# Events in Security Protocols

Federico Crazzolara
Glynn Winskel

BRICS Report Series RS-01-13

ISSN 0909-0878 April 2001

# Events in Security Protocols

Federico Crazzolara[*] Glynn Winskel
{fc232, gw104}@cl.cam.ac.uk

Computer Laboratory
University of Cambridge

**Abstract**

The events of a security protocol and their causal dependency can play an important role in the analysis of security properties. This insight underlies both strand spaces and the inductive method. But neither of these approaches builds up the events of a protocol in a compositional way, so that there is an informal spring from the protocol to its model. By broadening the models to certain kinds of Petri nets, a restricted form of contextual nets, a compositional event-based semantics is given to an economical, but expressive, language for describing security protocols; so the events and dependency of a wide range of protocols are determined once and for all. The net semantics is formally related to a transition semantics, strand spaces and inductive rules, as well as trace languages and event structures, so unifying a range of approaches, as well as providing conditions under which particular, more limited, models are adequate for the analysis of protocols. The net semantics allows the derivation of general properties and proof principles which are demonstrated in establishing an authentication property, following a diagrammatic style of proof.

## 1 Introduction

The last few years have seen the emergence of successful intensional, event-based, approaches to reasoning about security protocols. The meth-

[*]**BRICS** Basic Research in Computer Science, Centre of the Danish National Research Foundation

ods are concerned with reasoning about the events that a security protocol can perform, and make use of a causal dependency that exists between events. For example, to show secrecy in a protocol it is shown that there can be no earliest event violating a secrecy property; any such event is shown to depend on some earlier event which itself violates secrecy—because the behaviour of the protocol does not permit such an infinite regress, the secrecy property is established. In a similar way, dependency between events is used to establish forms of authentication by showing that a sequence of communication events of one agent entails a corresponding sequence of events of the intended participant.

Both the method of strand spaces [THG98c, THG98a, THG98b] and the inductive method of Paulson [Pau99, Pau98] have been designed to support such an intensional, event-based, style of reasoning. Strand spaces are based on an explicit causal dependency of events, whereas in Paulson's method the dependency is implicit in the inductive rules, which might express, for instance, that the input of a message depends on its previous output. Both methods have successfully tackled a number of protocols though in an *ad hoc* fashion. Both make an informal spring from a protocol to its representation, as either a strand space or a set of inductive rules. Both methods do not address how to build up their representation of a protocol in a compositional fashion. Both are remarkably similar, to the extent that a proof using one seems at an informal level to suggest a proof using the other.

We show that Petri nets, and specifically a restricted form of contextual nets [MR95], provide a common framework in which to understand both the strand-space and inductive methods, and it seems, although we understand it less well, the recent multiset rewriting and linear-logic methods of [CDL$^+$99, CDL$^+$00].[1] But, more importantly, by moving to a broader class of models we can show how event-based models can be structured in a compositional way and so used to give a formal semantics to security protocols which supports proofs of their correctness. To make the case, and provide semantics to a whole range of protocols once and for all, we study the semantics of **SPL** (Security Protocol Language).

We demonstrate the usefulness of the net semantics in deriving (in contrast to postulating) proof principles for security protocols and apply them to prove an authentication property—the diagrammatic style

[1]Although Petri nets have been used before in the analysis of security protocols, e.g. [NT92], our use is significantly different and more closely related to the strand-space and inductive methods.

of proof may be of interest in itself. (We hope in future to address the issue of logics for security protocols and see the semantics of **SPL** as providing a good basis for a model theory.) We establish precise relationships between the net semantics and transition semantics, strand spaces, inductive rules, and trace languages and event structures. The results formally back up the adequacy of strand-space and inductive-rule representations for broad classes of security protocols and properties—showing when nothing is lost in moving to these more restrictive models.

## 2 Security protocols

As a running example we consider the Needham-Schröder-Lowe (NSL) protocol:

$$\begin{array}{ll} (1) & A \longrightarrow B : \{m, A\}_{Pub(B)} \\ (2) & B \longrightarrow A : \{m, n, B\}_{Pub(A)} \\ (3) & A \longrightarrow B : \{n\}_{Pub(B)} \end{array}$$

This protocol, like many others of its kind has two roles: one for the initiator, here played by agent $A$ (say Alice), and one for the responder, here $B$ (Bob). It is a public-key protocol that assumes an underlying public-key infrastructure, such as RSA [RSA78]. Both agents have their own, secret private key. Public keys in contrast are available to all participants in the protocol. The NSL protocol makes use of *nonces* which one can think of as newly generated, unguessable numbers whose purpose is to ensure the freshness of messages.

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

Although in this informal explanation only two agents in their respective roles are described, the protocol is really a shorthand for a situation in which a network of distributed agents are each able participate in multiple concurrent sessions as both initiator and responder. There is no assurance that they all stick to the protocol, or indeed that communication goes to the intended agent. An attacker might dissemble and pretend to be one or several agents, taking advantage of any leaked keys it possesses in deciphering, and preparing the messages it sends.

The NSL protocol aims at distributing nonces $m$ and $n$ in a secure way, allowing no one but the initiator and the responder to know them (*secrecy*). Another aim of the protocol is that, for example, Bob should be guaranteed that $m$ is indeed the nonce sent by Alice (*authentication*). Lowe pointed out that the NSL protocol is prone to a "middle-man" attack, violating both these secrecy and authentication properties, if the name $B$ is not included in the second message [Low96]—the second message did not include $B$ in the original protocol of Needham and Schröder.

## 3 SPL—a language for security protocols

In order to be more explicit about the activities of participants in a protocol and those of a possible attacker, and to express these compositionally, we design an economical process language for the purpose. The language **SPL**(Security Protocol Language) is close to an asynchronous Pi-Calculus [Mil99] and is similar to that adopted in [AG97], though in its treatment of new names its transition semantics will be closer to that in [PS93] (it separates concerns of freshness from concerns of scope which are combined in the Pi-Calculus restriction).

### 3.1 The syntax of SPL

We start by giving the syntactic sets of the language:

- An infinite set of **Names**, with elements $n, m, \cdots, A, B, \cdots$. Names will range over nonces as well as agent names, and could also include other values.
- Variables over names $x, y, \cdots, X, Y, \cdots$.
- Variables over messages $\psi, \psi', \psi_1, \cdots$.
- Indices $i \in$ **Indices** with which to index components of parallel compositions.

| | | | |
|---|---|---|---|
| Name expressions | $v$ | $::=$ | $n, A, \cdots \mid x, X, \cdots$ |
| Key expressions | $k$ | $::=$ | $Pub(v) \mid Priv(v) \mid Key(v_1, v_2)$ |
| Messages | $M$ | $::=$ | $v \mid k \mid M_1, M_2 \mid \{M\}_k \mid \psi$ |
| Processes | $p$ | $::=$ | $out\ new\vec{x}M.p \mid$ |
| | | | $in\ pat\vec{x}\vec{\psi}M.p \mid$ |
| | | | $\Vert_{i\in I}p_i$ |

Figure 1: Syntax of **SPL**

The other syntactic sets of the language are described by the grammar shown in Figure 1. Note we use "vector" notation; for example, the "vector" $\vec{x}$ abbreviates some possibly empty list $x_1, \cdots, x_l$.

We take $fv(M)$, the free variables of a a message $M$, to be the set of variables which appear in $M$, and define the free variables of process terms by:

$$\begin{aligned} fv(out\ new\vec{x}M.p) &= (fv(p) \cup fv(M))\backslash\{\vec{x}\} \\ fv(in\ pat\vec{x}\vec{\psi}M.p) &= (fv(p) \cup fv(M))\backslash\{\vec{x}, \vec{\psi}\}) \\ fv(\Vert_{i\in I}p_i) &= \textstyle\bigcup_{i\in I} fv(p_i) \end{aligned}$$

As usual, we say that a process without free variables is closed, as is a message without variables. We shall use standard notation for substitution into the free variables of an expression, though we will only be concerned with the substitution of names or closed (variable-free) messages, obviating the problems of variable capture.

We use $Pub(v)$, $Priv(v)$ for the public, private keys of $v$, and we use $Key(v_1, v_2)$ for the symmetric key of $v_1$ and $v_2$. Keys can be used in building up encrypted messages. Messages may consist of a name or a key, be the composition of two messages $(M_1, M_2)$, or an encrypted message $\{M\}_k$ representing the message $M$ encrypted using the key $k$.

An informal explanation of the language:

$out\ new\vec{x}M.p$ This process chooses fresh, distinct names $\vec{n} = n_1, \cdots, n_l$ and binds them to the variables $\vec{x} = x_1, \cdots, x_l$. The message $M[\vec{n}/\vec{x}]$ is output to the network and the process resumes as $p[\vec{n}/\vec{x}]$. The communication is *asynchronous* in the sense that the action of output does not await input. The *new* construct is like that of Pitts and Stark [PS93] and abstracts out an important property of a value chosen randomly from some large set; such a value is likely to be new.

