Basic Research in Computer Science

BRICS RS-00-18 Crazzolara & Winskel: Language, Semantics, and Methods for Cryptographic Protocols

Language, Semantics, and Methods for Cryptographic Protocols

Federico Crazzolara
Glynn Winskel

BRICS Report Series **RS-00-18**

ISSN 0909-0878 **August 2000**

See back inner page for a list of recent BRICS Report Series publications.
Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/00/18/`

Language, semantics, and methods for cryptographic protocols

Federico Crazzolara, Glynn Winskel
{federico, gwinskel}@brics.dk

BRICS[*]
Department of Computer Science
University of Aarhus
Ny Munkegade, bldg. 540
DK-8000 Århus C

August 2000

Abstract

In this report we present a process language for security protocols together with an operational semantics and an alternative 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. This Petri-net semantics has a strong relation to both Paulson's inductive set of rules [Pau98] and strand spaces [THG98c]. By means of an example we illustrate how the Petri-net semantics can be used to prove properties such as secrecy and authentication. [1]

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

[1]This report is based closely on the Ph.D. progress report of the first author.

Contents

1 Introduction

In this section we introduce some basic concepts about security protocols, their properties, and the chosen model. We then briefly describe our work as well as related work.

1.1 Security protocols

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:

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

This protocol, like many others of this kind has two roles: one for the initiator, here A, and one for the responder, here B. It is a public-key protocol that assumes an underlying public-key infrastructure, such as RSA [RSA78]. Both initiator and receiver have their own, secret private key. Public keys in contrast are known to all participants in the protocol. In addition, the NSL protocol makes use of *nonces*. One can think of them as newly generated, unguessable numbers whose purpose is to ensure the freshness of messages.

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 his 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.

The NSL protocol aims at distributing nonces n and m in a secure way, allowing no one but the initiator and the responder to know them (*secrecy*). Another aim of the protocol is *authentication*: Bob should be guaranteed that n is indeed the nonce sent by Alice.

The protocol should be thought of a longer message-exchange sequence. After initiator and responder complete a protocol exchange, they will continue communication possibly using the exchanged nonces to establish a session key.

As experience shows, even if protocols are designed carefully, following established criteria [AN96], they may contain flaws. To be useful protocols in fact involve many concurrent runs among a set of distributed users. Then, the NSL protocol for example, is prone to an attack if the name of Bob, B, is not included in the second message. This attack on the original protocol of Needham and Shröder was discovered by Lowe [Low96]. Formal analysis of security protocols can both help to prove protocols correct with respect to a chosen model and to discover flaws.

1.2 Security properties

There is no common agreement on how to formalise the meaning of security. Just looking at secrecy or authentication properties, there are a variety of definitions [Aba98, Aba99, AG97, Low97, Ros98b, Sch96]. A property, even the formulation of a property, may be more interesting then another, depending on the protocol and the purpose for which it was designed. Given this abundance of different definitions, it is important to state precisely the properties that are proven and the conditions under which they hold.

Properties like authentication and secrecy can often be regarded as forms of safety properties in the sense that they reduce to a property holding of finite histories. In this report we consider them as such. When we talk about secrecy we mean that:

> "A message M is secret if it never appears unprotected on the medium."

This definition is used for example in Paulson's inductive method [Pau98] and in the strand-space approach [THG98c] but originates from Dolev and Yao [DY83]. A common definition of authentication is the agreement property defined in Lowe [Low97]:

> "An initiator A agrees with a responder B on same message M if whenever A completes a run of the protocol as initiator, using M apparently with B, then there exists a previous run of the protocol where B acts as responder using M apparently with A."

There are other security properties one would like to prove about protocols, such as anonymity, non-repudiation, etc. Some of them, e.g., non-interference, cannot be expressed as safety properties. Non interference provides a different definition of secrecy and is often used in process-algebra approaches [AG97, RS99]. Secrecy can be expressed by means of equivalence between processes [Aba99]:

> "Given a process $P(x)$ with only free variable x, P preserves secrecy of x if $P(M)$ and $P(N)$ are equivalent for all closed messages M and N."

We do not study this alternative version of secrecy in this report. However we do propose a process language for reasoning about protocols. It might be interesting in the future to investigate security properties expressed by process equivalences.

1.3 The Dolev-Yao model

The assumptions of the Dolev-Yao model are commonly used in modelling security protocols. First introduced in Dolev and Yao [DY83], their model underlies a variety of approaches, e.g., [Low96, MMS97, THG98c, Sch96, Ros98b, Pau98]. The basic assumptions of their model are:

- 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.

In the original Dolev-Yao model, it is shown that the problem *"Is a given protocol secure?"* is decidable for a very special kind of protocols, namely cascade protocols and name-stamp protocols [DY83]. In this context security is taken as being essentially secrecy. This class of protocols is rather restricted and current protocols are beyond its range. The security problem is undecidable for more general protocols [EG83].

1.4 This work

We present a language for modelling security protocols. The language was inspired by Paulson's inductive approach to crypto-protocol analysis [Pau99, Pau98]. 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 [Gel85] in some way. Both public-key and private-key cryptography can be handled and are treated as a black box.

We give both an operational and an event-based semantics for the language. The former gives a traditional style of operational semantics to describe the behaviour of interacting processes by means of an open network, the latter describes a process by its set of events giving rise to a Petri net. The main reason we introduced the event based semantics is to provide a basis for the kind of local reasoning employed for

cryptographic protocols. The difficulty we encountered in using the more traditional operational semantics was that actions of the same component can appear very far away from each other in a trace, separated by actions from other components. Proving them part of the same component turned out to be rather difficult. A semantics in terms of events, is our answer to this problem. Although the operational semantics is given first, it is the event-based semantics that best shows the connection to other approaches such as the inductive method [Pau99] and strand spaces [THG98c]. Though, in this report we do not formalise this correspondence, and we do not prove the relation between operational and Petri-net semantics. These questions are to be part of the Ph.D. work of the first author.

An example helps us to illustrate how proof techniques that are used in strand space like proofs can be transferred to our model. Our hope is that other different techniques transfer and thus can be compared.

Taking the Needham-Schröder-Lowe (NSL) protocol as an example we briefly introduce the inductive approach, in Section 2, as well as the strand-space approach, in Section 3. In Section 4 we introduce our language, give an operational and a Petri-net semantics. The NSL protocol example illustrates how a system of interactive agents as well as a malicious user can be programmed. Authentication and secrecy guarantees for the responder are proven in Section 5, on the basis of our Petri-net semantics. In Section 6 we conclude with an introduction of future work.

1.5 Related work

Our work is in many ways related to the strand-space approach [THG98a, THG98b, THG98c]. In fact the single processes, components of the system, can be viewed as strands and our proofs use them much in the same way. This fact is perhaps not surprising, since our language is inspired by Paulson's inductive approach [Pau, Pau98, Pau99]. Strand-space proofs are claimed to be shorter than the ones carried out in the inductive model. Moreover they can be done by hand. The usual reason given for the advantage of strands over a set of rules, is that strands take the dependencies among events of a crypto-protocol into better account. It seems to us however, that proofs are shorter mainly because the desired property is not proven directly, but via a "smart" invariant. We believe that in many cases the same reasoning could be done based in the inductive method. In fact the premises of a set of rules give the desired dependencies. We hope in the future to relax our semantics so as to capture precisely the inductive model, and formally compare the two known approaches.

Formalising a protocol using a language is merely a programming activity and requires less work than to give a set of rules or a strand space right away. Agents are simply processes and so is the intruder. The programmer has the freedom to tune the power of the malicious user, e.g., to a passive rather then an active intruder.

It is worth noting that strand spaces have been used recently as one basis for an automatic checker [Son99].

We discovered similarities between our approach and one using multiset rewritings based on linear logics [CDL+99], where logical formulas play a role similar to the processes, components of a system programmed in out language. A recent paper [CDL+00] connecting strands with multiset rewriting systems, suggests further similarities with our work.

With respect to other process-algebra approaches [AG97, Ros98b, Sch96] we choose the open-network view, rather then allowing private channels. Everything that is sent can in principle be eavesdropped and only encryption is used to protect the transmitted data. Our language is asynchronous, whereas the Spi calculus [AG97] is based on the synchronous π-calculus, and in contrast to CSP-based approaches [Ros98b, Sch96], there is no need to program the medium explicitly.

There are numerous model checking approaches, e.g., [Low96, MMS97, Ros95], sometimes connected with process calculi.

In comparison with logic approaches such as BAN logic [BAN89], with a process language we are closer to the implementation of protocols. Nevertheless we hope to take advantage of logics for reasoning about sequences of events.

We gave a Petri-net semantics to our language. Petri nets have previously been applied to the verification of cryptographic protocols [NT92], though this application of Petri nets does not seem to be fully explored yet. We are not sure how our approach relates to the existing work on Petri nets [NT92].

Finally we want to mention the world of computational and information-theoretic approaches [Can00]. There is a gap between formal methods and the cryptologist's techniques to prove protocols correct. A first attempt to bridge the gap starting from the formal methods side can be found in [LMMS98a, LMMS98b, LMMS99].

1.6 Prerequisites

Some familiarity with the inductive method and the strand-space method for protocol verification would be helpful in realising this report, as would basic knowledge of the π-calculus [Mil99, Bou91], in particular of its cryptographic version, the Spi calculus [AG97]. In giving semantics to our language we make some use of ideas from Petri nets.

2 The inductive approach

In this section we briefly summarise the inductive modelling approach to security protocols making use of the NSL-protocol example.

$$\lambda \in NSL \quad (empty)$$

$$\frac{t \in NSL \quad Says\ A\ B\ M \in set(t)}{(Gets\ B\ M)\hat{}t \in NSL}\ (receive)$$

$$\frac{t \in NSL \quad n \notin parts(set(t))}{(Says\ A\ B\ \{n, A\}_{Pub(B)})\hat{}t \in NSL}\ (1)$$

$$\frac{t \in NSL \quad Gets\ B\ \{n, A\}_{Pub(B)} \in set(t) \quad m \notin parts(set(t))}{(Says\ B\ A\ \{n, m, B\}_{Pub(A)})\hat{}t \in NSL}\ (2)$$

$$\frac{t \in NSL \quad Says\ A\ B\ \{n, A\}_{Pub(B)} \in set(t) \quad Gets\ A\ \{n, m, B\}_{Pub(A)} \in set(t)}{(Says\ A\ B\ \{m\}_{Pub(B)})\hat{}t \in NSL}\ (3)$$

$$\frac{t \in NSL \quad M \in spy(t)}{(Says\ Spy\ B\ M)\hat{}t \in NSL}\ (fake)$$

Figure 1: NSL-protocol rules

The protocol is modelled by a set NSL of traces, i.e., sequences of events: Event $Says\ A\ B\ M$, means that agent A sends the message M to B. Event $Gets\ A\ M$ instead means that A received the message M. The receiver does not know who is the sender of the message, unless the message itself contains enough information to authenticate it. The protocol traces are built up inductively by a set of rules as shown in Figure 1. The presented inductive definition, differs slightly from the one presented in Paulson [Pau98], taking explicit input events "*Gets*" as in [Pau99], rather then only output events "*Says*". One should think of the given rules, as schemata. A, B should be thought of as ranging over a set of *Agents* participating in the protocol. Similarly n, m range over a set of *Nonces* and *Keys*, M over all possible *Messages* that can be constructed from a set of values, by encryption and composition. We write $set(t)$ for the set of messages in the trace t, and $parts(S)$ for the set of sub-components of messages in a set S defined in the obvious way (see [Pau99] for a formal definition). To each message of the informal protocol description we have a corresponding rule:

(1) We can extend a given trace with the first output message of the protocol description, provided that A's nonce is new, i.e., not already present in the trace to be extended.

(2) Extending a trace with the second message of the protocol requires not only B's

nonce to be chosen freshly, i.e., $m \notin parts(set(t))$, but also A's message to be previously received by B.

(3) Similarly for the last message that A sends to B. In addition, the nonce sent by A in the first message has to correspond to the the nonce returned to A. This is achieved simply by adding A's first message as a premise.

