Petri nets in cryptographic protocols

Federico Crazzolara*  Glynn Winskel
Computer Laboratory
University of Cambridge
Pembroke Street
Cambridge CB2 3QG, UK
{fc232, gw104}@cl.cam.ac.uk

Abstract

A process language for security protocols is presented together with a semantics in terms of sets of events. The denotation of process is a set of events, and as each event specifies a set of pre and postconditions, this denotation can be viewed as a Petri net. By means of an example we illustrate how the Petri-net semantics can be used to prove security properties.

1. Introduction

Security protocols are concerned with exchanging messages between agents via an untrusted medium. The protocols aim at providing guarantees such as confidentiality of transmitted data, user authentication, anonymity etc. A protocol is often described as a sequence of messages, and usually encryption is used to achieve security goals.

As an example consider the Needham-Schröder-Lowe (NSL) protocol:

Suppose $A$ and $B$ are agent names standing for agents Alice and Bob. The protocol describes an interaction between the initiator Alice and the responder Bob as following: Alice sends to Bob a new nonce $n$ together with her own agent name $A$ both encrypted with Bob’s public key. When the message is received by Bob, he decrypts it with his secret private key. Once decrypted, Bob prepares an encrypted message for Alice that contains a new nonce together with the nonce received from Alice and his name $B$. Acting as responder, Bob sends it to Alice, who recovers the clear text using her private key. Alice convinces herself that this message really comes from Bob, by checking whether she got back the same nonce sent out in the first message. If that is the case, she acknowledges Bob by returning his nonce. He will do the same test.

In our approach we follow the assumptions of the Dolev-Yao model [6]:

- Cryptography is treated as a black box, that is, encrypted messages are assumed to be unforgeable by anyone who does not have the right key to decrypt. Keys are assumed to be unguessable.
- The adversary is an active intruder, not only capable of eavesdropping on messages passing through the communication medium. He can also modify, replay and suppress messages, and even participate in the protocol, masquerading as a trusted user.

The language we present was inspired by Paulson’s inductive approach to crypto-protocol analysis [11, 12].

---

*A BRICS Basic Research in Computer Science, Centre of the Danish National Research Foundation.
prove secrecy in Section 3. We first introduce the syntax of the language and then show how to program the NSL protocol. We then present a Petri-net semantics and a variation of it, which allows us to show how to program the NSL protocol. We then present a

Indexes

The set

2. A language for security protocols

We first introduce the syntax of the language and then show how to program the NSL protocol. We then present a Petri-net semantics and a variation of it, which allows us to prove secrecy in Section 3.

2.1. The language

We start by giving the syntactic sets of the language:

- A set of Names, with \( n, m, A, B \) ranging over it
- A set of indexes, \( i \in \text{Indexes} \)
- Variables over names \( x, y, z \in \text{Nvar} \)
- Variables over messages \( X, Y \in \text{Mvar} \)

The set Indexes contains Names and the set \( \text{IV} \) of natural numbers. The other syntactic sets of the language are built up by the grammar shown in Figure 1.

\[
\begin{align*}
\text{Values} & : = n \mid x \\
\text{Keys} & : = P_{ab}(v) \mid P_{riv}(v) \mid Key(v, v') \\
\text{Messages} & \quad \quad : = v \mid k \mid X \mid (M, N) \mid \{M\}_k \\
\text{Patterns} & \quad \quad : = v \mid k \mid X \mid (\Pi, \Pi') \\
\text{Processes} & \quad \quad : = \text{nil} \\
\quad \quad \quad \quad \mid \text{new}(x), P \\
\quad \quad \quad \quad \mid \text{out } M, P \\
\quad \quad \quad \quad \mid \text{in } \Pi, P \\
\quad \quad \quad \quad \mid [M = N], P \\
\quad \quad \quad \quad \mid [M > \Pi], P \\
\quad \quad \quad \quad \mid \bigcup_{i \in I} P_i
\end{align*}
\]

Figure 1. Syntactic sets

Whereas Paulson provides an analysis of protocols in a case-by-case manner, formalising each protocol directly by a set of rules to specify how traces of events are built-up, we sought a single language and semantics, capable of expressing a broad range of protocols. The result is an asynchronous language, that resembles Linda [8] and an asynchronous version of the Spi calculus [1] in some ways. We provide an event based-semantics for the language. This semantics describes a process by its set of events giving rise to a Petri net and supports the local reasoning that appears to be useful in analysing cryptographic protocols. The event-based semantics also shows the connection to other approaches such as the inductive method [12] and strand spaces [16]. Though, in this paper we do not prove this correspondence. An example helps us to illustrate how proof techniques that are used in strand space like proofs can be transferred to our model.