*in* $pat\vec{x}\vec{\psi}M.p$ This process awaits an input that matches the pattern $M$ for some binding of the pattern variables $\vec{x}\vec{\psi}$ and resumes as $p$ under this binding. All the pattern variables $\vec{x}\vec{\psi}$ must appear in the pattern $M$.

$\|_{i\in I}p_i$ This process is the parallel composition of all components $p_i$ for $i$ in the indexing set $I$. The set $I$ is a subset of **Indices**. Indices will help us distinguish in what agent, which role and what run a particular action occurs. The process, written *nil*, abbreviates the empty parallel composition (where the indexing set is empty).

**Convention 3.1** It simplifies the writing of process expressions if we adopt some conventions. Firstly, we simply write *out* $M.p$ when the list of "*new*" variables is empty. Secondly, and more significantly, we allow ourselves to write

$$\cdots in\ M.p \cdots$$

in an expression, to be understood as meaning the expression

$$\cdots in\ pat\vec{x}\vec{\psi}M.p \cdots$$

where the pattern variables $\vec{x}, \vec{\psi}$ are precisely those variables left free in $M$ by the surrounding expression. For example, we can describe a responder in NSL as the process

$$Resp(B)\ \equiv\ in\{x,Z\}_{Pub(B)}.\,out\ new\ y\{x,y,B\}_{Pub(Z)}.\,in\{y\}_{Pub(B)}.\,nil$$

For the first input, the variables $x, Z$ in $\{x,Z\}_{Pub(B)}$ are free in the whole expression, so by convention are pattern variables, and we could instead write *in pat* $x, Z\{x,Z\}_{Pub(B)}.\cdots$. On the other hand, in the second input the variable $y$ in $\{y\}_{Pub(B)}$ is bound by the outer *new* $y\ \cdots$ and so by the convention is not a pattern variable, and has to be that value sent out earlier. Often we will not write the *nil* process explicitly, so, for example, omitting its mention at the end of the responder code above. A parallel composition can be written in infix form via the notation

$$p_1\|p_2\cdots\|p_r \equiv \|_{i\in\{1,\cdots,r\}}p_i\ .$$

*Replication* of a process, $!p$, abbreviates $\|_{i\in\omega}p$, consisting of a countably infinite copies of $p$ set in parallel.

An obvious structural induction defines the set of names of a process. We define $size(p)$ of a process term $p$ to be an ordinal measuring the depth of process operations in the term.

$$
\begin{array}{lll}
Init(A,B) & \equiv & out\ new\ x\{x,A\}_{Pub(B)}\cdot \\
 & & in\{x,y,B\}_{Pub(A)}\cdot \\
 & & out\{y\}_{Pub(B)}\cdot \\
 & & nil \\
 & & \\
Resp(B) & \equiv & in\{x,Z\}_{Pub(B)}\cdot \\
 & & out\ new\ y\{x,y,B\}_{Pub(Z)}\cdot \\
 & & in\{y\}_{Pub(B)}\cdot \\
 & & nil
\end{array}
$$

Figure 2: Initiator and responder code

**Definition 3.1** The *size* of a closed process term is an ordinal given by the structural induction:

$$
\begin{array}{lll}
size(out\ new\vec{x}M.p) & = & 1 + size(p) \\
size(in\ pat\vec{x}\vec{\psi}M.p) & = & 1 + size(p) \\
size(\|_{i\in I}p_i) & = & 1 + sup_{i\in I}size(p_i).
\end{array}
$$

## 3.2 NSL as a process

As an illustration, we can program the NSL protocol in our language, and so formalise the introductory description given in the Section 2. We assume given a set of agent names, **Agents**, of agents participating in the protocol. The agents participating in the NSL protocol play two roles, as initiator and responder with any other agent. Abbreviate by $Init(A, B)$ the program of initiator $A \in$ **Agents** communicating with $B \in$ **Agents** and by $Resp(B)$ the program of responder $B \in$ **Agents**. The code of both an arbitrary initiator and an arbitrary responder is given in Figure 2. In the code we are forced to formalise aspects that are implicit in the informal description, such as the creation of new nonces, the decryption of messages and the matching of nonces.

We can model the attacker by directly programming it as a process. Figure 3, shows a general, active attacker or “spy”. The spy has the capability of composing eavesdropped messages, decomposing composite message, and using cryptography whenever the appropriate keys are available; the available keys are all the public keys and the leaked private keys. By choosing a different program for the spy we can restrict or augment 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 the

$$
\begin{array}{lcll}
\mathit{Spy}_1 & \equiv & \mathit{in}\ \psi_1.\mathit{in}\ \psi_2.\mathit{out}(\psi_1,\psi_2).\mathit{nil} & \text{(composing)} \\
\mathit{Spy}_2 & \equiv & \mathit{in}(\psi_1,\psi_2).\mathit{out}\ \psi_1.\mathit{out}\ \psi_2.\mathit{nil} & \text{(decomposing)} \\
\mathit{Spy}_3 & \equiv & \mathit{in}\ x.\mathit{in}\ \psi.\mathit{out}\ \{\psi\}_{\mathit{Pub}(x)}.\mathit{nil} & \text{(encryption)} \\
\mathit{Spy}_4 & \equiv & \mathit{in}\ \mathit{Priv}(x).\mathit{in}\ \{\psi\}_{\mathit{Pub}(x)}.\mathit{out}\ \psi.\mathit{nil} & \text{(decryption)} \\
 & & & \\
\mathit{Spy} & \equiv & \|_{i\in\{1,\ldots,4\}}\mathit{Spy}_i &
\end{array}
$$

Figure 3: Attacker code

$$
\begin{array}{lcl}
P_{\mathit{init}} & \equiv & \|_{A,B}\ !\ \mathit{Init}(A,B) \\
P_{\mathit{resp}} & \equiv & \|_{A}\ !\ \mathit{Resp}(A) \\
P_{\mathit{spy}} & \equiv & !\ \mathit{Spy} \\
 & & \\
\mathit{NSL} & \equiv & \|_{i\in\{\mathit{resp},\mathit{init},\mathit{spy}\}}\ P_i
\end{array}
$$

Figure 4: The system

protocol. The system is described in Figure 4.

## 3.3 A transition semantics

We first give a, fairly traditional, transition semantics to **SPL**. It says how input and output actions affect configurations; a configuration expresses the state of execution of the process, the messages so far output to the network and the names currently in use.

A *configuration* consists of a triple

$$\langle p, s, t\rangle$$

where $p$ is a closed process term, $s$ is a subset of the set of names **Names**, and $t$ is a subset of closed (i.e., variable-free) messages. We say the configuration is *proper* iff the names in $p$ and $t$ are included in $s$. The idea is that a closed process $p$ acts in the context of the set of names $s$ that have been used so far, and the set of messages $t$ which have been output, to input a message or to generate new names before outputting a message.

*Actions* $\alpha$ may be inputs or new-outputs, possibly tagged by indices to show at which parallel component they occur:

$$\alpha ::= \mathit{out}\ \mathit{new}\ \vec{n}.M \mid \mathit{in}\ M \mid i : \alpha$$

(output) Provided the names $\vec{n}$ are all distinct and not in $s$,

$$\langle out\ new\ \vec{x}M.p, s, t\rangle \xrightarrow{out\ new\ \vec{n}M[\vec{n}/\vec{x}]} \langle p[\vec{n}/\vec{x}], s \cup \{\vec{n}\}, t \cup \{M[\vec{n}/\vec{x}]\}\rangle$$

(input) Provided $M[\vec{n}/\vec{x}, \vec{N}/\vec{\psi}] \in t$,

$$\langle in\ pat\vec{x}\vec{\psi}M.p, s, t\rangle \xrightarrow{in\ M[\vec{n}/\vec{x},\vec{N}/\vec{\psi}]} \langle p[\vec{n}/\vec{x}, \vec{N}/\vec{\psi}], s, t\rangle$$

(par)

$$\frac{\langle p_j, s, t\rangle \xrightarrow{\alpha} \langle p'_j, s', t'\rangle}{\langle \|_{i\in I} p_i, s, t\rangle \xrightarrow{j:\alpha} \langle \|_{i\in I} p_i[p'_j/j], s', t'\rangle}\ j \in I$$

Figure 5: Transition semantics

where $M$ is a closed message, $\vec{n}$ are names and $i$ is an index drawn from **Indices**. We write *out* $M$ for an output action, outputting a message $M$, where no new names are generated.