(*receive*) A principal can get a message, only if it has previously been sent to him. The rule does not require any other premise, thus makes the reception of a message always possible.

($fake$) The last rule models the spy. We write $spy(t)$ for the set of messages the intruder can build up from past traffic. Typically Spy will not only be able to eavesdrop on all messages, i.e., all messages appearing in t are included in $spy(t)$, but also to build up new messages or extract parts of messages. The active spy will have the ability to encrypt and decrypt with all available keys, typically all public keys and the private keys of corrupted agents. A precise description of how this set is obtained can be found in [Pau98, Pau99].

It is noteworthy that once the premises hold for a rule, it can be applied any number of times. This adds stuttering to NSL traces, as in the following trace:

$$(Says A\, B\{n, A\}_{Pub(B)})(Gets\, B\{n, A\}_{Pub(B)})(Gets\, B\{n, A\}_{Pub(B)})(Says A\, B\{n, A\}_{Pub(B)})$$

One could think of a tighter model with a distinct receive rule for each different message of the protocol and with premises that enforce sequential events. To prove properties about protocols, it seems that the more "generous" model described in Figure 1 is sufficient. The reason is that we want protocols to be secure in a hostile environment that may have stuttering, which might be provided by the spy.

Security properties such as secrecy and authentication can be defined on the set of traces of the protocol. The rules are used to prove properties inductively. Secrecy of the initiator's nonce corresponds for example to:

$$\forall t \in NSL.(Says\ A\ B\ \{n, A\}_{Pub(B)}) \in set(t) \Rightarrow (Says\ Spy\ B\ n) \notin set(t)$$

This is indeed a sufficient condition to guarantee secrecy: For all messages M in the trace t, the nonce n does not appear in clear text. If n did appear in clear text, then Spy could see it, i.e., $n \in spy(t)$ and use the ($fake$) rule to extend the trace by ($Says\ Spy\ B\ n$). The previous theorem holds for the NSL protocol, provided that the secret keys of both initiator and responder are not leaked to the spy, i.e., $Priv(A), Priv(B) \notin spy(t)$.

Since proofs of security properties based on rule induction can be long, machine support, using, e.g., the Isabelle theorem prover [Pau94], is particularly valuable.

$$
\begin{array}{ll}
\{Init(A,B,n,m) \mid A,B \in Agents, n,m \in Nonces\} & (Init) \\
\cup \{Resp(A,B,n,m) \mid A.B \in Agents, n,m \in Nonces\} & (Resp) \\
\cup \{(GetsSpy\, M)(SaysSpy\, A\, M)(SaysSpy\, A\, M) \mid M \in Msg, A \in Agents\} & (stutt) \\
\cup \{(GetsSpy\, M)(GetsSpy\, N)(SaysSpy\, A\, (M,N)) \mid M,N \in Msg, A \in Agents\} & (tupl) \\
\cup \{(GetsSpy\, (M,N))(SaysSpy\, A\, M)(SaysSpy\, A\, N) \mid M,N \in Msg, A \in Agents\} & (dec) \\
\cup \{(SaysSpy\, A\, k) \mid k \in Bad, A \in Agents\} & (bad) \\
\cup \{(GetsSpy\, Pub(B))(GetsSpy\, M)(SaysSpy\, A\{M\}_{Pub(B)}) \mid M \in Msg, A,B \in Agents\} & (encr) \\
\cup \{(GetsSpy\, Priv(B))(GetsSpy\{M\}_{Pub(B)})(SaysSpy\, A\, M) \mid M \in Msg, A,B \in Agents\} & (decr)
\end{array}
$$

Figure 2: NSL-protocol strand space

3 The strand-space approach

In this section we briefly recall the strand-space approach to modelling security protocols [THG98c, THG98a, THG98b]. The NSL protocol is again our example.

As for the inductive approach, the model is based on sequences of events. Instead of taking traces of the entire protocol execution, consider sequences of events that correspond to single protocol exchanges of each distinct agent. No rules are needed to construct such sequences. They can simply be read from the informal protocol description. For reasons of uniformity with the previous section, let us keep the events to be of the same kind: $Says\ A\ B\ M$ and $Gets\ B\ M$. For the NSL protocol we have:

$$
\begin{array}{l}
Init(A,B,n,m) = (Says\, A\, B\{n,A\}_{Pub(B)})(Gets\, A\{n,m,B\}_{Pub(A)})(Says\, A\, B\{m\}_{Pub(B)}) \\
Resp(A,B,n,m) = (Gets\, B\{n,A\}_{Pub(B)})(Says\, B\, A\{n,m,B\}_{Pub(A)})(Gets\, B\{m\}_{Pub(B)})
\end{array}
$$

where $Init(A,B,n,m)$ stands for the trace of an initiator's run and $Resp(A,B,n,m)$ for the trace of a responder's run. We call these sequences *strands*. A *strand space* is simply a collection of strands. For simplicity we do not consider having repetitions of the same trace in the strand space. This would mean tagging each occurrence with a different index. A full treatment can be found in [THG98c]. Repetitions allow us to model an agent doing the same run of the protocol more than once. The NSL protocol indeed does not allow this, since a new nonce will be chosen each time, and so all traces are distinguished. Nevertheless we take care of repetitions in the rest of the report, since we are concerned with a semantics for a language that aims at modelling a range of protocols.

If we want to model a malicious intruder, we add strands for each capability of the intruder. Figure 2 shows an *infiltrated strand space* that models the NSL protocol:

($Init$) The initiator strands. We instantiate over the set of *Agents* participating in

the protocol. In this way we allow any agent to act as an initiator with any other participant. We instantiate as well over all possible *Nonces*.

(*Resp*) The responder strands.

(*sutt*) This infiltrated strand allows the spy to eavesdrop on any message and replicate it. It builds stuttering into the system.

(*tupl*) Allows the spy to concatenate any two eavesdropped messages.

(*dec*) The spy can also decompose any composite message.

(*encr*) Whenever the spy gets hold of a key, she can use it for encryption

(*decr*) or for decryption.

(*bad*) The intruder knows all leaked keys. With Bad we indicate the set of keys the intruder can get hold of.

Msg is the set of messages that can be built up from an initial set of values, by encryption, decryption, and composition.

A strand space has an obvious graph associated with it. The graph describes the dependencies among events in terms of possible interactions among agents and the control points on a particular strand. Each event of a strand is a node of the graph. The edges are among two consecutive events of the same strand and from events of the form $Gets\ C\ M$ to $Says\ A\ B\ M$ events. Therefore an input event can only occur if the same message has already been sent. Moreover an event can occur only if all the events preceeding it on the same strand, already occurred.

As we mentioned we are interested in showing that for each run of the protocol some property holds. A protocol exchange is defined in terms of a bundle, a finite and acyclic subgraph of the strand-space graph such that:

- For each event belonging to the subgraph all preceeding events of the same strand belong to the subgraph.
- For each input event there is a unique corresponding send.

Figure 3 shows a bundle for the NSL protocol. The secrecy property of the initiator's nonce can be stated in this model as the following:

$$\forall \mathcal{C}.(Says\ A\ B\ \{n, A\}_{Pub(B)}) \in \mathcal{C} \Rightarrow (Says\ Spy\ B\ n) \notin \mathcal{C}$$

where $\mathcal{C}$ is a bundle. Again one can show the property to hold provided that $Priv(A), Priv(B) \notin Bad$. Some more requirements are needed in reality: The nonce n is uniquely originating, i.e., there is only one strand with n appearing as

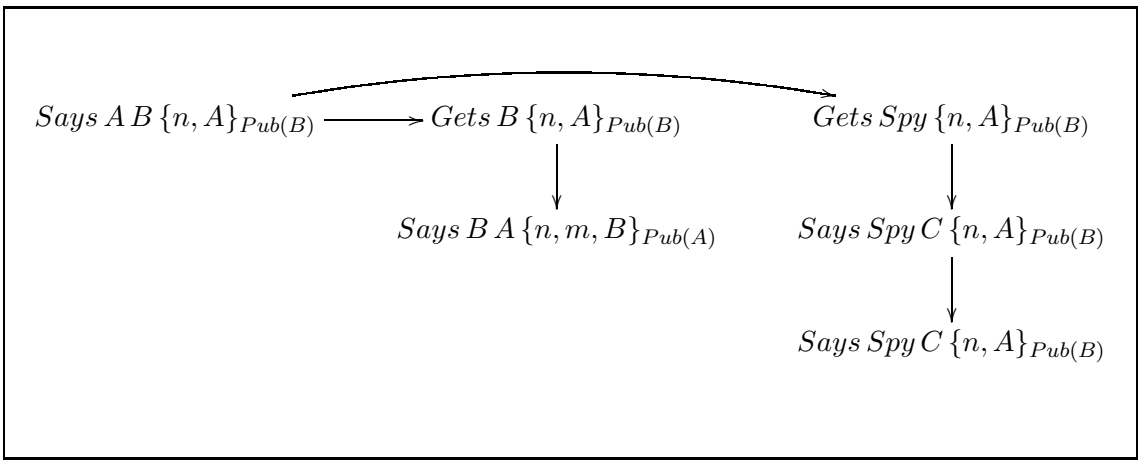


Figure 3: An NSL-protocol bundle

sub-message on a "*Says*" event and no other event containing n precedes it on the same strand. The concept of unique origination captures the idea that a nonce is fresh and is a central concept in proving properties in this model. For a better treatment of origination and the proof of the above property we refer to [THG98c]. Here we just recall that the argument of the proof is simplified by proving, inductively, a slightly stronger statement. In Section 5 we present a proof of secrecy that follows very much the same lines.

4 A language for security protocols

In this section we introduce a language for modelling and verifying security protocols. We first introduce the syntax of the language, its operational semantics 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 5.

4.1 The language

We start by giving the syntactic sets of the language:

- An infinite set of **Names**, with n, m ranging over it
- A set of agents, $A, B \in$ **Agents**
- A set of indexes, $i \in$ **Indexes**
- Variables over names $x, y, z \in$ **Nvar**
- Variables over agents $X, Y, Z \in$ **Avar**

Values	v	::=	$n \mid x \mid A \mid X$
Keys	k	::=	$Pub(v) \mid Priv(v) \mid Key(v)$
Patterns	Π, Π'	::=	$v \mid k \mid (\Pi, \Pi') \mid \psi$
Messages	M, N	::=	$\Pi \mid (M, N) \mid \{M\}_k \mid \psi$
Processes	P	::=	nil
			$\mid new(x).P$
			$\mid out(M).P$
			$\mid in(\Pi).P$
			$\mid [M = N].P$
			$\mid [M > \Pi].P$
			$\mid \|_{i \in I} P_i$

Figure 4: Syntactic sets

- Variables over messages $\psi \in \mathbf{Mvar}$

We take **Names** and **Agents** to be disjoint sets, as are the three sets of variables. The set **Indexes** contains **Agents** and the set $I\!N$ of natural numbers. The other syntactic sets of the language are built up by the grammar shown in Figure 4.

We use injections to distinguish between public, private and symmetric keys. Keys can be used in building up encrypted messages. As in the Spi calculus, $\{M\}_k$ denotes the message M encrypted using the key k. As we also, for instance, write $\{M\}_k$ for decryption under the key k a term of this form may or may not be ciphertext. The set of messages is obtained from the grammar in Figure 4, equated with the least substitutive equivalence relation such that:

$$\begin{aligned} \{\{M\}_{Priv(v)}\}_{Pub(v)} &= M \\ \{\{M\}_{Pub(v)}\}_{Priv(v)} &= M \\ \{\{M\}_{Key(v)}\}_{Key(v)} &= M \end{aligned}$$

The resulting message algebra allows an intuitive treatment of encryption and decryption. In particular, if M is already a ciphertext, then $\{M\}_k$ may stand for the clear text obtained decrypting with the right key. We believe that this algebra of messages is well suited to the treatment of both public and symmetric-key encryption. In contrast to the Spi calculus [AG97] and other approaches [THG98c, Pau98], we are no longer bound to the *strong encryption assumption*:

$$\{M\}_k = \{N\}_{k'} \Rightarrow M = N \ \wedge \ k = k'$$

It is for example the case that

$$\{\{\{n\}_{Pub(B)}\}_{Pub(A)}\}_{Priv(A)} = \{n\}_{Pub(B)}$$

but $\{\{n\}_{Pub(B)}\}_{Pub(A)} \neq n$ and $Pub(B) \neq Priv(A)$, which clearly does not respect that assumption. Our approach allows for more expressive power and still guarantees that *clear text can be obtained from ciphertext only using the right key.* 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.