We use injections with disjoint range to distinguish between public, private and symmetric keys. Public keys can be used for encryptions such as \( \{M\}_{P_{ab}(v)} \) and private keys can be used for decryption as for example \( \{M\}_{P_{riv}(v)} \). A symmetric key instead, may be used both for encryption and decryption, thus \( \{M\}_{Key(v, v')} \) may or may not be ciphertext. The set of messages is obtained from the grammar in Figure 1, equated with the least substitutive equivalence relation such that:

\[
\begin{align*}
\{\{M\}_{P_{ab}(v)}\}_{P_{riv}(v)} & = M \\
\{\{M\}_{P_{riv}(v)}\}_{P_{ab}(v)} & = M \\
\{\{M\}_{Key(v, v')}\}_{Key(v, v')} & = M
\end{align*}
\]

The resulting message algebra allows a combined treatment of encryption and decryption. With these message equations we are making some assumptions on the underlying cryptographic schemes:

(1) This equation reflects a property which is required for each public key encryption scheme, and allows an agent, if in possession of the right private key, to recover the clear text from a cipher.

(2) RSA can both be used as encryption scheme and as digital signature scheme. The private key is used to sign a message and the public key to recover, if possible, the original message and so verify the signature. This equation may be dropped or modified if one wished to assume different encryption primitives.

(3) This equation implicitly assumes that the same symmetric key is never used consecutively for multiple encryptions.

We say that a message is in reduced form if none of its components can be further reduced using the above message equations in a left to right fashion. Given a message \( M \) we write \( \text{red}(M) \) for its reduced form. We can show that for every message there is a unique reduced form. Moreover our approach guarantees that clear text can be obtained from ciphertext only using the right key [5].

Let \( \text{var}(M) \), \( \text{var}(\Pi) \) be the variables of a message \( M \) and pattern \( \Pi \) respectively. The set of free variables of a process term \( P \), \( \text{fv}(P) \), is defined by structural induction in
that are not free are said to be bound. Informally speaking, a variable is closed, as is a message without variables. We will abbreviate by \(\text{Proc} \) the program of initiator \( A \) communicating with \( B \) and by \( \text{Resp}(B) \) the program of responder \( B \). The code of both an arbitrary initiator and an arbitrary responder is given in Figure 3. In programming the protocol we are forced to formalise aspects that are hidden in the informal description, such as the creation of new nonces and the decryption of messages.

We can model the intruder by directly programming it as a process. Figure 3, shows a very general, active intruder, as inherited from the Dolev-Yao model. \( \text{Bad} \) is the set of agents that are corrupted, because their private key has been leaked. The spy has the capability of composing different eavesdropped messages, decomposing composite information, use cryptography by means of the available keys. In fact available keys are all the public keys and the leaked private keys. Choosing a different program for the spy corresponds to restricting or augmenting its power, e.g., to passive eavesdropping or active falsification.

The whole system is obtained by putting all components in parallel. Components are replicated, to model multiple concurrent runs of a protocol.

### 2.2. Example: Programming the NSL protocol

Given a set of \( \text{Agents} \subseteq \text{Names} \) we assume that each agent can participate in the protocol both as initiator and responder with any other agent. Abbreviate by \( \text{Init}(A,B) \) the program of initiator \( A \) communicating with \( B \) and by \( \text{Resp}(B) \) the program of responder \( B \). The code of both an arbitrary initiator and an arbitrary responder is given in Figure 3. In programming the protocol we are forced to formalise aspects that are hidden in the informal description, such as the creation of new nonces and the decryption of messages.

We can model the intruder by directly programming it as a process. Figure 3, shows a very general, active intruder,
away from each other, separated by actions of other evolving components. One goal of the following semantics is to take better account of the dependency among actions of the same component. It has been proven correct with respect to the operational semantics given in [5].

The building blocks of our semantics are events. Events consist of an action together with the preconditions necessary to enable it, and postconditions satisfied once it has occurred. The actions label events and are given by the grammar \( \alpha ::= \text{out}(M) \mid \text{in}(M) \mid i : \alpha \) where \( i \in \text{Indexes} \).