The way configurations evolve is expressed by transitions

$$\langle p, s, t\rangle \xrightarrow{\alpha} \langle p', s', t'\rangle\ ,$$

given by the rules displayed in Figure 5.

The transition semantics allows us to state formally many security properties. However, it does not support directly local reasoning of the kind one might wish to apply in the analysis of security protocols. To give an idea of the difficulty, imagine we wished to establish that the nonce generated by $B$ as responder in NSL was never revealed as an open message on the network. A reasonable way to prove such a property is to find a stronger invariant, a property which can be shown to be preserved by all the actions of the process. Equivalently, one can assume that there is an earliest action $\alpha_l$ in a run which violates the invariant, and derive a contradiction by showing that this action must depend on a previous action, which itself violates the invariant.

An action might depend on another action through being, for example, an input depending on a previous output, or simply through occurring at a later control point in a process. A problem with the transition semantics is that it masks such local dependency, and even the underlying process events on which the dependency rests. The wish to support arguments based on local dependency leads to a more refined semantics based on events.

## 4 The events of SPL

We must first address the issue of what constitutes an event of a security protocol. Here, we follow the lead from Petri nets,[2] and define events in terms of how they affect conditions. Conditions are to represent some form of local state and we discern conditions of three kinds: *control*, *output* and *name* conditions.

The set of *control conditions* **C** consists of output or input processes, perhaps tagged by indices, and is given by the grammar

$$b ::= out\ new\ \vec{x}M.p \mid in\ pat\vec{x}\vec{\psi}M.p \mid i : b$$

where $i \in$ **Indices**. A condition in **C** stands for the point of control in a (single-thread) process. When $C$ is a subset of control conditions we will write $i : C$ to mean $\{i : b \mid b \in C\}$.

The set of *output conditions* **O** consists of closed message expressions. An individual condition $M$ in **O** stands for the message $M$ having been output on the network. Output conditions are *persistent*; once output conditions are made to hold they continue to hold forever. This squares with our understanding that once a message has been output to the network it can never be removed, and can be input repeatedly.

The set of *name conditions* is precisely the set of names **Names**. A condition $n$ in **Names** stands for the name $n$ being in use.

We define the *initial conditions* of a closed process term $p$, to be the subset $Ic(p)$ of **C**, given by the following structural induction:

$$Ic(out\ new\ \vec{x}M.p) = \{out\ new\ \vec{x}M.p\}$$
$$Ic(in\ pat\vec{x}\vec{\psi}M.p) = \{in\ pat\vec{x}\vec{\psi}M.p\}$$
$$Ic(\|_{i\in I}p_i) = \bigcup_{i\in I} i : Ic(p_i)$$

where the last case also includes the base case *nil*, when the indexing set is empty.

[2]A brief summary of Petri nets is given in the Appendix.

We will shortly define the set of *events* **Events** as a subset of

$$\mathcal{P}ow(\mathbf{C})\times\mathcal{P}ow(\mathbf{O})\times\mathcal{P}ow(\mathbf{Names})\times\mathcal{P}ow(\mathbf{C})\times\mathcal{P}ow(\mathbf{O})\times\mathcal{P}ow(\mathbf{Names})\,.$$

So an individual event $e \in$ **Events** is a tuple

$$e = ({}^ce, {}^oe, {}^ne, e^c, e^o, e^n)$$

where ${}^ce$ is the set of **C**-preconditions of $e$, $e^c$ is the set of **C**-postconditions of $e$, etc. Write ${}^\cdot e$ for ${}^ce \cup {}^oe \cup {}^ne$, all preconditions of $e$, and $e^\cdot$ for all postconditions $e^c \cup e^o \cup e^n$.

Earlier in the transition semantics we used actions $\alpha$ to specify the nature of transitions. An event $e$ is associated with a unique action $act(e)$.

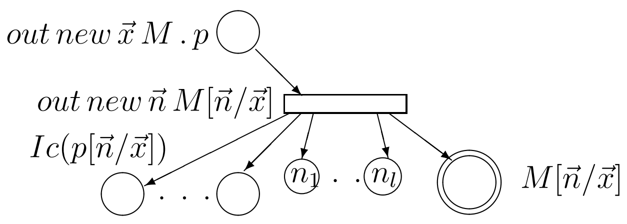
The set of events associated with **SPL** is given by an inductive definition. Define **Events** to be the smallest set which includes all output, input and indexed events:

- *output events* $\mathbf{Out}(out\ new\ \vec{x}M.p;\vec{n})$, where $\vec{n} = n_1, \cdots, n_l$ are *distinct* names to match the variables $\vec{x} = x_1, \cdots, x_l$, consists of an event $e$ with these pre- and postconditions:

  $$\begin{array}{l} {}^ce = \{out\ new\ \vec{x}M.p\}\ , \quad {}^oe = \emptyset\ , \quad {}^ne = \emptyset\ , \\ e^c = Ic(p[\vec{n}/\vec{x}])\ , \quad e^o = \{M[\vec{n}/\vec{x}]\} \quad e^n = \{n_1, \cdots, n_l\}\ . \end{array}$$

  The *action* of an output event is

  $$act(\mathbf{Out}(out\ new\ \vec{x}M.p;\vec{n})) = out\ new\ \vec{n}.M[\vec{n}/\vec{x}].$$

  
  $out\,new\,\vec{x}\,M\,.\,p$
  $out\,new\,\vec{n}\,M[\vec{n}/\vec{x}]$
  $Ic(p[\vec{n}/\vec{x}])$
  $n_1$
  $n_l$
  $M[\vec{n}/\vec{x}]$

  An occurrence of the output event $\mathbf{Out}(out\ new\ \vec{x}M.p;\vec{n})$ affects the control conditions and puts the new names $n_1, \cdots, n_l$ into use, necessarily for the first time as according to the token game the event occurrence must avoid contact with names already in use.

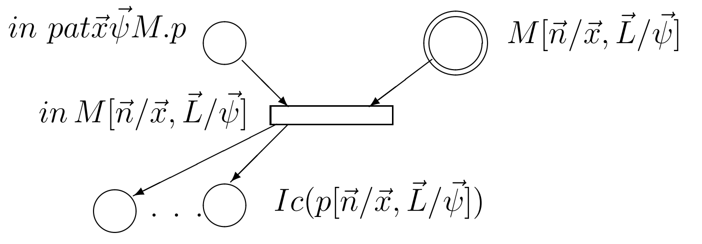
  The definition includes the special case when $\vec{x}$ and $\vec{n}$ are empty lists, and we write $\mathbf{Out}(out\ M.p)$ for the output event with no name conditions and action $out\ M$.

- *input events* $\mathbf{In}(in\ pat\vec{x}\vec{\psi}M.p; \vec{n}, \vec{L})$, where $\vec{n}$ is a list of names to match $\vec{x}$ and $\vec{L}$ is a list of closed messages to match $\vec{\psi}$, consists of an event $e$ with these pre- and postconditions:

  $$\begin{array}{l} {}^{c}e = \{in\ pat\vec{x}\vec{\psi}M.p\}\ ,\quad {}^{o}e = \{M[\vec{n}/\vec{x}, \vec{L}/\vec{\psi}]\}\ ,\quad {}^{n}e = \emptyset\ , \\ e^{c} = Ic(p[\vec{n}/\vec{x}, \vec{L}/\vec{\psi}])\ ,\quad e^{o} = \emptyset\ ,\quad e^{n} = \emptyset\ . \end{array}$$

  The action of an input event is

  $$act(\mathbf{In}(in\ pat\vec{x}\vec{\psi}M.p; \vec{n}, \vec{L})) = in\ M[\vec{n}/\vec{x}, \vec{L}/\vec{\psi}].$$


$in\ pat\vec{x}\vec{\psi}M.p$
$M[\vec{n}/\vec{x}, \vec{L}/\vec{\psi}]$
$in\,M[\vec{n}/\vec{x}, \vec{L}/\vec{\psi}]$
. . .
$Ic(p[\vec{n}/\vec{x}, \vec{L}/\vec{\psi}])$

- *indexed events* $i : e$ where $e \in \mathbf{Events}$, where $i \in \mathbf{Indices}$ and

  $$\begin{array}{l} {}^{c}(i:e) = i :{}^{c}e\ ,\quad {}^{o}(i:e) ={}^{o}e\ ,\quad {}^{n}(i:e) ={}^{n}e\ , \\ (i:e)^{c} = i : e^{c}\ ,\quad (i:e)^{o} = e^{o}\ ,\quad (i:e)^{n} = e^{n}\ . \end{array}$$

  The action of an indexed event $act(i : e)$ is $i : \alpha$, where $\alpha$ is the action of $e$.