Proposition 4.1 *For every message there is a unique reduced form. Any two messages M, N such that $M = N$ have the same reduced form.*

Proof. From the confluence property of the message reduction relation and by induction on the definition of equality among messages. □

Given a message M we write $red(M)$ for its reduced form. The following property holds for public and symmetric-key encryption. For simplicity we look at the symmetric keys only.

Proposition 4.2 *Let M, N be messages and k, k' symmetric keys. Then*

1. *If $\{M\}_k, \{N\}_{k'}$ are in reduced form then $\{M\}_k = \{N\}_{k'} \Rightarrow M = N \wedge k = k'$.*
2. *$\{\{M\}_k\}_{k'} = M \Rightarrow k = k'$.*

Proof. Property 1) follows from the existence of a unique reduced form, (Proposition 4.1) and 2) follows by well-founded induction from a) and from Proposition 4.1. □

We distinguish patterns from messages, in that they do not contain encryption. An example that shows how to recover clear text form an encrypted message is the following

$$in(\psi') \,.\, [\{\psi'\}_{Priv(A)} > \psi] \,.\, out(\psi) \,.\, nil$$

If a message $\{M\}_{Pub(A)}$ is received as input, thus substituted instead of ψ', then by virtue of the message algebra the *case* construct will extract the ciphertext M and bind it to ψ. Proposition 4.2 ensures that this can happen only if the right key is used for decryption.

Let $var(M)$, $var(\Pi)$ be the variables of a message M and pattern Π respectively. The set of free variables of a process, $fv(P)$, is defined by structural induction in Figure 5. As usual, we say that a process without free variables is closed, as is a message without variables. We will write **Proc** for the set of closed processes and **Msg** for the set of closed messages that are in reduced form. Variables that are not free are said to be bound. Informally speaking, the semantics of processes is as follows:

nil The nil process, with no actions.

$$
\begin{array}{lcl}
fv(nil) & = & \emptyset \\
fv(new(x).P) & = & fv(P)\backslash\{x\} \\
fv(out(M).P) & = & var(M) \cup fv(P) \\
fv(in(\Pi).P) & = & fv(P)\backslash var(\Pi) \\
fv([M = N].P) & = & var(M) \cup var(N) \cup fv(P) \\
fv([M > \Pi].P) & = & (fv(P)\backslash var(\Pi)) \cup var(M) \\
fv(\|_{i\in I} P_i) & = & \bigcup_{i\in I} fv(P_i)
\end{array}
$$

Figure 5: Free variables of process terms

$new(x).P$ This chooses a new, fresh name and binds it to the variable x in P. (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.) Our treatment of *new* is a little different from the restriction of the π-calculus [Mil99], as will be clear from the formal operational semantics of the language, given in the next section.

$out(M).P$ outputs the message M on the medium.

$in(\Pi).P$ waits for an input that matches the pattern Π and binds the variables occurring in the pattern.

$[M = N].P$ The *match* construct hands the control over to process P if $M = N$ and behaves like nil otherwise. This construct will be especially useful for testing nonces, as shown in the example of Section 4.3.

$[M > \Pi].P$ As we already mentioned, the *case* construct is used to decompose messages in subcomponents. It checks whether there is a substitution of the pattern variables of Π that matches the message M. If so it binds those variables, otherwise behaves like nil.

Protocols usually involve concurrent agents participating in different parallel runs. Parallel composition will therefore play an important role in our language. In expressing properties about protocols we want to say that some action is performed by a particular agent. Tagging actions with the name of that agent will allow us to do so. We have an indexed form of parallel composition:

$\|_{i\in I} P_i$ is the parallel composition of all components in the indexing set $I \subseteq \mathbf{Indexes}$. We can abbreviate $\|_{i\in\mathbb{N}} P$ to $!P$ and $\|_{i\in\{j\}} P_i$ to $j : P_j$.

Before defining a formal operational semantics in Section 4.2 we first define the set of *names* of a process. We make use of this later. Figure 6 shows the definition by

$$
\begin{array}{lcl}
names(nil) & = & \emptyset \\
names(new(x).P) & = & names(P) \\
names(out(M).P) & = & names(P) \cup names(M) \\
names(in(\Pi).P) & = & names(P) \cup names(\Pi) \\
names([M = N].P) & = & names(P) \cup names(M) \cup names(N) \\
names([M > \Pi].P) & = & names(P) \cup names(\Pi) \cup names(M) \\
names(\|_{i \in I} P_i) & = & \bigcup_{i \in I} names(P_i)
\end{array}
$$

Figure 6: Names of processes

structural induction on processes where $names(M)$ is the set of names of a message and is assumed to be defined in the obvious way.

We as well assume a definition of variable substitution σ for messages and processes. We require it to substitute for a variable $x \in \mathbf{Nvar}$ a name, for a variable $X \in \mathbf{Avar}$ an agent, and a closed and fully reduced message for a variable $\psi \in \mathbf{Mvar}$. Let $names(\sigma)$ be the set of names of the substituted values or messages, listed in σ. The substitution will be such that the following property is satisfied:

Proposition 4.3 $names(P[\sigma]) \subseteq names(P) \cup names(\sigma)$.

4.2 An operational semantics

We give the operational semantics for closed processes by $(S, L, Tran)$, a transition system where:

- The set of states S is the set of triples P, s, t with $P \in \mathbf{Proc}$ a closed process, $s \subseteq \mathbf{Names}$, and $t \subseteq \mathbf{Msg}$ a set of closed and reduced messages, such that $names(P) \subseteq s$ and $names(t) \subseteq s$.

- The set of labels L is the set of closed, possibly tagged actions $\mathbf{Act}$ defined as following:
$$\alpha ::= out(M) \mid in(M) \mid i : \alpha$$
with $M \in \mathbf{Msg}$ and $i \in \mathbf{Indexes}$.

- The transition relation $Tran \subseteq S \times L \times S$ is the smallest relation obtained inductively from the rules in Figure 7.

In Figure 7, by $\|_{i \in I} P_i[P'_j/j]$ we mean the process $\|_{i \in I} P_i$ where P'_j is substituted for P_j.

$$out(M).P, s, t \xrightarrow{out(red(M))} P, s, t \cup \{red(M)\} \ (send)$$

$$\frac{\Pi[\sigma] \in t}{in(\Pi).P, s, t \xrightarrow{in(\Pi[\sigma])} P[\sigma], s, t} \ (receive)$$

$$\frac{P[n/x], s \cup \{n\}, t \xrightarrow{\alpha} P', s', t'}{new(x).P, s, t \xrightarrow{\alpha} P', s', t'} \ n \notin s \ (new)$$

$$\frac{P, s, t \xrightarrow{\alpha} P', s', t'}{[M = M].P, s, t \xrightarrow{\alpha} P', s', t'} \ (match)$$

$$\frac{M = \Pi[\sigma] \quad P[\sigma], s, t \xrightarrow{\alpha} P', s', t'}{[M > \Pi].P, s, t \xrightarrow{\alpha} P', s', t'} \ (case)$$

$$\frac{P_j, s, t \xrightarrow{\alpha} P_j', s', t'}{\|_{i \in I} P_i, s, t \xrightarrow{j:\alpha} \|_{i \in I} P_i[P_j'/j], s', t'} \ (par)$$

Figure 7: Operational semantics: Transition relation

Lemma 4.4 *The relation defined in Figure 7 is well-defined, i.e.* $Trans \subseteq S \times L \times S$.

Proof. By rule induction: if $P, s, t \in S$ and $(P, s, t, \alpha, P', s', t') \in Trans$ then $P', s', t' \in S$. □

Given the transition system just defined, we can talk about computations of a closed process.

Definition 4.5 *For* $P \in$ **Proc**, *a sequence of transitions*

$$P_0, s_0, t_0 \xrightarrow{\alpha_0} P_1, s_1, t_1, \ldots, P_i, s_i, t_i \xrightarrow{\alpha_i} P_{i+1}, s_{i+1}, t_{i+1}$$

such that $(P_i, s_i, t_i, \alpha_i, P_{i+1}, s_{i+1}, t_{i+1}) \in Tran$ *and* $P_0 = P$ *is called a computation of* P *and the sequence of actions* $\alpha_0 \ldots \alpha_i$ *a trace of* P.

What we get is an asynchronous operational semantics, in fact at every moment the set of messages t, contains all the messages that have been sent out by the processes. Everything in t matching the required pattern can nondeterministically be received as input. The input action does not consume the occurrence of the

message in t, thus there is no restriction on the number of times the same message can be received. The set of messages t has analogy with the tuple space of Linda [Gel85]. In contrast to Linda, we did not find it necessary to model the fact that a message can be deleted from t. We can get the effect of suppressing a message, by allowing a malicious user to flood the tuple space with enough junk. A similar choice is made in the inductive approach [Pau98].

Observation 4.6 In the states of our transition system we need not only to carry along the set of messages t but also the set of names that has already been generated either by *new* or were originally present in the process. This avoids problems for terms like $new(x).new(y).P$. If we inferred the set of names from t we would not be able to detect that a new name has just been generated, when executing the second *new*. We would risk choosing the same name for both *new* occurrences.

4.3 Example: Programming the NSL protocol

We now program the NSL protocol in our language. Doing this we formalise the description given in the Introduction 1.1. Given the set of **Agents** participating in the protocol, we have seen that in the NSL protocol they play two roles, as initiator and responder. We assume that each agent can participate in the protocol both as initiator and responder with any other agent. Abbreviate by $Init(A, B)$ the program of initiator A communicating with B and by $Resp(B)$ the program of responder B. The code of both an arbitrary initiator and an arbitrary responder is given in Figure 8. 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 9, shows a very general, active intruder, as inherited from the Dolev-Yao model. *Bad* is the set of agents that are corrupted, because their private key has been leaked. As we saw in Section 3, 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 intruder 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. The system is described in Figure 10.

4.4 A Petri-net semantics

In trying to prove properties of protocols using our operational semantics a difficulty that we had was to connect the syntax of the programs with the mathematical

$$
\begin{array}{lcl}
Init(A,B) & = & new(x). \\
 & & out\{x, A\}_{Pub(B)}. \\
 & & in(\psi). \\
 & & [\{\psi\}_{Priv(A)} > (z, y, B)]. \\
 & & [x = z]. \\
 & & out\{y\}_{Pub(B)}. \\
 & & nil \\
 & & \\
Resp(B) & = & in(\psi). \\
 & & [\{\psi\}_{Priv(B)} > (x, X)]. \\
 & & new(y). \\
 & & out\{x, y, B\}_{Pub(X)}. \\
 & & in(\psi'). \\
 & & [\{\psi'\}_{Priv(B)} > z]. \\
 & & [y = z]. \\
 & & nil
\end{array}
$$

Figure 8: Initiator and responder code

reasoning on the formal semantics. In fact we often wanted to prove facts such as:

> Given an event occurring in a trace, and the event corresponds to a certain action in the program, then there must be events in the trace that correspond to all preceeding actions in the program.

This intuitively obvious statement turned out to be tedious to prove, mostly because of the operational treatment of parallel composition; actions of the same component may occur in a computation very far away form 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. The idea is to consider a semantics based on events encapsulating all the necessary information needed to enable them. The information in the event allows us to reason backwards and conclude what other events had to precede it. Developing this idea brought us to a semantics given in terms of Petri nets (see for example [Win87]).

Building blocks of our semantics are events. As we mentioned, we will consider events, as an action together with the necessary preconditions to enable it. We will also add to an event the conditions satisfied once it has occurred.

