] R. M. Needham and M. D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Communications of the ACM*, 21:993–999, 1978.

[14] L. C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. In *Journal of Computer Security*, volume 6, pages 85–128, 1998.

[15] C. S. Escobar and J.Meseguer. A rewriting-based inference system for the NRL protocol analyzer and its meta-logical properties. *Theoretical Computer Science*, 367:162–202, 2006.

[16] B. Schneider. *Applied Cryptography : Protocols, Algorithms, and Source Code in C*. 1994.

[17] S. Schneider. Security Properties and CSP. *IEEE Symposium on Security and Privacy*, pages 174–187, 1996.

[18] S. Schneider. Verifying Authentication Protocols in CSP. In *IEEE Transactions on Software Engineering*, volume 24, pages 741–758, 1998.

[19] S. Schneider and R. Delicata. Verifying Security Protocols: An Application of CSP. In *25 Years Communicating Sequential Processes*, pages 246–263, 2004.

[20] H. R. Shahriari and Jalili. Using CSP to Model and Analyze Transmission. In *Networking and Communication Conference*, pages 42–47, 2004.

[21] J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. In *21th International Conference on Computer Aided Verification (CAV 2009)*, 2009.

[22] J. Sun, Y. Liu, J. S. Dong, and H. Wang. Specifying and Verifying Event-based Fairness Enhanced Systems. In *10th International Conference on Formal Engineering Methods (ICFEM 2008)*, 2008.

[23] J. Sun, Y. Liu, J. S. Dong, and X. Zhang. Verifying Stateful Timed CSP Using Implicit Clocks and Zone Abstraction. In *Proceedings of the 11th IEEEInternational Conference on Formal Engineering Methods (ICFEM 2009)*, volume 5885 of *Lecture Notes in Computer Science*, pages 581–600, 2009.

[24] P. Syverson and P. V. Oorschot. On unifying some cryptographic protocol logics. *IEEE Symposium on Security and Privacy*, 23:14–28, 1994.

[25] L. A. Tuan. The *RTS* model of security protocols. http://www.comp.nus.edu.sg/˜pat/security-rts.zip, 2009.

[26] J. Zhou and D. Gollmann. A Fair Non-Repudiation Protocol. In *Proc. of the 15th. IEEE Symposium on Security and Privacy*, pages 55–61, 1996.