  When $E$ is a subset of events we will generally use $i : E$ to mean $\{i : e \mid e \in E\}$.

In defining the set of conditions and, inductively, the set of events, we have in fact defined a (rather large) net from the syntax of **SPL**. The **SPL**-net has conditions $\mathbf{C} \cup \mathbf{O} \cup \mathbf{Names}$ and events **Events**. Its markings $\mathcal{M}$ will be subsets of conditions and so of the form

$$\mathcal{M} = c \cup s \cup t$$

where $c \subseteq \mathbf{C}$, $s \subseteq \mathbf{Names}$, and $t \subseteq \mathbf{O}$. By assumption the set of conditions **O** are persistent so the net is a contextual net with the following token game—see Appendix C.

Letting $c \cup s \cup t$ and $c' \cup s' \cup t'$ be two markings, $c \cup s \cup t \xrightarrow{e} c' \cup s' \cup t'$ iff

$${}^{\cdot}e \subseteq c \cup s \cup t\ \&\ e^{c} \cap c = \emptyset\ \&\ e^{n} \cap s = \emptyset \text{ (event } e \text{ has concession), and}$$
$$c' = (c \setminus {}^{c}e) \cup e^{c}\ \&\ s' = s \cup e^{n}\ \&\ t' = t \cup e^{o}\ .$$

In particular, the occurrence of $e$ begins the holding of its name postconditions $e^n$—these names have to be distinct from those already in use to avoid contact.

## 5 Relating net and transition semantics

The behaviour of the **SPL**-net is closely related to the transition semantics given earlier.

**Theorem 5.1**

*i) If $\langle p, s, t\rangle \xrightarrow{\alpha} \langle p', s', t'\rangle$, then $Ic(p) \cup s \cup t \xrightarrow{e} Ic(p') \cup s' \cup t'$ in the* **SPL***-net, for some $e \in$* **Events** *with $act(e) = \alpha$.*

*ii) If $Ic(p) \cup s \cup t \xrightarrow{e} \mathcal{M}'$ in the* **SPL***-net, then $\langle p, s, t\rangle \xrightarrow{act(e)} \langle p', s', t'\rangle$ and $\mathcal{M}' = Ic(p') \cup s' \cup t'$, for some closed process $p'$, $s' \subseteq$* **Names** *and $t' \subseteq$* **O***.*

**Definition 5.1** Let $e \in$ **Events**. Let $p$ be a closed process, $s \subseteq$ **Names**, and $t \subseteq$ **O**. Write $\langle p,\ s,\ t\rangle \xrightarrow{e} \langle p',\ s',\ t'\rangle$ iff $Ic(p) \cup s \cup t \xrightarrow{e} Ic(p') \cup s' \cup t'$ in the **SPL**-net.

## 6 The events of a process

Generally for a process $p$ only a small subset of the events **Events** can ever come into play. For this reason it's useful to restrict the events to those reachable in the behaviour of a process.

The set $Ev(p)$ of events of a closed process term $p$ are defined by induction on size:

$$\begin{aligned}
Ev(out\ new\ \vec{x}M.p) =& \{\mathbf{Out}(out\ new\ \vec{x}M.p; \vec{n}) \mid \vec{n} \text{ distinct names}\} \\
& \cup \bigcup \{Ev(p[\vec{n}/\vec{x}]) \mid \vec{n} \text{ distinct names}\}\ , \\
Ev(in\ pat \vec{x}\vec{\psi}M.p) =& \{\mathbf{In}(in\ pat\vec{x}\vec{\psi}M.p; \vec{n}, \vec{L}) \mid \vec{n} \text{ names}, \vec{L} \text{ closed messages}\} \\
& \cup \bigcup \{Ev(p[\vec{n}/\vec{x}, \vec{L}/\vec{\psi}]) \mid \vec{n} \text{ names}, \vec{L} \text{ closed messages}\}\ , \\
Ev(\|_{i\in I} p_i) =& \bigcup_{i \in I} i : Ev(p_i)\ .
\end{aligned}$$

As an example, the events $Ev(NSL)$ of NSL are shown in the Appendix.

A closed process term $p$ denotes a net $Net(p)$ consisting of the global set of conditions $\mathbf{C} \cup \mathbf{O} \cup \mathbf{Names}$ built from **SPL**, events $Ev(p)$ and initial control conditions $Ic(p)$. We can define the token game on the net $Net(p)$ exactly as we did earlier for the **SPL**-net, but this time events are restricted to being in the set $Ev(p)$. It's clear that if an event transition is possible in the restricted net $Net(p)$ then so is it in the **SPL**-net. The converse also holds provided one starts from a marking whose control conditions either belong to $Ic(p)$ or are conditions of events in $Ev(p)$.

**Definition 6.1** Let $p$ be a closed process term. Define the *control-conditions of* $p$ to be

$$p^c = Ic(p) \cup \bigcup \{ {}^c e \mid e \in Ev(p)\} \ .$$

**Lemma 6.1** *Let $\mathcal{M} \cap \mathbf{C} \subseteq p^c$. Let $e \in$ **Events**. Then,*

$$\mathcal{M} \xrightarrow{e} \mathcal{M}' \text{ in the } \mathbf{SPL}\text{-net iff } e \in Ev(p) \ \& \ \mathcal{M} \xrightarrow{e} \mathcal{M}' \text{ in } Net(p) \ .$$

Consequently, in analysing those sequences of event transitions

$$\langle p_0, s_0, t_0 \rangle \xrightarrow{e_1} \cdots \xrightarrow{e_r} \langle p_r, s_r, t_r \rangle \xrightarrow{e_{r+1}} \cdots \ ,$$

a closed process $p$ can perform, or correspondingly those of the transition semantics, it suffices to study the behaviour of $Net(p)$ with its restricted set of events $Ev(p)$. This simplification is especially useful in proving invariance properties because these amount to an argument by cases on the form of events a process can do.

Recall that we say a configuration $\langle p, s, t \rangle$ is *proper* iff the names in $p$ and $t$ are included in $s$.

**Proposition 6.2** *Let $e \in$ **Events**. Suppose that $\langle p, s, t \rangle$ and $\langle p', s', t' \rangle$ are configurations, and that $\langle p, s, t \rangle$ is proper. If $\langle p, s, t \rangle \xrightarrow{e} \langle p', s', t' \rangle$, then $\langle p', s', t' \rangle$ is also proper.*

**Important convention:** From now on we assume that all configurations $\langle p, s, t \rangle$ are proper.

# 7 Proving security properties

To demonstrate the viability of the net semantics as a tool in proving security properties, we use the semantics to derive general principles for proving secrecy and authentication. The principles capture the kind of dependency reasoning found in the strand spaces and inductive methods. To illustrate the principles in action, we apply them to establish an authentication guarantee for the responder part of the NSL protocol. We introduce a diagrammatic style of reasoning which we find helpful.

## 7.1 General proof principles

From the net semantics we can derive several principles useful in proving authentication and secrecy of security protocols. Write $M \sqsubset M'$ to mean message $M$ in a subexpression of message $M'$, i.e., $\sqsubset$ is the smallest binary relation on messages such that:

$$\begin{array}{l} M \sqsubset M \\ M \sqsubset N \ \Rightarrow \ M \sqsubset (N, N') \wedge M \sqsubset (N', N) \\ M \sqsubset N \ \Rightarrow \ M \sqsubset \{N\}_k \end{array}$$

where $M, N, N'$ are messages and $k$ is a key expression. We also write $M \sqsubset t$ iff $\exists M' \,.\, M \sqsubset M' \wedge M' \in t$, for a set of messages $t$.

**Proposition 7.1** *(Well-foundedness) Given a property $\mathcal{P}$ on configurations, if a run*

$$\langle p_0, s_0, t_0 \rangle \xrightarrow{e_1} \cdots \xrightarrow{e_r} \langle p_r, s_r, t_r \rangle \xrightarrow{e_{r+1}} \cdots ,$$

*contains a configurations such that $\mathcal{P}(p_0, s_0, t_0)$ and $\neg\mathcal{P}(p_j, s_j, t_j)$, then there is an event $e_h$, $0 < h \leq j$, such that $\mathcal{P}(p_i, s_i, t_i)$ for all $i \leq h$ and $\neg\mathcal{P}(p_h, s_h, t_h)$.*

We say that a name $m \in$ **Names** is *fresh on an event* $e$ if $m \in e^n$ and we write $Fresh(m, e)$.