Definition 4.7 *An event is a tuple* $e = (Pre, \alpha, Post)$*, with* $Pre = \{c_1, \ldots, c_h\}$ *a set of preconditions, and* $Post = \{d_1, \ldots, d_l\}$ *a set of postconditions and* α *an action.*

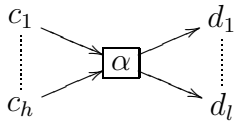
$$
\begin{array}{lcll}
Spy_1 & = & in(\psi).in(\psi').out(\psi,\psi').nil & \text{(composing)} \\
Spy_2 & = & in(\psi,\psi').out(\psi).out(\psi').nil & \text{(decomposing)} \\
Spy_3 & = & in(Pub(x)).in(\psi).out(\{\psi\}_{Pub(x)}).nil & \text{(encr.,decr.)} \\
Spy_4 & = & in(Priv(x)).in(\psi).out(\{\psi\}_{Priv(x)}).nil & \\
Spy_5 & = & \|_A\, out(Pub(A)).nil & \text{(public keys)} \\
Spy_6 & = & \|_{A\in Bad}\, out(Priv(A)).nil & \text{(leaked keys)} \\
 & & & \\
Spy & = & \|_{i\in\{1,\ldots,6\}} Spy_i &
\end{array}
$$

Figure 9: Intruder code

$$
\begin{array}{lcl}
P_1 & = & \|_A\, \|_B\, !\, Init(A,B) \\
P_2 & = & \|_A\, !\, Resp(A) \\
P_3 & = & !\, Spy \\
 & & \\
NSL & = & \|_{i\in\{1,2,3\}}\, P_i
\end{array}
$$

Figure 10: The system

We will often represent an event as:



As, for instance, in Winskel [Win87] we will use ${}^{\cdot}e$ for the preconditions of e and $e^{\cdot}$ for its postconditions. We have three kinds of conditions:

- Control conditions (P) with $P \in \mathbf{Proc}$ for keeping track of the evolving program.
- $\langle s \rangle$ conditions on names, requiring $s \subseteq \mathbf{Names}$ to be the set of used names.
- Network conditions $(\!(M)\!)$ with $M \in \mathbf{Msg}$, requiring the message M to be present on the network.

Control conditions and name conditions can be used only once and for one single event only. They are usual Petri-net conditions. Network conditions are of a special kind; they are "durable", meaning that once satisfied, they hold forever. This view

corresponds to the operational one of never removing messages from the network. Once sent, the message remains available on the medium. In particular we will write ${}^{(\cdot)}e$ for the "transient" preconditions of e, meaning the ones concerning names and control. An event can occur only if all the preconditions are available. A *marking* $\mathcal{M}$ 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 4.9.

We define a semantics that associates with every closed process P a set of events and an initial marking:

$$[\![P]\!] = (Ev(P), \mathcal{M}_0)$$

The initial marking

$$\mathcal{M}_0 = \{\langle s_0 \rangle, ((M_1)), \ldots, ((M_j))\} \cup Ic(P)$$

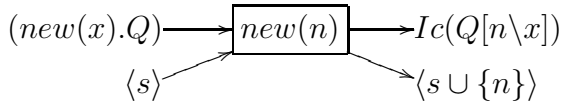
consists of an initial name condition $\langle s_0 \rangle$, where s_0 is the set of names considered already as used from the start, and therefore:

$$names(M_1, \ldots, M_j) \cup names(P) \subseteq s_0$$

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 $Ic(P)$: Control is handed over to the different parallel components of the program.

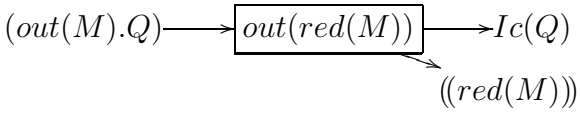
Figure 11 shows how to construct inductively the set of events $Ev(P)$ and initial control conditions $Ic(P)$ associated with a closed process P. We have three kinds of basic events:

- *New* events: for $n \notin s$



  $new(n)$ is an explicit event. This seems 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*. An action $new(n)$ will happen whenever the control point of the program is the expected one and set of used names does not contain name n. The next control point will be marked in the process where n is substituted for x and the set of used names is consequently modified.

- *Out* events:



  The output event not only causes the program control to evolve, but also puts the message M on the network. The message will remain always visible on the network as the "durable" condition $((red(M)))$ is marked.

By induction on the number of process constructors:

$$Ic(nil) = \quad \{(nil)\}$$
$$Ev(nil) = \quad \emptyset$$

$$Ic(new(x).Q) = \quad \{(new(x).Q)\}$$
$$Ev(new(x).Q) = \quad \Big\{(new(x).Q) \longrightarrow \boxed{new(n)} \longrightarrow Ic(Q[n\backslash x]) \;\Big|\; n \notin s\Big\} \bigcup_n Ev(Q[n\backslash x])$$
$$\langle s \rangle \qquad\qquad \langle s \cup \{n\} \rangle$$

$$Ic(out(M).Q) = \quad \{(out(M).Q)\}$$
$$Ev(out(M).Q) = \quad \Big\{(out(M).Q) \longrightarrow \boxed{out(red(M))} \longrightarrow Ic(Q)\Big\} \cup Ev(Q)$$
$$(\!(red(M))\!)$$

$$Ic(in(\Pi).Q) = \quad \{(in(\Pi).Q)\}$$
$$Ev(in(\Pi).Q) = \quad \Big\{(in(\Pi).Q) \longrightarrow \boxed{in(\Pi[\sigma])} \longrightarrow Ic(Q[\sigma]) \;\Big|\; \Pi[\sigma] \in \mathbf{Msg}\Big\} \bigcup_\sigma Ev(Q[\sigma])$$
$$(\!(\Pi[\sigma])\!)$$

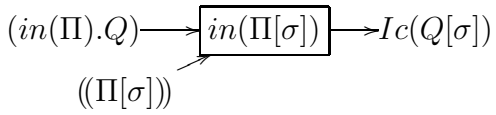
$$Ic([M > \Pi].Q) = \begin{cases} Ic(Q[\sigma]) & \Pi[\sigma] = M \\ \{(nil)\} & \text{otherwise} \end{cases}$$
$$Ev([M > \Pi].Q) = \begin{cases} Ev(Q[\sigma]) & \Pi[\sigma] = M \\ \emptyset & \text{otherwise} \end{cases}$$

$$Ic([M = N].Q) = \begin{cases} Ic(Q) & M = N \\ \{(nil)\} & \text{otherwise} \end{cases}$$
$$Ev([M = N].Q) = \begin{cases} Ev(Q) & M = N \\ \emptyset & \text{otherwise} \end{cases}$$

$$Ic(\|_{i \in I} P_i) = \quad \bigcup_{i \in I} \{(i : P) \mid (P) \in Ic(P_i)\}$$
$$Ev(\|_{i \in I} P_i) = \quad \bigcup_{i \in I} \{tag(i, e) \mid e \in Ev(P_i)\}$$

Figure 11: A Petri net semantics for closed processes

- *In* events: for $\Pi[\sigma] \in \mathbf{Msg}$



  In events require a message, that matches the pattern, to be already present on the network and causes control to evolve.

The set of events of a process P, written $Ev(P)$, will consist of events like the above. It is constructed inductively on the number of process constructors in the process term P. The set of initial control conditions $Ic(P)$ is also built up by induction. Referring to Figure 11, we have:

nil The nil process is described by the empty set of events. In fact no event can happen when a process terminates. A process terminates by marking the control condition (nil).

$new(x).Q$ We add to the events of $Q[n\backslash x]$ *new* events for each name n and set of names s such that $n \notin s$. The initial control condition is as expected. Control is then passed to the components $Ic(Q[n\backslash x])$ of the system in which n is substituted for the bound variable x.

$out(M).Q$ An output event is added to the events of Q.

$in(\Pi).Q$ To the events of the rest of the program, we add *in* events for each message that matches the required pattern.

$[M > \Pi].Q$ We do not associate any explicit events to a *case* construction. If there is no substitution σ matching M with $\Pi[\sigma]$, we take the empty set of events as denotation for *case*. Otherwise it is denoted with the events of the rest of the program.

$[M = N].Q$ *Match* is treated much in the same way as *case*, with a simplification due to the fact that there are no bindings to take care of.

$\|_{i \in I} P_i$ Events belonging to different components are indexed differently. We will write $tag(i, e)$ for the event obtained from e by tagging action, control pre- and postconditions of e with the index i. This makes a disjoint union of events from different components, avoiding confusing them.

We gave a way of constructing a Petri net which describes the behaviour of a process P written in our language. Sometimes we may add more events than necessary in $Ev(P)$. It is important to bear in mind that only the *reachable* ones will contribute to a computation.

If C is a set of conditions let $names(C)$ be the set of names present in the conditions of C defined in the obvious way. Given the definition of Figure 11 the following properties hold:

Proposition 4.8 *Given a closed process Q*

1. $names(Ic(Q)) \subseteq names(Q)$
2. $\forall e \in Ev(Q)\,.\, names(e^{\cdot}) \subseteq \begin{cases} names(^{\cdot}e) \cup (s' \setminus s) & \text{if } \langle s \rangle \in {}^{\cdot}e \text{ and } \langle s' \rangle \in e^{\cdot} \\ names(^{\cdot}e) & \text{otherwise} \end{cases}$

Proof.

1. Follows from the definition of $Ic(Q)$.
2. By well founded induction using the definitions of $Ev(Q)$, and process names (Figure 6) as well as point 1 and Proposition 4.3. We consider two interesting cases, the rest follows similarly. Consider the set $Ev(new(x).Q)$. If $e \in Ev(Q[n\backslash x])$ then the property holds by induction hypothesis and if e is the added event for *new* then

   $$names(Ic(Q[n\backslash x])) \overset{1.}{\subseteq} names(Q[n\backslash x]) \overset{4.3}{\subseteq} names(Q) \cup \{n\} \overset{6}{=} names(new(x).Q) \cup \{n\}$$

   Consider the set $Ev(M > \Pi.Q)$. If this set is empty, then the property is vacuously true, otherwise for all $e \in Ev(Q[\sigma])$ the property holds by induction hypothesis.

□

This property will be useful in showing Lemma 4.10.

We now introduce a notion of computation, expressed by means of reachable markings. The token game on Petri nets adapted to our special kind of nets. It describes a computation, a sequence of moves to a reachable marking. Petri nets, being a non interleaving model for concurrency, can keep track of events occurring concurrently. For the purposes of this work we are mainly concerned with safety properties, thus we will restrict to the case where only one event happens at a time and ignore the independence of events. Define the following transition relation between markings [Win87]:

Definition 4.9 (Token game) *Let P be a closed process and $\mathcal{M}_0$ an initial marking for P. Let $\mathcal{M}, \mathcal{M}'$ be markings and $e \in Ev(P)$ an event. Define $\mathcal{M} \xrightarrow{e} \mathcal{M}'$ if both of the following hold*

1. ${}^{\cdot}e \subseteq \mathcal{M}$
2. $\mathcal{M}' = (\mathcal{M} \setminus^{(\cdot)} e) \cup e^{\cdot}$

A marking $\mathcal{M}$ is reachable, if $\mathcal{M}_0 \xrightarrow{e_0} \cdots \xrightarrow{e_{i-1}} \mathcal{M}_i = \mathcal{M}$.

We would expect a reachable marking to contain a name condition, telling what are the used names up to that marking. In fact the following properties hold:

Lemma 4.10 *Given a closed process P, for every reachable marking $\mathcal{M}_i$, it is the case that:*

1. *There is exactly one name condition $\langle s_i \rangle$ in $\mathcal{M}_i$*
2. *For any two reachable markings $\mathcal{M}_j, \mathcal{M}_i$ in the same sequence of transitions, if $\mathcal{M}_j < \mathcal{M}_i$ then $s_j \subseteq s_i$.*
3. *$names(\mathcal{M}_i) \subseteq s_i$*

Proof. The three properties are all proven by induction on the sequence of markings.