**Definition 2.1** An event is a tuple \( e = (\text{Pre} \circ \alpha, \text{Post}) \), with \( \text{Pre} = \{c_1, \ldots, c_k\} \) a set of preconditions, \( \alpha \) an action, and \( \text{Post} = \{d_1, \ldots, d_l\} \) a set of postconditions.

We will often represent an event as:

```
\begin{align*}
  &c_1 & \cdots & c_k & \alpha & d_1 & \cdots & d_l \\
  \end{align*}
```

As, for instance, in [18] we will use \( e \) for the preconditions of \( e \) and \( e' \) for its postconditions. We have three kinds of conditions:

- Control conditions \( P \in \text{Proc} \), for keeping track of a control point in the evolving program.
- Name conditions \( s \subseteq \text{Names} \), expressing that \( s \) is the present set of names used.
- Network conditions \( (M) \) with \( M \in \text{Msg} \), expressing that the message \( M \) is present on the network.

Control conditions and name conditions are used up once a single event depending on them occurs. They are examples of Petri-net conditions which may hold with only multiplicity 1. Network conditions are of a special kind; they are “persistent”, meaning that once satisfied, they hold forever - they might even be represented by Petri-net conditions which hold with infinite multiplicity. In particular we will write \(^1)e\ for the “transient” preconditions of \( e \), meaning the ones concerning names and control. A marking \( M \) is a set of conditions that hold. These are the parts of the Petri net which evolve as events occur, in a way described later in Definition 2.2.

We assume a definition of variable substitution \( \sigma \) for messages and processes. We require it to substitute for a variable \( x \in \text{Nvar} \) a name and a closed and fully reduced message for a variable \( X \in \text{Mvar} \). We write \( \text{name}(P) \), \( \text{name}(M) \), and \( \text{name}(t) \) for the set of names respectively of a process \( P \), a message \( M \) and a set of messages \( t \). Further let \( \text{name}(\sigma) \) be the set of names of the substituted values or messages, listed in \( \sigma \). We assume those sets to be defined in the obvious way.

---

### Figure 4. Initial control conditions

We define a semantics that associates with every closed process term \( P \) a set of events and an initial marking, i.e. \( \llbracket P \rrbracket = (\text{Ev}(P), \mathcal{M}_0) \). The initial marking \( \mathcal{M}_0 = \{s_0, ((M_1)) \ldots, ((M_j))\} \cup \text{Ic}(P) \) has an initial name condition \( s_0 \) which is the set of names considered already as used from the start, and therefore we require \( \text{name}(M_1, \ldots, M_j) \subseteq s_0 \) and \( \text{name}(P) \subseteq s_0 \). We may assume from the beginning that the network already contains some messages \( M_1, \ldots, M_j \). An important part of the initial marking are the control conditions \( \text{Ic}(P) \), defined inductively on the size \(^1\) of process terms as shown in Figure 4; control is handed over to the different parallel components of the program. Figure 5 shows how to construct inductively on the size of a process the set of events \( \text{Ev}(P) \) associated with a closed process \( P \). We have three kinds of basic events:

- **New** events mark the next control point in the process where a new name \( n \) is substituted in for \( x \) and update the set of new names consequently. These events seem to be necessary if we want to give a Petri-net semantics without restricting the processes. We will investigate this further in the next section and show how to take, with some restriction, a more implicit account of new.
- **Out** events not only cause the program control to evolve, but also put the message \( M \) on the network. The message will always remain visible on the network as the “persistent” condition \( (\text{red}(M)) \) is marked.
- **In** events require a message, that matches the pattern, to be already present on the network and cause control to evolve.

Sometimes we may add more events than necessary in \( \text{Ev}(P) \). It is important to bear in mind that only the reachable ones will contribute to a computation.

As we have seen we defined our semantics by induction on the size of closed process terms. If we wanted a denotational semantics instead, defined by structural induction and

\(^1\)The size of a term is an ordinal measure of the height of operators in a term, and so for example \( \text{size}(\text{new}(x).P) = 1 + \text{size}(P) \).

\( \text{size}(\bigcup_{i \in I} P_i) = 1 + \sup_{i \in I} \text{size}(P_i) \).
A marking $\mathcal{M}$ is reachable, if $\mathcal{M}_0 \xrightarrow{e_0} \cdots \xrightarrow{e_{n-1}} \mathcal{M}_n = \mathcal{M}$.