**Proposition 7.2** *(Freshness) Within a run*

$$\langle p_0, s_0, t_0 \rangle \xrightarrow{e_1} \cdots \xrightarrow{e_r} \langle p_r, s_r, t_r \rangle \xrightarrow{e_{r+1}} \cdots ,$$

*the following properties hold:*

1. *If $n \in s_i$ then either $n \in s_0$ or there is a previous event $e_j$ such that $Fresh(n, e_j)$.*

2. *Given a name $n$ there exists at most one event $e_i$ s.t. $Fresh(n, e_i)$.*
3. *If $Fresh(n, e_i)$ then for all $j < i$ the name $n$ does not appear in $\langle p_j, s_j, t_j\rangle$.*

**Proposition 7.3** *(Control precedence) Within a run*

$$\langle p_0, s_0, t_0\rangle \xrightarrow{e_1} \cdots \xrightarrow{e_r} \langle p_r, s_r, t_r\rangle \xrightarrow{e_{r+1}} \cdots ,$$

*if $b \in {}^c e_i$ either $b \in Ic(p_0)$ or there is an earlier event $e_j$, $j < i$, such that $b \in {e_j}^c$.*

**Proposition 7.4** *(Output-input precedence) In a run*

$$\langle p_0, s_0, t_0\rangle \xrightarrow{e_1} \cdots \xrightarrow{e_r} \langle p_r, s_r, t_r\rangle \xrightarrow{e_{r+1}} \cdots ,$$

*if $M \in {}^o e_i$, then either $M \in t_0$ or there is an earlier event $e_j$, $j < i$, such that $M \in {e_j}^o$.*

## 7.2 An example: authentication for NSL

We will prove authentication for a responder in an NSL protocol in the sense that: to any complete session of agent $B_0$ as responder, apparently with agent $A_0$, there corresponds a complete session of agent $A_0$ as initiator. We refer to the Appendix for the events of NSL.

In the proof it's helpful to make use of a form of diagrammatic reasoning which captures the precedence of events. When the run is understood we draw $e \longrightarrow e'$ when $e$ precedes $e'$ in the run, allowing $e = e'$.

**Theorem 7.5 (Authentication)** *If a run of NSL*

$$\langle NSL, s_0, t_0\rangle \xrightarrow{e_1} \cdots \xrightarrow{e_r} \langle p_r, s_r, t_r\rangle \xrightarrow{e_{r+1}} \cdots ,$$

*contains the responder events $b_1, b_2, b_3$, with actions*

$$\begin{aligned} act(b_1) &= resp : B_0 : i : in\,\{m_0, A_0\}_{Pub(B_0)} \ , \\ act(b_2) &= resp : B_0 : i : out\,new\,n_0\,\{m_0, n_0, B_0\}_{Pub(A_0)} \ , \\ act(b_3) &= resp : B_0 : i : in\{n_0\}_{Pub(B_0)}) \ , \end{aligned}$$

*for an index $i$, and $Priv(A_0) \not\sqsubseteq t_0$, then the run contains initiator events $a_1, a_2, a_3$ with $a_3 \longrightarrow b_3$ , where, for some index $j$,*

$$\begin{aligned} act(a_1) &= init : (A_0, B_0) : j : out\,new\,m_0\,\{m_0, A_0\}_{Pub(B_0)} \ , \\ act(a_2) &= init : (A_0, B_0) : j : in\{m_0, n_0, B_0\}_{Pub(A_0)} \ , \\ act(a_3) &= init : (A_0, B_0) : j : out\{n_0\}_{Pub(B_0)} \ . \end{aligned}$$

*Proof.* By control precedence we obtain: $b_1 \longrightarrow b_2 \longrightarrow b_3$ .

Consider the property of configurations

$$Q(p,s,t) \Leftrightarrow \forall M \in t.\ n_0 \sqsubset M \Rightarrow \{m_0, n_0, B_0\}_{Pub(A_0)} \sqsubset M\ .$$

By freshness, the property $Q$ holds immediately after $b_2$, but clearly not immediately before $b_3$. By well-foundedness there is a earliest event following $b_2$ but preceding $b_3$ that violates $Q$. Let $e$ be such an event.

$$\begin{tikzcd}
b_1 \arrow[r] & b_2 \arrow[rr] \arrow[dr] & & b_3 \\
& & e \arrow[ur] &
\end{tikzcd}$$

Inspecting the events of the $NSL$ protocol (see Appendix), using the assumption that $Priv(A_0) \not\sqsubset t_0$, one can show that $e$ can only be an initiator event $a'_3$ with action

$$act(a'_3) = init : (A, B_0) : j : out\{n_0\}_{Pub(B_0)}$$

for some index $j$ and agent $A$. There must also be preceding events $a'_1, a'_2$ with actions

$$\begin{array}{rcl} act(a'_1) & = & init : (A, B_0) : j : out\,new\,m\,\{m, A\}_{Pub(B_0)} \\ act(a'_2) & = & init : (A, B_0) : j : in\{m, n_0, B_0\}_{Pub(A)} \end{array}$$

$$\begin{tikzcd}
b_1 \arrow[r] & b_2 \arrow[rr] \arrow[dr] & & b_3 \\
a'_1 \arrow[r] & a'_2 \arrow[r] & a'_3 \arrow[ur] &
\end{tikzcd}$$

Since $Fresh(b_2, n_0)$, the event $b_2$ must precede $a'_2$. The property $Q$ holds on configurations up to $a'_3$ and, in particular, on the configuration immediately before $a'_2$. From this we conclude that $m = m_0$ and $A = A_0$. Hence $a'_3 = a_3$, $a'_2 = a_2$, and $a'_1 = a_1$ as described below.

$$\begin{tikzcd}
b_1 \arrow[r] & b_2 \arrow[r] \arrow[d] & b_3 \\
a_1 \arrow[r] & a_2 \arrow[r] & a_3 \arrow[u]
\end{tikzcd}$$

(Since $Fresh(a_1, m_0)$, the event $a_1$ precedes $b_1$.) □

## 8 Relating security models

We have related our net semantics of **SPL** to a transition semantics. Now we establish its relations to the security models of strand spaces, inductive rules, as well as other traditional models. In security protocols we are largely interested in safety properties, which reduce to a property holding of all finite behaviours. Thus it suffices to show how a finite behaviour in one model can be matched by the finite behaviour in another. In relating the net semantics to strand spaces and inductive rules we need to constrain process terms, to allow some repetition of actions, though this does not seem unduly restrictive in formalising security protocols.

### 8.1 Strand spaces

In relating the net semantics to strand spaces we must face the fact that strand spaces don't compose readily, not using traditional process operations at least. Their form doesn't allow prefixing by a single event. Nondeterminism only arises through the choice as to where input comes from, and there is not a recognisable nondeterministic sum of strand spaces. Even an easy definition of parallel composition by juxtaposition is thwarted if "unique origination" is handled as a global condition on the entire strand space. This complicates the relation between a compositional semantics and strand spaces.

We can however relate the net behaviour of a *!-par* process to to that of an associated strand space; a *!-par* process is a closed process of the form $!\|_{i\in I} p_i$ for which no subterm $p_i$ contains a parallel composition. In proving the relation (though unfortunately not in this short write-up) we find it useful to extend strand spaces in order to compose them, chiefly with conflict to permit their nondeterministic sum, and then finally to observe that for processes with replication the conflict can be eliminated, without upsetting the strand-space behaviour. (Strand spaces can be viewed as special forms of event structures—see below; so ideas, such as the use of a conflict relation, can be adapted from there.)

**Definition 8.1** A strand space consists of $S = \langle S_i \rangle_{i\in I}$ an indexed set of strands. An individual strand $S_i$, where $i \in I$, is a finite sequence of output or input events carrying respectively output or input actions of the kind $out\, new \vec{n} M$ or $in\, M$, where $M$ is a closed message and $\vec{n}$ a list of distinct names that are intended to be fresh ("uniquely originating") at the event. We permit only strands on which any "*new*" names do appear in previous actions of the strand. (A set of strands is canonically a strand space in which each strand has itself as index.)

As usual, a strand space can be seen as a graph whose nodes are of the form $(i, l)$ with $i \in I$ index of a strand and $l$ position of an event in that strand ($1 \leq l \leq \mathit{length}(s_i)$). Each node uniquely identifies an event in a strand. Edges are of two different kinds: $\Rightarrow$ between two nodes that identify two events of a same strand, one immediately preceding the other and $\rightarrow$ between two nodes identifying respectively an output event and an input event with the same message. A bundle of a strand space $S$ is a finite, acyclic subgraph such that

- if a node belongs to the bundle then so do all the nodes that precede it on its strand,
- each input node has exactly one incoming $\rightarrow$ edge,
- two different strands that have a "*new*" name in common don't both contribute to the same bundle.

Our definition is not quite standard. But the only significant difference is in the treatment of unique origination which is taken care of in the definition of bundle rather than being a condition on the entire strand space—the "parametric strand spaces" of [CDL$^+$00] achieve the same effect and are closely related.