1. The initial marking $\mathcal{M}_0$ contains by definition exactly one name condition. The induction hypothesis together with *new* events being the only ones that consume and mark exactly one name condition, proves the induction step.
2. Observe that *new* events can only add names to the set of used names.
3. The initial marking $\mathcal{M}_0$ satisfies the property by definition. Suppose that the property holds for the i-th marking $\mathcal{M}_i$ in the sequence. By the token game we know that $\mathcal{M}_{i+1} = (\mathcal{M}_i \backslash^{(\cdot)} e_i) \cup e_i^\cdot$, therefore $names(\mathcal{M}_{i+1}) = names(\mathcal{M}_i \backslash^{(\cdot)} e_i) \cup names(e_i^\cdot)$. By induction hypothesis and point 2 it is the case that $names(\mathcal{M}_i \backslash^{(\cdot)} e_i) \subseteq s_{i+1}$. We distinguish two cases:
   (a) If $s_i \in {}^\cdot e_i$ and $s_{i+1} \in e_i^\cdot$ then
   $names(e_i^\cdot) \overset{Prop.4.8}{\subseteq} names({}^\cdot e_i) \cup (s_{i+1} \backslash s_i) \overset{I.H.}{\subseteq} s_i \cup (s_{i+1} \backslash s_i) = s_{i+1}$.
   (b) Otherwise $s_{i+1} = s_i$ and $names(e_i^\cdot) \overset{Prop.4.8}{\subseteq} names({}^\cdot e_i) \overset{I.H.}{\subseteq} s_i = s_{i+1}$.

□

Remark 4.11 The nets we just introduced are similar to safe nets [Win87]. We do not have multiplicities. This simplifies to a treatment with sets instead of multi-sets. The rather unusual conditions we introduced, the "durable" network conditions can be thought of as holding with infinite multiplicity when marked.

The design of our Petri-net semantics suggests it to be equivalent in terms of traces to the operational semantics. This is going to be true of course only if *new* events in a sequence are hidden. Let $Op(P, s_0, t)$ be the set of traces obtained from an initial configuration P, s_0, t by the operational semantics of Section 4.2. Let $Pn(P, \mathcal{M}_0)$ be the set of traces obtained from the initial marking $\mathcal{M}_0$ by the Petri-net semantics. By this we mean the sequences of actions "filtered" out of the sequences of events. Further let $Pn(P, \mathcal{M}_0)_{|in,out}$ be the traces, where the *new* actions are hidden. We expect to prove the following:

Conjecture 4.12 *Given P closed process and $\mathcal{M}_0 = \{\langle s_0 \rangle, (\!(M_1)\!), \ldots, (\!(M_j)\!)\} \cup Ic(P)$ initial marking, then $Op(P, s_0, \{M_1, \ldots, M_j\}) = Pn(P, \mathcal{M}_0)_{|in,out}$.*

4.5 How to treat *new* implicitly

In this section we study how to treat *new* events as implicit events. We want to think of it as a binder rather than as a visible action. This simplifies the reasoning about protocols and allows us to talk about *fresh* events with respect to a name, that identify the first point in a run where that name has been used.

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 gives rise to more then one possible substitution, in fact infinitely many. If the remaining program is a parallel composition of processes and the same variable is bound by *new* in more then one component, then events with different values for that variable can get confused. We will see a concrete example later in this section.

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 4.13 *Our system is a process that does not have a $new(x)$ right 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 program: The program of an agent is a list of prefix actions (see example in Section 4.3). For such processes the previous assumption holds.

As basic events we have *out* and *in* events as before, but no *new* events. Figure 12 shows the change in our previous semantics. The initial control condition remains unchanged. Only the events for $new(x).Q$ change: Take all events of $Q[n\backslash x]$ 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

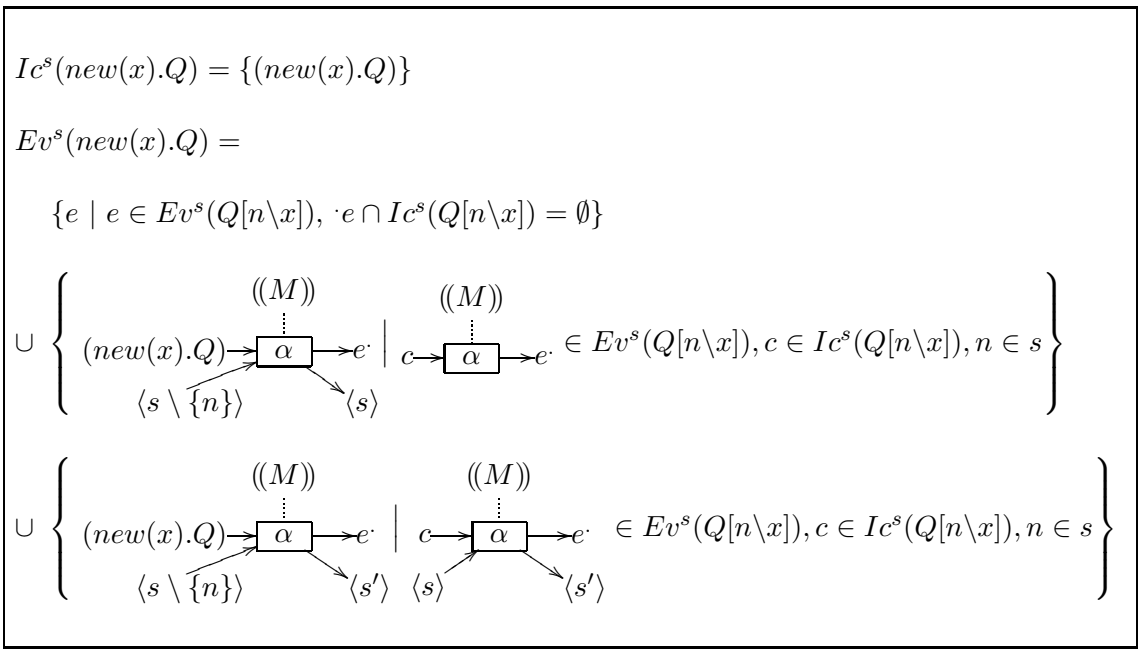


Figure 12: Treating *new* implicitly

in the name condition $\langle s\rangle$ and ask for a pre-condition that does not contain it, i.e., $\langle s \setminus \{n\}\rangle$.

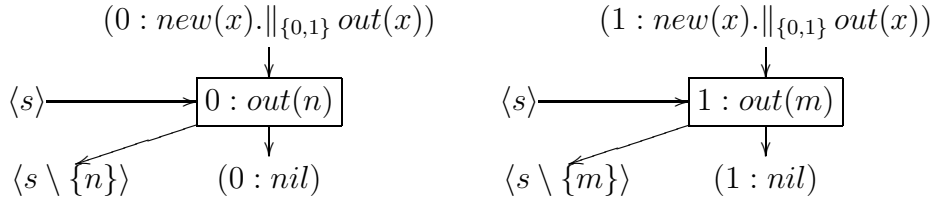
Observation 4.14 This modified semantics still has the properties described in Proposition 4.8, therefore Lemma 4.10 continues to hold also in this case.

The following example illustrates what goes wrong if we do not restrict to straight line programs, even though this assumption could be weakened, as the example suggests.

Example 4.15 Consider the following program, which does not respect Assumption 4.13.

$$new(x).\|_{\{0,1\}}\, out(x)$$

Suppose we treat *New* events implicitly as described above. The events associated to this program are:



It is easy to see that this may give rise to undesired behaviour: There is no keeping-track of the value chosen to be substituted for x in one component and the one chosen to be substituted for x in the other component. As a consequence, different values can be used. Take as initial marking

$$\{(0 : new(x).\|_{\{0,1\}} out(x)), (1 : new(x).\|_{\{0,1\}} out(x)), \langle\emptyset\rangle\}.$$

We clearly can have a trace $0 : out(n)\, 1 : out(m)$ with $n \neq m$ which is not what we want.

Let $Pn^s(P, \mathcal{M}_0)$ be the set of traces from sequences of events obtained from the modified semantics $(Ev^s(P), \mathcal{M}_0)$. We would expect the following:

Conjecture 4.16 Given a closed process P that satisfies Assumption 4.13 and an initial marking $\mathcal{M}_0$, we have $Pn^s(P, \mathcal{M}_0) = Pn(P, \mathcal{M}_0)_{|in,out}$

5 The NSL protocol: authentication and secrecy

In this section we apply our framework to prove authentication and secrecy guarantees for the responder part of the NSL protocol. We use the modified semantics of Section 4.5, where *new* is treated like a binder. We can do this because our NSL system (Figure 10) is a parallel composition of straight line processes. The proof is along the lines of a strand-space argument [THG98c].

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 desired guarantees.

5.1 Definitions and useful principles

We introduce some definitions and principles useful in stating and proving authentication and secrecy of security protocols. In the rest of this section we will make the following convention:

Convention 5.1 *We write $n \in \mathcal{M}$ to mean $n \in s$ where $s \in \mathcal{M}$.*

A *run* of a protocol is a sequence of alternating marking and events, obtained applying the token game (Definition 4.9) starting from the initial marking $\mathcal{M}_0$. We write η 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 η and $\mathcal{M} \in \eta$ if the marking $\mathcal{M}$ occurs in η. If we write $a \in \eta$ then a is either a marking or and an event occurring in η. We use $<_\eta$ for the order relation among markings and events of a partial run η and $\leq_\eta$ for its reflexive closure; so for

instance, $e <_\eta \mathcal{M}$ means that the event e precedes the marking $\mathcal{M}$ in the sequence η. If the partial run is understood from the context we omit the subscript η. Furthermore when we write $a <_\eta a'$ or $a \leq_\eta a'$ we understand that $a, a' \in \eta$. We write $\sqsubset$ for the sub-message relation among messages in reduced form (see [THG98c]), defined as the least binary relation such that for all $M, N \in \mathbf{Msg}$ and for all keys k all the following hold

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

Definition 5.2 *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 η.*

Convention 5.3 *Given a property $\mathcal{P}$ only on markings and η a partial run of a protocol, we may write $\mathcal{P}[\eta]$ for the property $\forall$ 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 5.4 (Well-foundness Principle) *Let η be partial run and let $[a_i \ldots a_j]$, $[a_i \ldots a_h]$ be subintervals of η. Given a property $\mathcal{P}$ on markings and events such that $\mathcal{P}[a_i \ldots a_j]$ and $\neg\mathcal{P}[a_i \ldots a_h]$ where $a_j <_\eta a_h$ then there exists $a_l \in \eta$ where $a_j <_\eta a_l \leq_\eta a_h$ such that $\mathcal{P}[a_i \ldots a_{l-1}]$ and $\neg\mathcal{P}[a_i \ldots a_l]$.* □

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 5.5 *Given an event e, a name $n \in$ **Names** is fresh on e, written $Fresh(n, e)$, if $\langle s \rangle \in {}^{\cdot}e$, $\langle s' \rangle \in e^{\cdot}$, $n \notin s$, 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 5.6 (Freshness Principle) *Let η be a complete run of a protocol, starting from an initial marking $\mathcal{M}_0$.*

1. $\forall \mathcal{M} \in \eta \,.\, \forall n \in \mathcal{M} \,.\, (n \in \mathcal{M}_0) \vee (\exists e \in \eta \,.\, e <_\eta \mathcal{M} \wedge Fresh(n, e))$.
2. *Given $n \in$ **Names**, there exists at most one event $e \in \eta$ s.t. $Fresh(n, e)$.*
3. *Given $e \in \eta, n \in$ **Names** s.t. $Fresh(n, e)$ then $\forall \mathcal{M} \in \eta \,.\, \mathcal{M} <_\eta e \Rightarrow n \notin \mathcal{M}$.*

Proof.

1. Obvious.

2. Suppose the contrary: Given a name $n \in \mathbf{Names}$, there exists e_j, e_h events in η, $e_j \neq e_h$ such that $Fresh(n, e_j)$ and $Fresh(n, e_h)$. Assume $e_j < e_h$. Then by Lemma 4.10 we get a contradiction, since $n \in s'_j$, $n \notin s_j$ and $s'_j \subseteq s_h$. Similarly for $e_h < e_j$.