A run of a protocol is a sequence of alternating marking and events, obtained applying the token game starting from the initial marking $\mathcal{M}_0$.

2.4. How to treat new implicitly

In this section we study how to treat new events as implicit events. We want to think of new as a binder rather than as a visible action. The idea is to move the effects to the first visible event that follows. We did something similar already for case and match. Although there were no problems in doing so for these two constructs, new events cause some difficulties. Unlike case they give rise to infinitely many possible substitutions. If the remaining program is a parallel composition of processes and the same variable is bound by new in more than one component, then events with different values for that variable can get confused. The following assumption allows us to treat new as a binder. It avoids the problematic situation where a new name is created right in front of a parallel composition.

Assumption 2.3 The system is a process that does not have a new$(x)$ immediately in front of a parallel composition.

Typically, security protocols are modelled as a parallel composition of processes, representing agents, each of which is a sequential process. For such processes the previous assumption holds.

As basic events we have out and in events as before, but no new events. Figure 6 shows the change in our previous semantics. Only the events for new$(x).Q$ change: Take all events of $Q[n \setminus x]$ and add name conditions to those events to which control is initially handed over. If they do not contain any name conditions, add the appropriate ones. If they already do contain name conditions because of a later new, we require $n$ already present in the name condition $s$ and ask for a precondition that does not contain it, i.e., $s \setminus \{n\}$.

3. The NSL protocol: secrecy

In this section we apply our framework to prove secrecy guarantees for the NSL protocol. We use the modified semantics of Section 2.4, where new is treated like a binder. We can do this because our NSL system is a parallel composition of straight line processes. The proof is along the lines of a strand-space argument [16].

We first give some facts that are true in general and are important building blocks in our proofs. Then we describe the events associated to the NSL protocol and prove some desired guarantees.
3.1. Definitions and useful principles

In the rest of this section we will make the following convention and write \( n \in \mathcal{M} \) to mean \( n \in s \) where \( s \in \mathcal{M} \). We write \( \eta \) for a partial run, i.e., a subinterval of a run of a protocol beginning and ending with either a marking or an event. We write \( e \in \eta \) if the event \( e \) occurs in the partial run \( \eta \) and \( M \in \eta \) if the marking \( M \) occurs in \( \eta \). If we write \( a \in \eta \) then \( a \) is either a marking or an event occurring in \( \eta \). We use \( <_{\eta} \) for the order relation among markings and events of a partial run \( \eta \) and \( \leq_{\eta} \) for its reflexive closure; so for instance, \( e <_{\eta} M \) means that the event \( e \) precedes the marking \( M \) in the sequence \( \eta \). If the partial run is understood from the context we omit the subscript \( \eta \). Furthermore when we write \( a <_{\eta} a' \) or \( a \leq_{\eta} a' \) we understand that \( a, a' \in \Omega \). We write \( \Box \) for the sub-message relation among messages in reduced form.

**Definition 3.1** Given a property \( \mathcal{P} \) of markings and events of a protocol, we write \( \mathcal{P}[\eta] \) for the invariance property \( \forall a \in \eta. \mathcal{P}(a) \) of a partial run \( \eta \).

**Convention 3.2** Given a property \( \mathcal{P} \) only on markings and \( \eta \) a partial run of a protocol, we may write \( \mathcal{P}[\eta] \) for the property: \( \forall \text{ markings } \mathcal{M} \in \eta. \mathcal{P}(\mathcal{M}) \). This property can be easily extended to an invariance property on markings and events.

The following fact is useful in proving properties of partial runs.

**Principle 3.3 (Well-foundedness Principle)** Let \( \eta \) be partial run and let \([a_1 \ldots a_j]\) and \([a_1 \ldots a_k]\) be subintervals of \( \eta \). Given a property \( \mathcal{P} \) on markings and events such that \( \mathcal{P}\{a_1 \ldots a_j\} \) and \( \neg \mathcal{P}\{a_1 \ldots a_k\} \) where \( a_j <_{\eta} a_k \) then there exists \( a_i \in \eta \) where \( a_j <_{\eta} a_i \leq_{\eta} a_k \) such that \( \mathcal{P}\{a_1 \ldots a_{i-1}\} \) and \( \neg \mathcal{P}\{a_1 \ldots a_i\} \).