A strand space can be seen as a form of *event structure* [Win87a]. A strand space determines a *stable event structure*, whose family of configurations is the same as the bundles of the strand space; the bundles of a strand space when ordered by inclusion form a *stable family* which ensures not only that each configuration of events in the family can be equipped with a local partial order of causal dependency, but that at the cost renaming events these local partial orders can be extended to a global partial order of causal dependency, yielding a *prime event structure.*

Often in strand spaces the precise identity of indices doesn't matter. A *re-indexing* of a strand space $S = \langle S_i \rangle_{i \in I}$ is a permutation $\pi$ of $I$ such that $S_i$ and $S_{\pi(i)}$ are sequences of the same length with the same actions at corresponding events. A re-indexing of a strand space induces a re-indexing on its bundles; a bundle's nodes and arcs are changed according to the correspondence given by $\pi$.

To relate the net behaviour of a process to its behaviour as a strand space we need to linearise bundles. More precisely:

**Definition 8.2** Given a bundle $\mathcal{C}$ of a strand space $S$, a *linearisation* of $\mathcal{C}$ is a sequence of nodes $e_1 \dots e_k$ such that $\{e_1, \dots, e_k\} = \mathcal{C}$ and for all $e$ in $\mathcal{C}$ and for all $e_i$ in $L$, if $e \Rightarrow_{\mathcal{C}} e_i$ or $e \rightarrow_{\mathcal{C}} e_i$ then $e$ precedes $e_i$ in the sequence. An *event-linearisation* of a bundle is the sequence of strand-space events associated with the nodes of a linearisation.

Let $p$ be a *!-par* process and $s$ a set of names containing all names in $p$. Take $Tr(p, s)$ to be the strand space with strands consisting of all the maximal sequences $e_1 \dots e_k$ of events in $Ev(p)$ such that:

*i)* ${}^{c}e_1 \subseteq Ic(p)$ and

*ii)* for all $i$, $1 \leq i < k$, we have ${e_i}^{n} \cap (s \cup \{{e_j}^{n} \mid j < i\}) = \emptyset$ and ${e_i}^{c} = {}^{c}e_{i+1}$.

Sequences satisfying the above conditions are necessarily finite as the size of control conditions strictly decreases along the sequence. The events of the net are already associated with input and output actions. The net and strand space behaviour are closely related:

**Theorem 8.3** *Given p a !-par process and s set of names containing all names in p, we have that:*

1. *The sequence of events in a finite run in $Net(p)$ from the initial configuration $\langle p, s, \emptyset \rangle$ is an event-linearisation of a bundle over $Tr(p, s)$.*
2. *Every bundle over $Tr(p, s)$ can be re-indexed so that any of its event-linearisations is a run in $Net(p)$.*

The only way a strand space can cope with there being a nonempty set of initial output messages is through the slight clumsiness of introducing extra output events; we avoid this above by assuming the initial set of output messages is empty.

## 8.2 Inductive rules

Paulson's inductive rules for a security protocol capture the actions it and a spy can perform [Pau98]. Through allowing persistent conditions, we can represent a collection of inductive rules as a net in which the events stand for rule instances and runs to sequences of rule instances which form a derivation from the rules. In particular, instances of inductive rules for security protocols can be represented as events in a net for which all but the name conditions are persistent. According to such a semantics, once a protocol can input it can do so repeatedly. Once it can output generating new names it can do so repeatedly, provided this doesn't lead to clashes with names already in use. Paulson's traces and the associated runs of the net will necessarily include such "stuttering."

We define a net of rule instances from a closed process term. Take the set of "rule-conditions" to consist of name conditions and persistent output conditions, as before, but now with additional persistent conditions consisting of closed input and output process terms. Let $r$ be the function from **SPL**-conditions to rule-conditions which removes the indices tagging control conditions and leaves output and name conditions unchanged. Extend $r$ to **SPL**-events: let $r$ replace all the control conditions of an **SPL**-event by their images under $r$—intuitively, an event is replaced by a rule instance. Define the "net of rule instances" $R(p)$ of a closed process term $p$ to be the net with rule-conditions and events the image $r\,Ev(p)$.

For a closed process term $p$, let $p^*$ be the process term obtained by inserting a replication before every input and output process subterm in $p$. Note that $R(p^*) = R(p)$ as $R$ drops indices. Now, having restricted to a process with sufficient replication, we can establish a close relation between the behaviours of $Net(p^*)$ and $R(p^*)$.

**Theorem 8.4** *Let $p$ be a closed process term. Let $t$ be a subset of closed messages and $s$ a subset of names including those of $p$ and $t$. Let $\mathcal{M}_0 = Ic(p^*) \cup s \cup t$.*

1. *A run $\mathcal{M}_0 \xrightarrow{e_1} \cdots \xrightarrow{e_l} \mathcal{M}_l$ of $Net(p^*)$ yields $r\mathcal{M}_0 \xrightarrow{r(e_1)} \cdots \xrightarrow{r(e_l)} r\mathcal{M}_l$ a run of $R(p^*)$.*

2. *If $\mathcal{M}'_0 \xrightarrow{e'_1} \cdots \xrightarrow{e'_l} \mathcal{M}'_l$ is a run of $R(p^*)$ with $\mathcal{M}'_0 = r\mathcal{M}_0$, then there is a run $\mathcal{M}_0 \xrightarrow{e_1} \cdots \xrightarrow{e_l} \mathcal{M}_l$ of $Net(p^*)$, with $r(e_i) = e'_i$ and $r(\mathcal{M}_i) = \mathcal{M}'_i$ for all $i$, $0 < i \leq l$.*

## 8.3 Basic nets, trace languages and event structures

Because strand spaces can be easily turned into event structures, Section 8.1 yields an event structure for each *!-par* process. But, without any restrictions, we can relate the net semantics to traditional independence models such as event structures and Mazurkiewicz trace languages. The crux of the construction is that of eliminating the persistent conditions from the net $Net(p)$, of a closed process term $p$, in an initial marking $Init(p) \cup s \cup t$, to produce a basic net. It's well-known how to "unfold" a basic net to a Mazurkiewicz trace language and event structure. Assume $Net(p)$ has input events $In$ and output events $Out$. Then:

**Theorem 8.5** *There is a basic net $\mathcal{N}$ with events*

$$\begin{array}{rcl} E & = & Out \\ & & \cup\ \{(*, e) \mid e \in In \ \&\ {}^{o}e \subseteq t\} \\ & & \cup\ \{(e_1, e) \mid e \in In \ \&\ e_1 \in Out \ \&\ \ {}^{o}e = e_1^{o}\} \end{array}$$

*Let the map $\sigma : E \to Out \cup In$ leave events in $Out$ unchanged and project pairs $(*, e)$, $(e_1, e)$ to the component $e$.*

*i) If $\mathcal{N}$ has a run with events $e'_1, \cdots, e'_k$, then there is a run with events $\sigma(e'_1), \cdots, \sigma(e'_k)$ of $Net(p)$ from the initial marking $Init(p) \cup s \cup t$.*

*ii) If $Net(p)$ has a run $e_1, \cdots, e_k$ from the initial marking $Init(p) \cup s \cup t$, then $\mathcal{N}$ has a run $e'_1, \cdots, e'_k$ where $e_1, \cdots, e_k = \sigma(e'_1), \cdots, \sigma(e'_k)$.*

The construction used to obtain $\mathcal{N}$ above is an example of the construction for eliminating colours from a coloured net—see [Win87b]; first colours are introduced to the persistent conditions and input events of $Net(p)$ to distinguish the different ways in which they are made to occur, and then eliminated through splitting the conditions and events according to their colours. The result in this case is a basic net. Its runs form a Mazurkiewicz trace language from which we can then obtain an event structure–see [WN95] for details.

# Appendix

## A Petri nets

The explanation of general Petri nets involves a little algebra of multisets (or bags), which are like sets but where multiplicities of elements matters. It's convenient to also allow infinite multiplicities, so we adjoin an extra element $\infty$ to the natural numbers, though care must be taken to avoid subtracting $\infty$. $\infty$-Multisets support addition $+$ and multiset inclusion $\leq$, and even multiset subtraction $X - Y$ provided $Y \leq X$ and $Y$ has no infinite multiplicities, in which case we call $Y$ simply a multiset.

## A.1 General Petri nets

A *general Petri net* (often called a *place-transition system*) consists of

- a set of *conditions* (or *places*), $P$,
- a set of *events* (or *transitions*), $T$,
- a *precondition map pre*, which to each $t \in T$ assigns a multiset $pre(t)$ over $P$. It is traditional to write ${}^{\cdot}t$ for $pre(t)$.
- a *postcondition* map *post* which to each $t \in T$ assigns an $\infty$-multiset $post(t)$ over $P$, traditionally written $t^{\cdot}$.
- a *capacity* function $Cap$ which is an $\infty$-multiset over $P$, assigning a nonnegative number or $\infty$ to each condition $p$, bounding the multiplicity to which the condition can hold; a capacity of $\infty$ means the capacity is unbounded.