3. Suppose that there exists $\mathcal{M} < e$ such that $n \in \mathcal{M}$. We get a contradiction from the previous points 1 and 2.

□

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

Principle 5.7 (Precedence Principle) *Given η a run of a protocol from an initial marking $\mathcal{M}_0$ then:*

1. $\forall e \in \eta \,.\, \forall c \in {}^{\cdot}e \,.\, (c \in \mathcal{M}_0) \vee (\exists e' \in \eta \,.\, e' <_\eta e \wedge c \in e'^{\cdot})$.

2. $\forall e \in \eta \,.\, \forall c \in {}^{\cdot}e \,.\, \exists \mathcal{M} \in \eta \,.\, \mathcal{M} <_\eta e \wedge c \in \mathcal{M}$.

3. $\forall \mathcal{M} \in \eta \,.\, \forall c \in \mathcal{M} \,.\, (c \in \mathcal{M}_0) \vee (\exists e \in \eta \,.\, e <_\eta \mathcal{M} \wedge c \in e^{\cdot})$.

Proof. By definition of the token game. □

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. This is what we mean when we refer to the *Output-input Principle*. Moreover, if the Precedence Principle is applied to control conditions, one can determine preceeding events belonging to the same component.

The last principle recalls the token game:

Principle 5.8 (Token Game) *Given $[\mathcal{M}e\mathcal{M}']$ a subinterval of a run of a protocol, then $\mathcal{M}' = (\mathcal{M} \setminus {}^{(\cdot)}e) \cup e^{\cdot}$.* □

5.2 NSL-protocol events

In Section 4.3 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 4.5, with the implicit treatment of *new* as shown in Figure 12. We do not give the initial control conditions $Ic(NSL)$ explicitly. They will be consist of parallel components of the system, indexed accordingly.

In Figure 13 and in Figure 14 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 report we partially omit tagging and use only the indexes that are necessary to make clear 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 () or (i) with i indexing tag, 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 8. Consider for example the initiator $Init(A, B)$. Since it is a straight line program its first set of events share as a control pre-condition the process that is the component itself. These first events are *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 *out* preceeded by a *new*. The next action in the initiator's process is an *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 *case* and a *match*, both constructs will allow us to pass the control to an event only if the message received is of the required form. The properties in Proposition 4.2 ensure that decryption is made only with the right key and there are *in* events leading to *nil* corresponding to unsuccessful applications of *case* and *match*. Finally we have *out* events that wait for a marked control condition coming from the successful tests.

The intuition behind the responder events of Figure 13 and Spy events of Figure 14 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.

An enabled sequence of events from an initial marking, will represent a possible run of the NSL protocol.

5.3 Secrecy of private keys

We show a lemma that will be useful for the proofs of authentication and 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 [THG98c, Pau99]. We recall that Bad is the set of key initially known to be leaked to the intruder.

Lemma 5.9 *Given η a run of NSL, if $Priv(A) \notin Bad$ then the invariance property $I[\eta] \equiv \forall \mathcal{M} \in \eta \,.\, \forall (\!(M)\!) \in \mathcal{M} \,.\, Priv(A) \not\sqsubseteq M$ holds (and analogously for $Priv(B)$).*

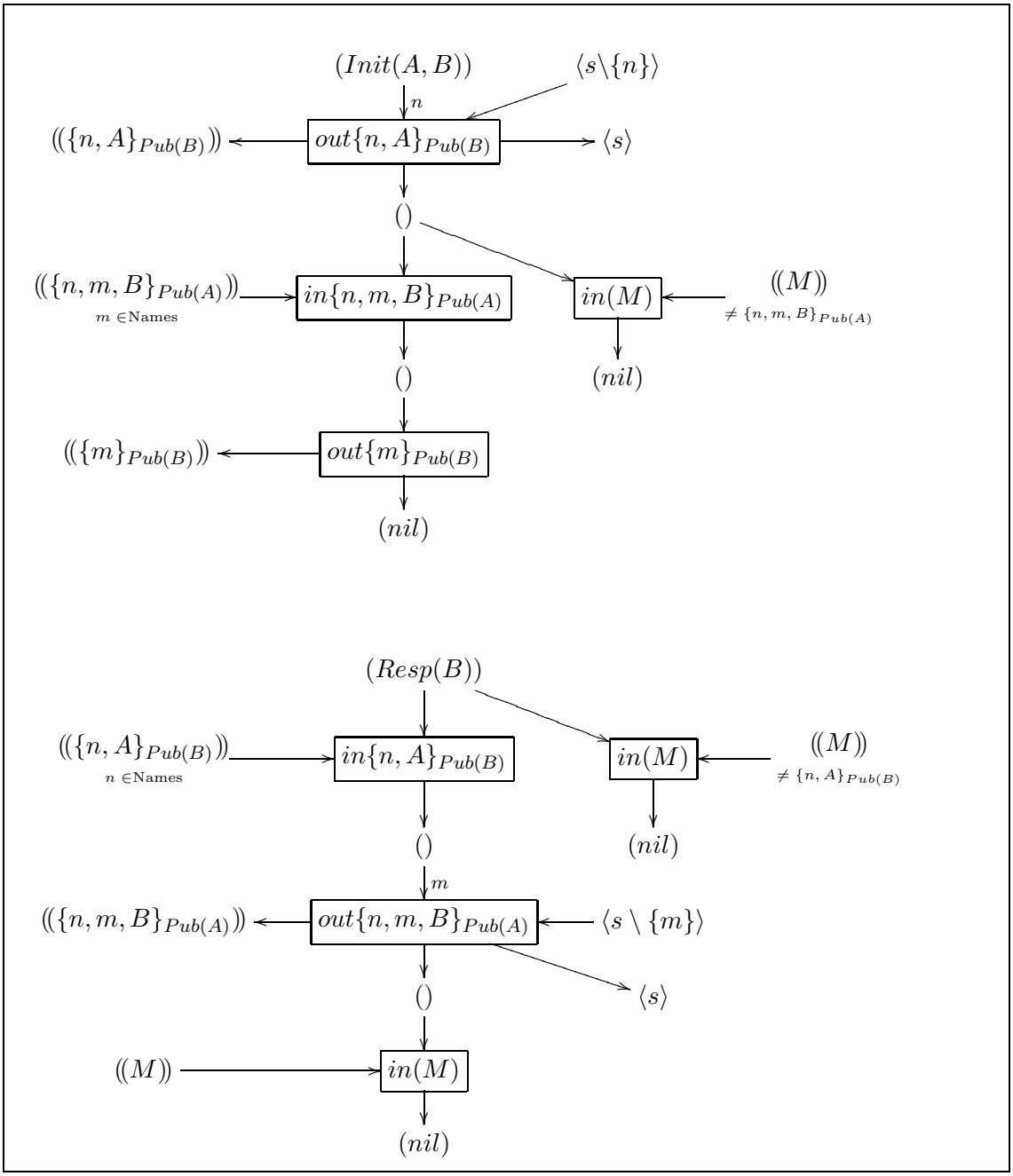


Figure 13: Initiator and responder nets

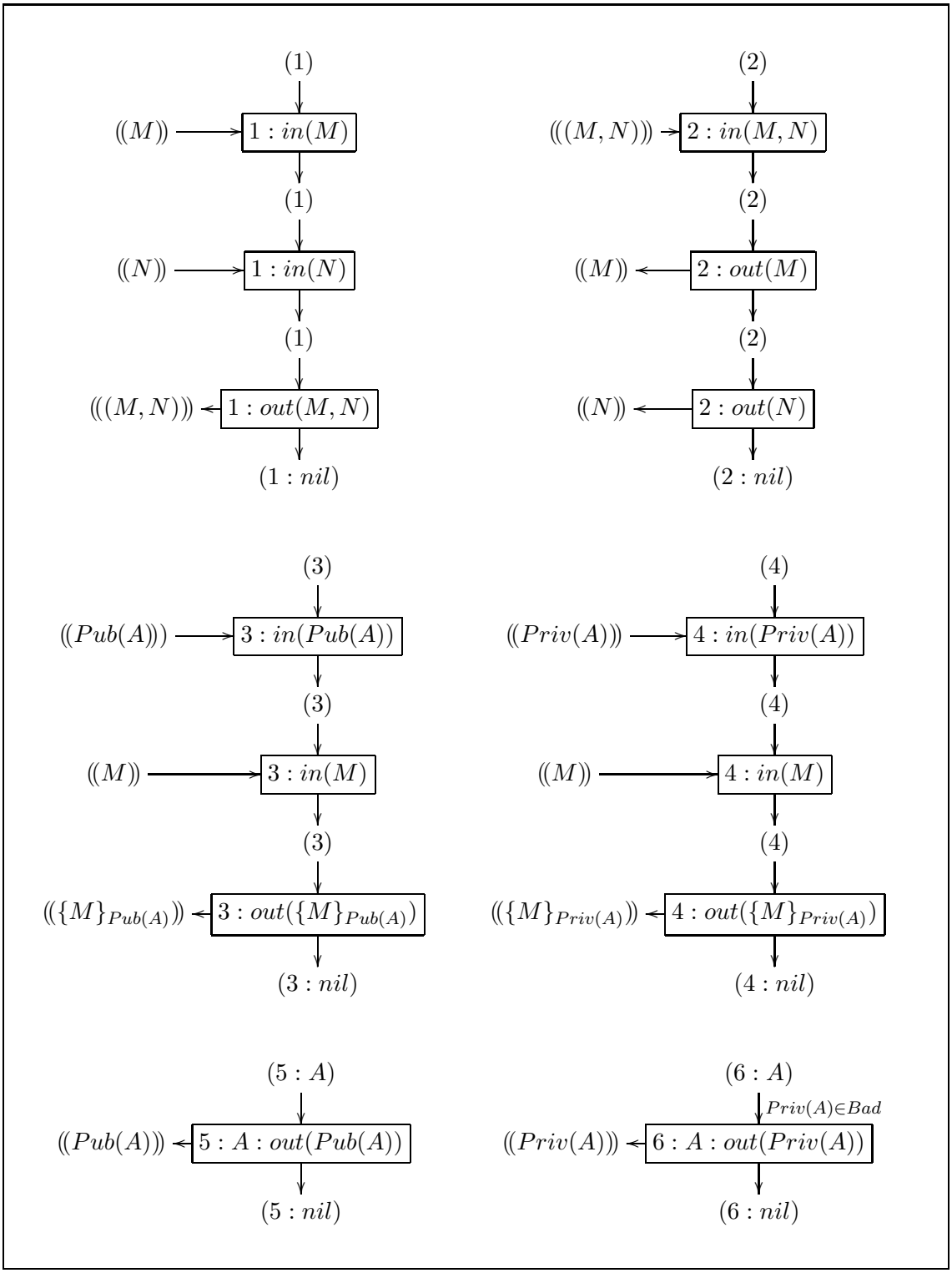
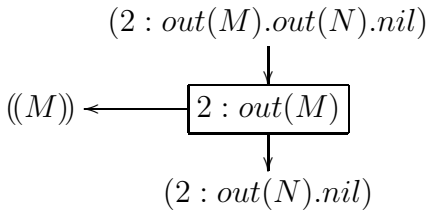


Figure 14: Spy nets

Proof. Let η be a run of NSL starting with the initial marking $\mathcal{M}_0$. It is easy to see that $I[\mathcal{M}_0]$. Suppose that there exists $e \in \eta$ such that $I[\mathcal{M}_0 \dots e]$ and $\neg I[\mathcal{M}_0 \dots e\mathcal{M}]$. We prove that there is no such event e associated to NSL that belongs to the protocol run η. The Token Game 5.8 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 13), 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



with $Priv(A) \sqsubset M$. By the Precedence Principle 5.7 and the events in Figure 14 and Figure 13, there is a marking $\mathcal{M}' < e$ such that $(((M, N))) \in \mathcal{M}'$, which contradicts $I[\mathcal{M}_0 \dots e]$ since $Priv(A) \sqsubset (M, N)$. The reasoning is similar for the other events, bearing in mind that $Priv(A) \notin Bad$. □

5.4 Authentication: the responders guarantee

In the following we will prove authentication for a principle that is responder in an NSL protocol exchange. It is helpful, for the remaining part of this report, to give names to some particular events. Given fixed A, B, n, m we call e_1^A, e_2^A, e_3^A the first, second, and third event (as appearing in the net of Figure 13), where A acts as initiator apparently with B and n is the nonce of A and m the nonce of B. Similarly we will use e_1^B, e_2^B, e_3^B for the events of B responding to A using nonces n, m.