A notion of freshness arises from our event-based semantics. We say that a name is fresh on the occurrence of an event, if that event introduced the name for the first time. Formally:

**Definition 3.4** Given an event \( e \), a name \( n \in \text{Names} \) is fresh on \( e \), written \( \text{Fresh}(n,e) \), if \( s \in e', n \not\in s \) and \( n \in s' \).

A name can be fresh only once in a protocol run: Once added to the set of used names, it will stay used.

**Principle 3.5 (Freshness Principle)** Let \( \eta \) be a complete run of a protocol, starting from an initial marking \( M_0 \).

1. \( \forall M \in \eta. \forall n \in M. (n \in M_0) \vee (\exists e \in \eta. e <_{\eta} M \land \text{Fresh}(n,e)). \)

2. Given \( n \in \text{Names} \), there exists at most one event \( e \) in \( \eta \) s.t. \( \text{Fresh}(n,e) \).

3. Given \( e \in \eta \) and \( n \in \text{Names} \) s.t. \( \text{Fresh}(n,e) \) then \( \forall M \in \eta. M \leq_{\eta} e \Rightarrow n \not\in M. \)

A principle that follows directly from the token game on our Petri net semantics is the following:

**Principle 3.6 (Precedence Principle)** Given \( \eta \) a run of a protocol from an initial marking \( M_0 \) then \( \forall e \in \eta. \forall c \in e \) either \( c \in M_0 \) or \( \exists e' \in \eta. e' <_{\eta} e \land c \in e' \).

A particular instance of this principle is saying that for every occurrence of an input event there exists a previously corresponding occurrence of an output event. Moreover, if the Precedence Principle is applied to control conditions, one can determine preceding events belonging to the same component.
3.2. NSL-protocol events

In Section 2.2 we have seen how to program the NSL protocol. Associated to the program is a set of events and an initial marking. We use the semantics of Section 2.4, with the implicit treatment of new as shown in Figure 6. We do not give the initial control conditions $Ic(NSL)$ explicitly. They will consist of parallel components of the system, indexed accordingly.

In Figure 7 and in Figure 8 we give the events of the components of the system. They extend in an obvious way (by indexing) for replication and parallel composition to the components that form the NSL system. In the rest of this paper we omit tagging. It will be clear from the context which is the component of the system we refer to. For convenience we put events together in nets by joining them at their corresponding control points. In this way the dependencies between events are better displayed. Sometimes we will write (\textit{true}) for a control condition that is clear from the context.

Let us start with the components of an arbitrary initiator and an arbitrary responder. The initiator and responder events are just as one would expect looking at the processes in Figure 3. Consider for example the initiator $Init(A,B)$. Since it is a straight line program its first set of events share as a control precondition the process that is the component itself. These first events are \textit{out} events, moreover each of them is fresh for a name $n$. This is as expected from the program, where the first action is an \textit{out} preceded by a \textit{new}. The next action in the initiator’s process is an \textit{in}. It will be executed only after the previous output and only if the network contains a message matching the required pattern. Following the input we have a \textit{case} and a \textit{match}, both constructs will allow us to pass the control to an event only if the message received is of the required form. There are \textit{in} events leading to \textit{nil} corresponding to unsuccessful applications of \textit{case} and \textit{match}. Finally we have \textit{out} events that wait for a marked control condition coming from the successful tests.

The intuition behind the responder events of Figure 7 and Spy events of Figure 8 is the same as explained for the initiator events. The spy is a component of the system, that usually remains fixed for all the protocols.

3.3. Secrecy

We show a lemma that is useful for the proofs of secrecy. In fact it is a secrecy theorem in its own right: If the private keys of the agents are not leaked, then they will stay secret during all the runs of the protocol. Furthermore, they will never appear as part of the content of a message sent on the medium. This has proven to be a useful lemma in previous approaches [12, 16]. We recall that $Bad$ is a set of corrupted agents, whose private keys are leaked to the intruder.

\textbf{Lemma 3.7} Let $\eta$ be a run of NSL. If $A \not\in Bad$ and $Priv(A)$ does not appear as component of network conditions in the initial marking $M_0$ then the invariance property $I[\eta] \equiv \forall M \in \eta. \forall((M)) \in M. Priv(A) \not\subseteq M$ holds (and analogously for $Priv(B)$).