A state of a Petri net consists of a *marking* which is an $\infty$-multiset $\mathcal{M}$ over $P$ bounded by the capacity function, *i.e.*

$$\mathcal{M} \leq Cap \ .$$

A marking captures a notion of distributed, global state.
**Token game for general nets:** Markings can change as events occur, precisely how being expressed by the transitions

$$\mathcal{M} \xrightarrow{t} \mathcal{M}'$$

events $t$ determine between markings $\mathcal{M}$ and $\mathcal{M}'$. For markings $\mathcal{M}$, $\mathcal{M}'$ and $t \in T$, define

$$\mathcal{M} \xrightarrow{t} \mathcal{M}' \text{ iff } {}^{\cdot}t \leq \mathcal{M} \text{ and } \mathcal{M}' = \mathcal{M} - {}^{\cdot}t + t^{\cdot} \ .$$

An event $t$ is said to have *concession* (or be *enabled*) at a marking $\mathcal{M}$ iff its occurrence would lead to a marking, *i.e.*iff

$${}^{\cdot}t \leq \mathcal{M} \text{ and } \mathcal{M} - {}^{\cdot}t + t^{\cdot} \leq Cap \ .$$

There is a widely-used graphical notation for nets in which events are represented by squares, conditions by circles and the pre- and postcondition maps by directed arcs carrying numbers or $\infty$. A marking is represented by the presence of tokens on a condition, the number of tokens representing the multiplicity to which the condition holds. When an event with concession occurs tokens are removed from its preconditions and put on its postconditions with multiplicities according to the pre- and postcondition maps. Because of this presentation, the transition relation on Petri nets is described as the "token game".

## B Basic nets

We instantiate the definition of general Petri nets to an important case where in all the multisets the multiplicities are either 0 or 1, and so can be regarded as sets. In particular, we take the capacity function to assign 1 to every condition, so that markings become simply subsets of conditions. The general definition now specialises to the following.

A *basic Petri net* consists of

- a set of *conditions*, $B$,
- a set of *events*, $E$, and
- two maps: a *precondition* map $pre : E \to \mathcal{P}ow(B)$, a *postcondition* map $post : E \to \mathcal{P}ow(B)$. We can still write ${}^{\cdot}e$ for the preconditions and $e^{\cdot}$ for the postconditions of $e \in E$ and we require ${}^{\cdot}e \cup e^{\cdot} \neq \emptyset$.

Now a *marking* consists of a subset of conditions, specifying those conditions which hold.

**Token game for basic nets:** Markings can change as events occur, precisely how being expressed by the transitions

$$\mathcal{M} \xrightarrow{e} \mathcal{M}'$$

events $e$ determine between markings $\mathcal{M}, \mathcal{M}'$.

For $\mathcal{M}, \mathcal{M}' \subseteq B$ and $e \in E$, define

$$\begin{aligned} \mathcal{M} \xrightarrow{e} \mathcal{M}' \text{ iff } & (1)\ {}^{\cdot}e \subseteq \mathcal{M} \ \&\ (\mathcal{M} \setminus {}^{\cdot}e) \cap e^{\cdot} = \emptyset \text{ (Concession), and} \\ & (2)\ \mathcal{M}' = (\mathcal{M} \setminus {}^{\cdot}e) \cup e^{\cdot}\ . \end{aligned}$$

Property (1) expresses that the event $e$ has *concession* at the marking $\mathcal{M}$. Returning to the definition of concession for general nets, of which it is an instance, it ensures that the event does not load another token on a condition that is already marked. Property (2) expresses in terms of sets the marking that results from the occurrence of an event. So, an occurrence of the event ends the holding of its preconditions and begins the holding of its postconditions. (It is possible for a condition to be both a precondition and a postcondition of the same event, in which case the event is imagined to end the precondition before immediately restarting it.)

There is *contact* at a marking $\mathcal{M}$ when for some event $e$

$$^{\cdot}e \subseteq \mathcal{M} \ \& \ (\mathcal{M} \setminus {}^{\cdot}e) \cap e^{\cdot} \neq \emptyset .$$

The occurrence of an event is blocked through conditions, which it should begin, holding already. Blocking through contact is consistent with the understanding that the occurrence of an event should end the holding of its preconditions and begin the holding of its postconditions; if the postconditions already hold, and are not also preconditions of the event, then they cannot begin to hold on the occurrence of the event. Avoiding contact ensures the freshness of names in the semantics of name creation.

Basic nets are important because they are related to many other models of concurrent computation, in particular, Mazurkiewicz trace languages (languages subject to trace equivalence determined by the independence of actions) and event structures (sets of events with extra relations of causality and conflict)—see [WN95].

## C Nets with persistent conditions

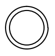
Sometimes we have use for conditions which once established continue to hold and can be used repeatedly. This is true of assertions in traditional logic, for example, where once an assertion is established to be true it can be used again and again in the proof of further assertions. Similarly, if we are to use net events to represent rules of the kind we find in inductive definitions, we need conditions that persist.

Persistent conditions can be understood as an abbreviation for conditions within general nets which once they hold, do so with infinite multiplicity. Consequently any number of events can make use of them as preconditions but without their ever ceasing to hold. Such conditions, having unbounded capacity, can be postconditions of several events without there being conflict.

To be more precise, we modify the definition of basic net given above by allowing certain conditions to be *persistent*. A net with persistent conditions will still consist of events and conditions related by pre- and postcondition maps which to an event will assign a set of preconditions and a set of postconditions. But, now amongst the conditions are the persistent conditions forming a subset $P$. A marking of a net with persistent conditions will be simply a subset of conditions, of which some may be persistent. Nets with persistent conditions have arisen independently several times and have been studied for example in *contextual nets* [MR95].

A net with persistent conditions can be understood on its own terms, or as standing for a general net with the same sets for conditions and events. The general net's capacity function will be either 1 or $\infty$ on a condition, being $\infty$ precisely on the persistent conditions. When $p$ is persistent, $p \in e^{\cdot}$ is interpreted in the general net as $(e^{\cdot})_p = \infty$, and $p \in {}^{\cdot}e$ as $({}^{\cdot}e)_p = 1$. A marking of a net with persistent conditions will correspond to a marking in the general Petri net in which those persistent conditions which hold do so with infinite multiplicity.

Graphically, we'll distinguish persistent conditions by drawing them as double circles:



**Token game with persistent conditions:** The token game is modified to account for the subset of conditions $P$ being persistent. Let $\mathcal{M}$ and $\mathcal{M}'$ be markings (*i.e.* subsets of conditions), and $e$ an event. Define

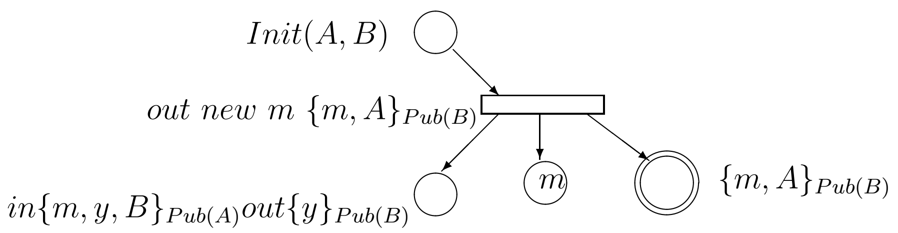
$$\mathcal{M} \xrightarrow{e} \mathcal{M}' \text{ iff } {}^{\cdot}e \subseteq \mathcal{M} \ \& \ (\mathcal{M} \setminus ({}^{\cdot}e \cup P)) \cap e^{\cdot} = \emptyset \ (e \text{ has concession}), \text{ and}$$
$$\mathcal{M}' = (\mathcal{M} \setminus {}^{\cdot}e) \cup e^{\cdot} \cup (\mathcal{M} \cap P) \ .$$

The token game fits our understanding of persistence, and specifically it matches the token game in its interpretation as a general net. In this paper, these special contextual nets are used in modelling and analysing security protocols.