Partial runs support a form of diagrammatic reasoning. When the partial run η is understood we draw

$$e \longrightarrow e'$$

to mean $e \leq_\eta e'$. We may annotate the arrows with the principle used in showing the corresponding precedence relation. We also draw

$$e \overset{Inv}{\Longrightarrow} e'$$

to indicate that $Inv[e \dots e']$ holds, where Inv is an invariance property. Such diagrams help us to follow the various steps of a proof of authentication.

Theorem 5.10 (Authentication) *If $Priv(A) \notin Bad$ then for every run η of NSL, if $e_1^B < e_2^B < e_3^B$, then $e_1^A < e_1^B < e_2^B < e_2^A < e_3^A < e_3^B$ in η.*

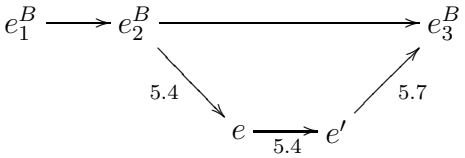
Proof. Let η' be a partial run of NSL. Let us consider the following invariance property:

$$Inv[\eta'] \equiv \forall \mathcal{M} \in \eta' \,.\, \forall (\!(M)\!) \in \mathcal{M} \,.\, m \sqsubset M \Rightarrow \{n, m, B\}_{Pub(A)} \sqsubset M$$

Let η be a run of NSL, starting with $\mathcal{M}_0$, such that $e_1^A < e_1^B < e_2^B$. It is easy to see that for $[e_2^B \mathcal{M} e]$ a subinterval of η, we have that $Inv[e_2^B \mathcal{M} e]$ holds. This follows from $Fresh(m, e_2^B)$ and from the Freshness Principle 5.6 (3), together with Lemma 4.10, and the Token Game 5.8. Since $\neg Inv[e_2^B \ldots e_3^B]$, we choose the event $e \in \eta$ for which $[e_2^B \ldots e]$ is the maximum subinterval of η such that $Inv[e_2^B \ldots e]$ (Well-foundness Principle 5.4). Part of the proof of authentication is showing $e = e_3^A$. Initially we know

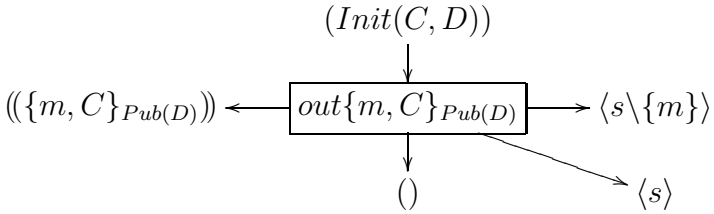
$$e_1^B \longrightarrow e_2^B \longrightarrow e_3^B \,.$$

Since $(\!(\{m\}_{Pub(B)})\!) \in {}^{\cdot}e_3^B$ and $m \notin \mathcal{M}_0$ by the Output-input Principle 5.7 there must be $e' < e_3^B$ such that $(\!(\{m\}_{Pub(B)})\!) \in e'^{\cdot}$. Since $Inv[e_2^B \ldots e]$ and $\neg Inv[e_2^B \ldots e' \mathcal{M}]$, it is $e \leq e'$:



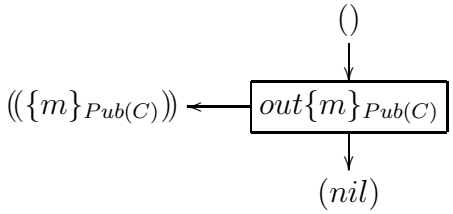
We check the events associated to the NSL protocol and show $e = e_3^A$. By the Token Game 5.8 we need to concentrate on output events only. Let us look at the initiator events first. Since e has to "produce" a marking that violates the invariant, we are interested only in events that mark messages containing m and do not have $\{n, m, A\}_{Pub(B)}$ as sub-message. We distinguish the following cases:

The case $e \equiv$

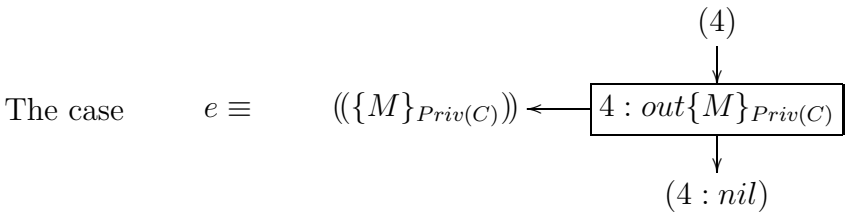


This case is excluded by the Freshness Principle 5.6, since we would have $Fresh(m, e)$ and $e \neq e_2^B$.

The case $e \equiv$



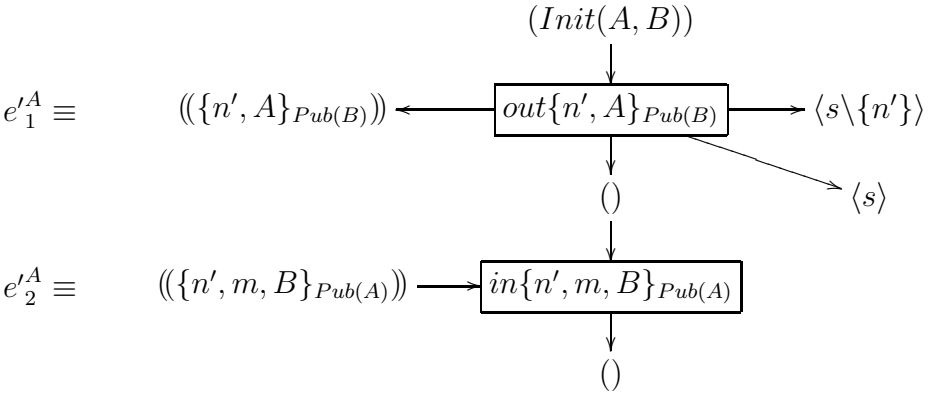
This case is excluded if $C \neq B$, since by the Precedence Principle 5.7 and Input-Output Principle 5.7 it contradicts $Inv[e_2^B \dots e]$. Instead if $C = B$ the event e can occur in the sequence only if it is an event of $Init(A, B)$ i.e. $e = e_3^A$, otherwise we reach a contradiction as before. The responder cases are similar to the ones we just saw, and so are most of the spy cases. We just look at the most interesting case for the spy, which shows the importance of the assumption $Priv(A) \notin Bad$. Consider



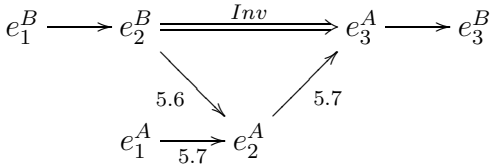
with $m \sqsubset \{M\}_{Priv(C)}$ and $\{n, m, B\}_{Pub(A)} \not\sqsubset \{M\}_{Priv(C)}$. It is the case that $C \neq A$. In fact $Priv(A) \notin Bad$ thus the Precedence Principle 5.7 and Lemma 5.9 apply. Since decryption can happen only with the right key (Proposition 4.2) and $\{n, m, B\}_{Pub(A)} \not\sqsubset \{M\}_{Priv(C)}$ it is $\{n, m, B\}_{Pub(A)} \not\sqsubset M$. Again the Precedence Principle 5.7 tells us that there is a preceeding marking containing $(\!(M)\!)$, which contradicts $Inv[e_2^B \dots e]$. At this point we know:

$$e_1^B \longrightarrow e_2^B \overset{Inv}{\Longrightarrow} e_3^A \longrightarrow e_3^B$$

In η there has to be events ${e'}_1^A, {e'}_2^A$ preceeding e_3^A (Precedence Principle 5.7) such that:



for some name n'. Furthermore $e_2^B < {e'}_2^A$ (Freshness Principle 5.6) and since $Inv[e_2^B \dots e_3^A]$, it is the case that $n' = n$, thus ${e'}_2^A = e_2^A$ and ${e'}_1^A = e_1^A$:



Since $Fresh(n, e_1^A)$ we have $e_1^A < e_1^B$ (Freshness Principle 5.6), thus:

$$e_1^A \xrightarrow{5.6} e_1^B \longrightarrow e_2^B \longrightarrow e_2^A \longrightarrow e_3^A \longrightarrow e_3^B$$

□

5.5 Secrecy of the responder's nonce

We state and proof secrecy for a nonce of a principal which uses the NSL protocol as a responder.

Theorem 5.11 (Secrecy) *If $Priv(A) \notin Bad$ and $Priv(B) \notin Bad$ then for every run η of NSL such that $e_2^B \in \eta$, the invariant $Sec[\eta] \equiv \forall \mathcal{M} \in \eta \,.(\!(m)\!) \notin \mathcal{M}$ holds.*

Before proving this, we observe that if we allow either the private key of the responder or the private key of the initiator to be leaked, the theorem does not hold. For example if we have $Priv(A) \in Bad$, i.e., the initiator A is corrupted, then there is an obvious attack as the following action trace for NSL shows:

$$\begin{array}{l} 6 : A : out(Priv(A)) \\ A : B : out\{n, A\}_{Pub(B)} \\ B : in\{n, A\}_{Pub(B)} \\ B : out\{n, m, B\}_{Pub(A)} \\ 4 : in(Priv(A)) \\ 4 : in\{n, m, B\}_{Pub(A)} \\ 4 : out(n, m, B) \\ 3 : in(n, m, B) \\ 3 : out(n) \end{array}$$

To prove the stated secrecy theorem, we show a stronger result, following the proof strategy taken in [THG98c].

Proof. We will prove a stronger invariant:

$$Sec'[\eta] \equiv \forall \mathcal{M} \in \eta \,. \forall (\!(M)\!) \in \mathcal{M} \,. m \sqsubset M \Rightarrow \{n, m, B\}_{Pub(A)} \sqsubset M \vee \{m\}_{Pub(B)} \sqsubset M$$

It is easy to see (Freshness Principle 5.6) that $Sec'[\mathcal{M}_0 \ldots e_2^B \mathcal{M}]$. Suppose there exists an event e such that $Sec'[\mathcal{M}_0 \ldots e]$ and $\neg Sec'[\mathcal{M}_0 \ldots e\mathcal{M}']$. We look for the various possibilities for e. By the Token Game 5.8 we have just to inspect the output events. Using the principles in a similar way as done for authentication, we can easily see that no initiator, responder, or spy event can produce a marking $\mathcal{M}'$ such that $\neg Sec'[\mathcal{M}_0 \ldots e\mathcal{M}']$. □

We saw how to state and prove secrecy and authentication guarantees for the responder part of an NSL protocol. The initiator guarantees can be stated and proven very much in the same way.

6 Conclusions and future work

In this report we introduced a new language for crypto-protocols, and showed an operational, as well as a Petri-net semantics. We showed how to prove secrecy for data that is supposed to stay confidential during the exchange over an insecure medium. We borrow proof techniques from the strand-space approach and the inductive approach. We saw how Petri nets provide a natural denotation for security protocols. The model can help unify a variety of approaches, from 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. We now outline some possible directions of future work.

6.1 Proof methods

Logics. As a first logic approach, the BAN logic [BAN89] showed how to prove properties about crypto-protocols at a higher level of abstraction. Although it may be useful to reason about freshness of nonces and keys, it is inadequate to reason about secrecy and other properties. As a future direction we plan to explore ways of giving a logic based on our language. We think it could be a logic on sequences of alternating markings and events, that are the runs of a protocol, and expect it to be useful in proving both authentication and secrecy. Part of the reasoning when proving secrecy or authentication involves showing an invariance property on a partial run. Often reasoning is based on the principles listed in Section 5 which could constitute building blocks for a logic. We wish the logic to interact closely with the program representing an interacting system, so that a case analysis on events can be done by looking at the program. At the moment we don't know if an existing logic suffices. The BAN logic lacks an operationally based semantics. Possibly sequences of markings and events could form the basis of an informative model for checking soundness of principles of BAN logic. Recently Fiore has developed a model checking algorithm for a language very like the one here. Clearly model checking is an area for further investigation.