\textbf{Proof.} Let $\eta$ be a run of NSL starting with the initial marking $M_0$. Obviously $I[M_0]$. Suppose that there exists $e \in \eta$ such that $I[M_0 \ldots e]$ and $\neg I[M_0 \ldots e,M]$. We prove that there is no such event $e$ associated to NSL that belongs to the protocol run $\eta$. The Token Game 2.2 tells us that among all the events associated to the NSL protocol, we have to consider only output events. An inspection of the agent nets (Figure 7), shows that there are no output events associated with agents that could mark a message containing $Priv(A)$. It remains to consider Spy events. For example take the event

$$\text{out } M \text{. out } N \text{. nil}$$

with $Priv(A) \subseteq M$. By the Token Game 2.2, the Precedence Principle 3.2, and the events in Figure 8, there is a marking $M' < e$ such that $((M,N)) \in M'$, which contradicts $I[M_0 \ldots e]$ since $Priv(A) \subseteq (M,N)$. The reasoning is similar for the other events, bearing in mind that $A \not\in Bad$.

We state and sketch the proof of secrecy for a nonce of a principal which uses the NSL protocol as a responder. Let $e^B_2$ be the second event, as appearing in the net of Figure 7, where $B$ responds to $A$ using nonces $n, m$.

\textbf{Theorem 3.8 (Secrecy)} If $A \not\in Bad$ and $B \not\in Bad$ then for every run $\eta$ of NSL such that $e^B_2 \in \eta$, the invariant $Sec[\eta] \equiv \forall (M) \in \eta. (m) \not\in M$ holds.

\textbf{Proof. (Sketch)} To prove this secrecy theorem we show a stronger invariant, following the proof strategy taken in [16]. Let $Sec'(\eta)$ be $\forall M \in \eta$ and $\forall((M)) \in M$ if $m \not\subseteq M$ then $\{ (m,(m,B)) \}_{Priv(A)} \subseteq M \cup \{ m \}_{Priv(B)} \subseteq M$. It is easy to see (Freshness Principle 3.5) that $Sec'(M_0 \ldots e^B_2)$\footnote{Footnote text}. Suppose there exists an event $e$ such that $Sec'(M_0 \ldots e)$ and $\neg Sec'(M_0 \ldots e,M')$. We look for the various possibilities for $e$. By the Token Game 2.2 we have just to inspect the output events. Using the principles previously described we can see that no initiator, responder, or spy event can produce a marking $M'$ such that $\neg Sec'(M_0 \ldots e,M')$. \hfill $\square$
Figure 7. Initiator and responder nets

Figure 8. Spy nets
4. Conclusions

The Petri net model and the inductive rules of Paulson are closely related. In Petri-net terms Paulson’s rules correspond to a particular kind of events, where both network and control conditions are “persistent” conditions. Our work is in many ways related to the strand-space approach [15, 16]. In fact the single processes, components of the system, can be viewed as strands and our proofs use them much in the same way. Strand spaces are closely related to event structures [17], another model emphasising the causal dependencies between events (more precisely, the bundles of a strand space, ordered by inclusion form a stable family [17] and so are isomorphic to the configurations of an event structure). There are well-known relations between Petri nets and event structures. The particular nets that we introduced, with “persistent” conditions, can be proven to correspond to safe nets. To do this colour the network conditions with information about sender and receiver, obtaining a coloured net, then use the trick illustrated in [18] to obtain an ordinary Petri net that is safe. There are similarities between our approach and one using multiset rewritings based on linear logic [3, 4], where logical formulas play a role similar to the processes. Petri nets have previously been applied to the verification of cryptographic protocols [2, 7, 10], though these approaches are very different.

This paper has been closely based on the first author’s Ph.D. progress report [5], a Ph.D research proposal. The language, its semantics and proof techniques continue to evolve. We presently restrict ourselves to a simpler language, with no case and match constructs and where all pattern-matching is done at input time. This avoids dealing with message equations but also makes fixed assumptions about the underlying cryptographic primitives. Moreover the new is incorporated in the out operator since usually a new nonce is created and sent out. The semantics can be simplified too, especially in the treatment of new events. Instead of a single global condition s specifying all the names in use, we currently use an individual condition for each name. We are using the model to unify a variety of approaches, process algebra, the use of strand spaces and Paulson’s inductive method. We hope especially that it will guide us to a more systematic method for verifying crypto-protocols and in particular to useful logics.

References