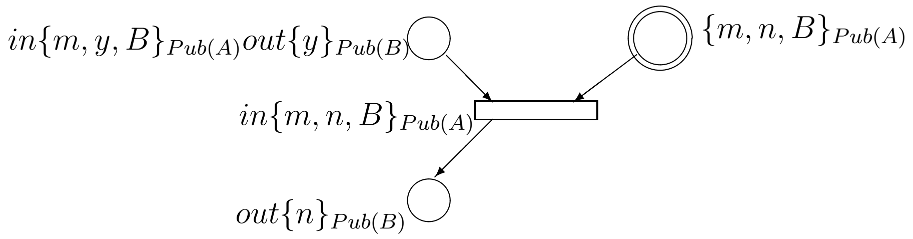
## D The events of NSL

We can classify the events $Ev(NSL)$ involved in the NSL protocol.
**Initiator events:**

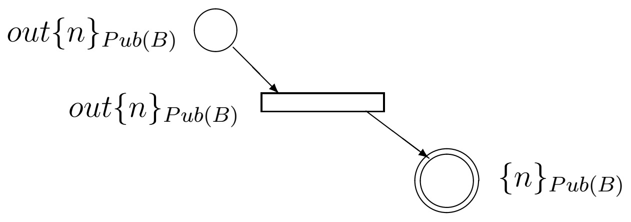
**Out**$(Init(A, B); m)$:


$Init(A,B)$
$out\ new\ m\ \{m,A\}_{Pub(B)}$
$in\{m,y,B\}_{Pub(A)}out\{y\}_{Pub(B)}$
$m$
$\{m,A\}_{Pub(B)}$

**In**$(in\{m,y,B\}_{Pub(A)}out\{y\}_{Pub(B)}; n)$:

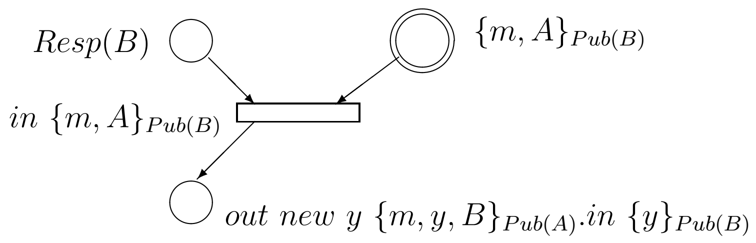


**Out**$(out\{n\}_{Pub(B)})$:

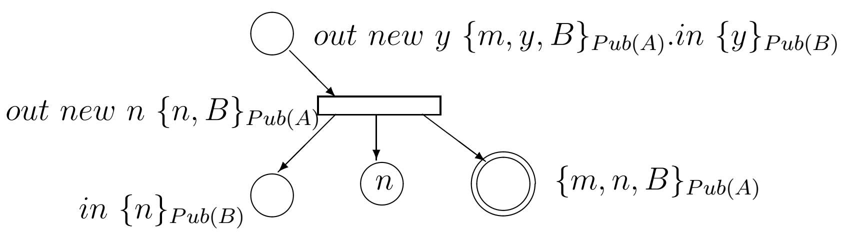


**Responder events:**

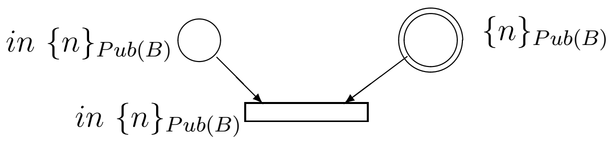
**In**$(Resp(B); m, A)$:



**Out**$(out\ new\ y\ \{m,y,B\}_{Pub(A)}.in\ \{y\}_{Pub(B)}; n)$:

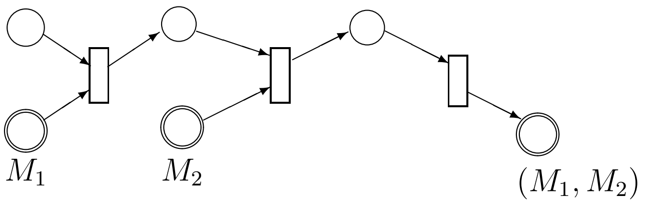

$out\ new\ y\ \{m,y,B\}_{Pub(A)}.in\ \{y\}_{Pub(B)}$
$out\ new\ n\ \{n,B\}_{Pub(A)}$
$in\ \{n\}_{Pub(B)}$
$n$
$\{m,n,B\}_{Pub(A)}$

**In**$(in\ \{n\}_{Pub(B)})$:

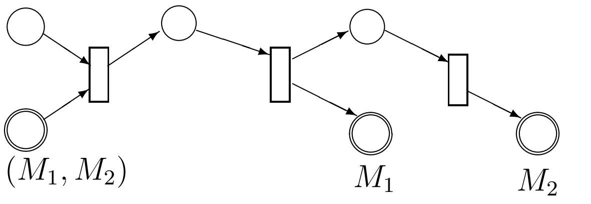


**Spy events:**

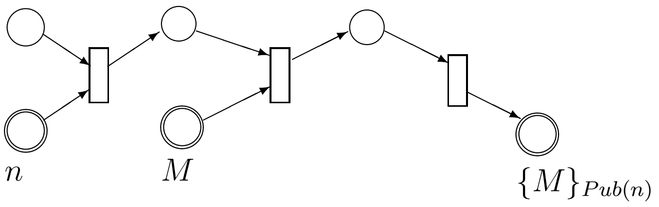
Composing, $Spy_1 \equiv in\ \psi_1.in\ \psi_2.out(\psi_1, \psi_2)$:



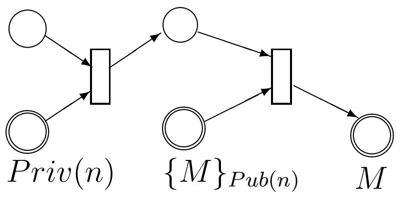
Decomposing, $Spy_2 \equiv in(\psi_1, \psi_2).out\ \psi_1.out\ \psi_2$:



Encryption, $Spy_3 \equiv in\ x.in\ \psi.out\ \{\psi\}_{Pub(x)}$:



Decryption, $Spy_4 \equiv in\ Priv(x).in\ \{\psi\}_{Pub(x)}.out\ \psi$:



## References

[AG97] M. Abadi and A. Gordon. A calculus for cryptographic protocols: The Spi calculus. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security*. ACM Press, 1997.

[CDL+99] I. Cervesato, N. A. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov. A meta-notation for protocol analysis. In R. Gorrieri, editor, *Proceedings of the 12th IEEE Computer Security Foundations Workshop - CSFW'99*, pages 55–69, Mordano, Italy, 28–30 June 1999. IEEE Computer Society Press.

[CDL+00] I. Cervesato, N. A. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov. Relating strands and multiset rewriting for security protocol analysis. In P. Syverson, editor, *13th IEEE Computer Security Foundations Workshop - CSFW'00*, Cambridge, UK, 3-5 July 2000. IEEE Computer Society Press.

[Low96] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *2nd International Workshop on Tools and Algorithms for the construction and Analysis of Systems*. Springer-Verlag, 1996.

[Mil99] R. Milner. *Communicating and mobile systems: The $\pi$-calculus*. Cambridge University Press, 1999.

[MR95] U. Montanari and F. Rossi. Contextual nets. *Acta Informatica*, (32), 1995.

[NT92] B. B. Nieh and S. E. Tavares. Modelling and analyzing cryptographic protocols using Petri Nets. In *Advances in Cryptology-AUSCRYPT '92*, volume 718 of *LNCS*, pages 275–295. Springer-Verlag, 1992.

[Pau98] L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

[Pau99] L. C. Paulson. Proving security protocols correct. In *Proceedings of the 14th Symposium on Logic in Computer Science*, July 1999.

[PS93] A. M. Pitts and I. Stark. Observable properties of higher order functions that dynamically create local names, or: What's new? In *Mathematical Foundations of Computer Science, Proc. 18th Int. Symp., Gdańsk, 1993*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141. Springer-Verlag, Berlin, 1993.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[THG98a] J. Thayer, J. Herzog, and J. Guttman. Honest ideals on strand spaces. In *Proceedings of the 11th IEEE Computer Security Foundations Workshop.* IEEE Computer Society Press, 1998.

[THG98b] J. Thayer, J. Herzog, and J. Guttman. Strand space pictures. Workshop on Formal Methods and Security Protocols, Indianapolis, Indiana, June 1998.

[THG98c] J. Thayer, J. Herzog, and J. Guttman. Strand spaces: Why is a security protocol correct? In *1998 IEEE Symposium on Security and Privacy.* IEEE Computer Society Press, 1998.

[Win87a] G. Winskel. Event structures. In *Proceedings of the Advanced Course on Petri nets*, volume 255 of *LNCS*. Springer–Verlag, 1987.

[Win87b] G. Winskel. Petri nets, algebras, morphisms, and compositionality. *Information and Computation*, 72:197–238, 1987.

[WN95] G. Winskel and M. Nielsen. *Models for concurrency*, volume IV. Oxford University Press, 1995.