Open-ended protocols. Most formal methods for analysing security protocols seem to restrict to closed systems, with a fixed number of agents and sometimes only a fixed number of protocol runs. An emerging issue as discussed in [Mea00] is the study of open-ended systems. The strand-space approach and the inductive approach which are both strongly related to the work presented in this report, are moving in this direction. If we look at the proof of secrecy of the previous section, we did not make any assumption on the number of agents and runs of the protocol. It seems that the methods that we are developing support open-ended systems to some extent. If we had to automatise proof methods, we can perhaps

hope to tune data independence techniques [Ros98a] to our language: In the case of the NSL protocol, intuition seems to suggest that a property proven for a small system, with two trusted agents, a corrupted agent and the spy, will scale to an arbitrary large system. Another technique that might come to hand is abstract interpretation [CC79]. Often a system that communicates following a protocol, is a parallel composition of components, for each participant in the protocol. Usually the programs of the participants are the same process. In a situation of this kind, there might be a way of relating a set of event sequences of arbitrary system with the one of an abstract "small" system in such a way that an abstract security property implies a security property of the system we want to verify. An other interesting problem, is to study how a protocol combines with other protocols. Work on the strand-space model [THG99] done in this direction may give us some suggestions.

Beyond safety. In this report we concentrated on security properties being safety properties. It is a point of discussion, whether restricting to properties on traces is enough to express properties of protocols. Even if it seems to be so for security and authentication, it may not be the case for others, e.g. non-interference, mentioned earlier in this report. A process language like the one introduced seems to ask for a notion of process equivalence. Defining it and studying security properties that can be defined and proven using such equivalence can be a future direction.

6.2 Language extensions and implementation

Modelling probabilistic polynomial time processes. The Dolev-Yao model we looked at, and other models on which formal methods are based for analysing security protocols, take cryptography as a black box. This is a serious restriction of the intruder's capabilities: In practice protocols that are proven correct by formal methods may allow cryptographic attacks. In practice the attacker, even if not able to recover the complete clear-text from a ciphertext, may learn partial information from eavesdropped messages, which might lead to a later attack. An interesting direction is therefore to study ways of removing the black box assumption for cryptography. On the other hand, doing so asks for an other bound on the adversary, and the agents in general. In fact we may want to look at them as probabilistic polynomial time bounded. This would prevent adversaries with unbounded computing power that could just compute all clear-text from the ciphertext. Some existing work in this direction [LMMS98a, LMMS98b, LMMS99] is based on the asynchronous π-calculus [Bou91]. There is the issue of how to give a probabilistic semantics to our language.

Time-stamps. Nonces and time-stamps are an important part of the machinery used in security protocols to achieve guarantees such as authentication and secrecy.

We have seen how to treat nonces by new name generation, following the assumption that random numbers are unguessable. Protocols that incorporate time stamps seem to become more and more interesting not only because they may be useful in providing user authentication; in many cases it is important to certify the date and time a key, or more general a document was issued. Agents that get time-stamped messages or documents may want to check them to be recent, with respect to an accessible clock, or simply to prove that they where issued after a certain point in time. Often the concept of relative time comes into the picture. Desired properties of time stamps are [HS91]: "First they should not relay on the medium on which the data appears, and so avoid the possibility of changing the document, without the change being evident. Second it should be impossible to stamp a document with a time and date different from the actual one." At this point one might ask: What are the assumptions and the extensions of the calculus, in order to allow the treatment of time stamps?

Implementation of the calculus. Recently a flaw in an implementation of the SSL protocol in one of the main browsers has been found [ACR00]:

> "The flaw we have found effectively disables one of the two basic SSL functionalities: to assure users that they are really communicating with the intended web server - and not with a fake one. Using this flaw, the attacker can make users send secret information (like credit card data and passwords) to his web server rather than the real one - even if the communication is protected by SSL protocol."

Although the SSL protocol has been proven to substantially guarantee authentication, e.g.,[MSS99], this example shows how an inaccurate implementation may be target of serious attacks. We think that an implementation of our language could help to bridge the gap between informal description of a protocol and its implementation. Assuming a correct implementation of the language, guarantees that can be proven are already valid properties of an implemented protocol.

6.3 Models

The models strand spaces, the inductive rules of Paulson, and Petri nets are closely related. The inductive method is couched in terms of rules and this has some consequence such as "stuttering" in the traces generated (see [Pau98, Pau99]), although they appear harmless in establishing safety properties. In Petri-net terms Paulson's rules correspond to events and provide a description of "flat" nets in which certain events lack control preconditions, and so can occur repeatedly. Strand spaces are closely related to event structures [Win86], another model emphasising the causal dependencies between events (more precisely, the bundles of a strand space, ordered

by inclusion form a stable family [Win86] and so are isomorphic to the configurations of an event structure). There are well-known relations between Petri nets and event structures (a further part to the story of [CDL+00, CDL+99]). Incidentally, event structures can express non-deterministic branching not supported by strand spaces (for example, branching of a strand), though this greater detail carries no gain when analysing for safety properties.

References

[Aba98] M. Abadi. Two facets of authentication. In *Proceedings of the 11th IEEE Computer Security Foundations Workshop*, pages 27–32, 1998.

[Aba99] M. Abadi. Security protocols and specifications. In *Foundations of Software Science and Computation Structures: Second International Conference, FOSSACS '99*, pages 1–13, 1999.

[ACR00] ACROS. Bypassing warnings for invalid SSL certificates in Netscape Navigator. ACROS Security Problem Report, april 2000. http://www.cert.org/advisories/CA-2000-05.html.

[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.

[AN96] M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, 1996.

[BAN89] M. Burrows, M. Abadi, and R. M. Needham. A logic of authentication. *Proc. R. Soc. Lond.*, A 426:233–271, 1989.

[Bou91] G. Boudol. Asynchrony and the pi-calculus. Technical Report RR-1702, Inria, Institut National de Recherche en Informatique et en Automatique, 1991.

[Can00] R. Canetti. Security and composition of multiparty cryptographic protocols. *J. Cryptology*, 13:143–202, 2000.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979.

[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.

[DY83] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 2(29), 1983.

[EG83] S. Even and O. Goldreich. On the security of multi-party ping-pong protocols. In *Proceedings of the 24th Symposium on Foundations of Computer Science*. IEEE Computer Society Press, November 1983.

[Gel85] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[HS91] S. Haber and W. S. Stornetta. How to time-stamp a digital document. *Journal of Cryptology*, 3(2):99–111, 1991.

[LMMS98a] P. D. Lincoln, J. C. Mitchell, M. Mitchell, and A. Scedrow. A linguistic characterization of bounded oracle computation and probabilistic polynomial time. In *IEEE Foundations of Computer Science*, 1998.

[LMMS98b] P. D. Lincoln, J. C. Mitchell, M. Mitchell, and A. Scedrow. A probabilistic poly-time framework for protocol analysis. In *ACM Computer and Communication Security (CCS-5)*, 1998.

[LMMS99] P. D. Lincoln, J. C. Mitchell, M. Mitchell, and A. Scedrow. Probabilistic polynomial-time equivalence and security protocols. In *FM'99 World Congress On Formal Methods in the Development of Computing Systems*, 1999.

[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.

[Low97] G. Lowe. A hierarchy of authentication specifications. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop*, pages 31–43, 1997.

[Mea00] C. Meadows. Open issues in formal methods for cryptographic protocol analysis. In *Proceedings of DISCEX 2000*. IEEE Computer Society Press, January 2000.

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

[MMS97] J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using murphi. In *1997 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 1997.

[MSS99] J. C. Mitchell, V. Shmatikov, and U. Stern. Finite-state analysis of SSL 3.0. In *Seventh USENIX Security Symposium*, pages 201–216, San Antonio, 199.

[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.

[Pau] L. C. Paulson. Relations between secrets: Two formal analyses of the Yahalom protocol. *JCS*. in press.

[Pau94] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer-Verlag, 1994.

[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.

[Ros95] A. W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *In 8th IEEE Computer Security Foundations Workshop*, pages 98–107. IEEE Computer Society, 1995.

[Ros98a] A. W. Roscoe. Proving security protocols with model checkers by data independence techniques. In *Proceedings of the 11th Computer Security Foundations Workshop*, pages 84–95. IEEE Computer Society Press, 1998.

[Ros98b] A. W. Roscoe. *The Theory and Practice of Concurrency*, chapter 15.3, pages 446–464. Prentice Hall, 1998.

[RS99] P. Y. A. Ryan and S. A. Schneider. Process algebra and non-interference. In *12th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1999.

[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.

[Sch96] S. Schneider. Security properties and CSP. In *Symposium on Security and Privacy*, pages 174–187. IEEE Computer Society, 1996.

[Son99] D. Song. Athena: An automatic checker for security protocols. In *Proceedings of the 12th IEEE Computer Security Foundation workshop*. IEEE Computer Society Press, June 1999.

[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.

[THG99] J. Thayer, J. Herzog, and J. Guttman. Mixed strand spaces. In *Proceedings of the 12th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1999.

[Win86] G. Winskel. *Event Structures*, volume 255 of *LNCS*, pages 325–392. Springer-Verlag, 1986.

[Win87] G. Winskel. Petri nets, algebras, morphisms, and compositionality. *Information and Computation*, 72:197–238, 1987.

Recent BRICS Report Series Publications

RS-00-18 **Federico Crazzolara and Glynn Winskel.** ***Language, Semantics, and Methods for Cryptographic Protocols*****. August 2000. ii+42 pp.**

RS-00-17 **Thomas S. Hune.** ***Modeling a Language for Embedded Systems in Timed Automata*****. August 2000. 26 pp. Earlier version entitled** ***Modelling a Real-Time Language*** **appeared in Gnesi and Latella, editors,** ***Fourth International ERCIM Workshop on Formal Methods for Industrial Critical Systems,*** **FMICS '99 Proceedings of the FLoC Workshop, 1999, pages 259–282.**

RS-00-16 **Jiří Srba.** ***Complexity of Weak Bisimilarity and Regularity for BPA and BPP*****. June 2000. 20 pp. To appear in Aceto and Victor, editors,** ***Expressiveness in Concurrency: Fifth International Workshop EXPRESS '00 Proceedings,*** **ENTCS, 2000.**

RS-00-15 **Daniel Damian and Olivier Danvy.** ***Syntactic Accidents in Program Analysis: On the Impact of the CPS Transformation*****. June 2000. Extended version of an article to appear in** ***Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming,*** **2000.**

RS-00-14 **Ronald Cramer, Ivan B. Damgård, and Jesper Buus Nielsen.** ***Multiparty Computation from Threshold Homomorphic Encryption*****. June 2000. ii+38 pp.**

RS-00-13 **Ondřej Klíma and Jiří Srba.** ***Matching Modulo Associativity and Idempotency is NP-Complete*****. June 2000. 19 pp. To appear in** ***Mathematical Foundations of Computer Science: 25th International Symposium,*** **MFCS '00 Proceedings, LNCS, 2000.**

RS-00-12 **Ulrich Kohlenbach.** ***Intuitionistic Choice and Restricted Classical Logic*****. May 2000. 9 pp.**

RS-00-11 **Jakob Pagter.** ***On Ajtai's Lower Bound Technique for R-way Branching Programs and the Hamming Distance Problem*****. May 2000. 18 pp.**

RS-00-10 **Stefan Dantchev and Søren Riis.** ***A Tough Nut for Tree Resolution*****. May 2000. 13 pp.**

RS-00-9 **Ulrich Kohlenbach.** ***Effective Uniform Bounds on the Krasnoselski-Mann Iteration*****. May 2000. 34 pp.**