Language, Semantics, and Methods for Security Protocols

Federico Crazzolara

BRICS Dissertation Series DS-03-4

ISSN 1396-7002 May 2003

See back inner page for a list of recent BRICS Dissertation 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 `DS/03/4/`

Language, Semantics, and Methods for Security Protocols

Federico Crazzolara

PhD Dissertation

Department of Computer Science
University of Aarhus
Denmark

Language, Semantics, and Methods for Security Protocols

A dissertation
presented to the Faculty of Science
of the University of Aarhus
in partial fulfillment of the requirements for the
PhD degree.

by
Federico Crazzolara
February 3, 2003

Acknowledgements

I am extremely grateful to my supervisor Glynn Winskel. This work would have not been possible without his guidance. Glynn is co-author of much of this research; his many enlightening suggestions and the numerous discussions were a precious help in overcoming difficulties and in improving many of the results discussed in this thesis.

I would like to thank the director of BRICS, Mogens Nielsen, and the administrative staff at DAIMI. They have always been very helpful in finding solutions to the many practical problems that I encountered during my journey as a PhD student. Thanks also to the administrative staff of the Computer Lab at Cambridge University for making my long visit to the lab possible and very pleasant.

Special thanks go to my colleagues, fellow PhD students at BRICS. I found great friends with whom I shared more than one adventure over the last few years.

I am grateful to Roberto Giacobazzi for his friendship and his encouragement, specially during my first steps as a PhD student.

My deepest gratitude goes to my father Benedetto and my mother Gusta for all their support. Thanks to my sister Ruth for helping my parents at home and making my living abroad possible.

A mi pêr y mia uma

Abstract

Security protocols help in establishing secure channels between communicating systems. Great care needs therefore to be taken in developing and implementing robust protocols. The complexity of security-protocol interactions can hide, however, security weaknesses that only a formal analysis can reveal. The last few years have seen the emergence of successful intensional, event-based, formal approaches to reasoning about security protocols. The methods are concerned with reasoning about the events that a security protocol can perform, and make use of a causal dependency that exists between events. Methods like strand spaces and the inductive method of Paulson have been designed to support an intensional, event-based, style of reasoning. These methods have successfully tackled a number of protocols though in an *ad hoc* fashion. They make an informal spring from a protocol to its representation and do not address how to build up protocol representations in a compositional fashion.

This dissertation presents a new, event-based approach to reasoning about security protocols. We seek a broader class of models to show how event-based models can be structured in a compositional way and so used to give a formal semantics to security protocols which supports proofs of their correctness. More precisely, we give a compositional event-based semantics to an economical, but expressive, language for describing security protocols (SPL); so the events and dependency of a wide range of protocols are determined once and for all. The net semantics allows the derivation of general properties and proof principles the use of which is demonstrated in establishing security properties for a number of protocols. The NSL public-key protocol, the ISO 5-pass authentication and the Π_3 key-translation protocols are analysed for their level of secrecy and authentication and for their robustness in runs where some session-keys are compromised. Particularly useful in the analysis are general results that show which events of a protocol run can not be those of a spy.

The net semantics of SPL is formally related to a transition semantics which can easily be implemented. (An existing SPL implementation bridges the gap that often exists between abstract protocol models on which verification of security properties is carried out and the implemented protocol.) SPL-nets are a particular kind of contextual Petri-nets. They have persistent conditions and as we show in this thesis, unfold under reasonable assumptions to a more basic kind of nets. We relate SPL-nets to strand spaces and inductive rules, as well as trace languages and event structures so unifying a range of approaches, as well as providing conditions under which particular, more limited, models are adequate for the analysis of protocols. The relations that are described in this thesis help integrate process calculi and algebraic reasoning with event-based methods to obtain a more accurate analysis of security properties of protocols.

Contents

Chapter 1

Introduction

Regardless of who is involved, to one degree or another, all parties to a transaction must have confidence that certain objectives associated with information security have been met. [Handbook of applied cryptography]

The security of information has always been an issue independently of the medium through which information is conveyed. Nowadays information has prevalently an electronic form and is transmitted over computer networks. Copying and altering information therefore becomes easier and security methodologies gain greater importance. Information security can have several objectives, which depend among other factors from the importance of the information, the hostility of the communication environment and the cost of security measures. The parties of a transaction may, for example, want that the transmitted information stays secret from all but those who are authorised. The impossibility of altering information without the alteration being detected is sometimes of great importance and also the ability of corroborating the source of information. To achieve these and other objectives security methods often use cryptography.

Several models have been developed to evaluate security methods and gain confidence that security objectives are achieved. Usually these models include an adversary, a malicious user that tries to tamper security. The adversary can be passive and only listen to network traffic with the hope of getting hold of confidential information. Active adversaries, instead, participate actively in the communication for example through impersonation of agents and manipulation of messages. Often the level of security has to be such that methods are correct even if an active adversary is involved, however the computational power of the adversary need not be unlimited. An information-theoretic measure of security assumes an adversary with unlimited computational power. This measure expresses whether or not the system has unconditional security. A complexity theoretic measure instead assumes an adversary with only polynomial computational power. Yet another measure of security is obtained when one restricts even more the abilities of the adversary so that it can't break cryptographic primitives and can't guess random numbers. In this way one can express the level of security of the sole interactions between agents that communicate over a computer network through a security protocol.

Security protocols are sequences of messages that describe how users can communicate securely over a network. Although simple in appearance, security protocols involve complicated interactions. Unfortunately it is rather common that protocols have security breaches even if informal security arguments support opposite claims. A formal approach to the analysis of security protocols helps to determine the precise conditions under which properties of a protocol hold or to reveal possible attacks to a protocol's security. Frequently the assumption that cryptography is unbreakable simplifies formal reasoning about protocols and their properties. The restrictions and capabilities of the adversary which are most common for a formal methods approach to security protocol analysis are those originally described by Dolev and Yao [28].

1.1 Security protocols

Security protocols describe a way of exchanging data over an untrusted medium so that, for example, data is not leaked and authentication between the participants in the protocol is guaranteed or, instead, communication is anonymous. A protocol is often described as a sequence of messages, and usually encryption is used to achieve security goals.

As a running example, throughout the first chapters of this thesis we consider the Needham-Schröder-Lowe (NSL) protocol:

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

This protocol, like many others of its kind, has two roles: one for the initiator, here played by agent A (say Alice), and one for the responder, here B (Bob). It is a public-key protocol that assumes an underlying public-key infrastructure, such as RSA [67]. Both agents have their own, secret private key. Public keys in contrast are available to all participants in the protocol. The NSL protocol makes use of *nonces* which one can think of as newly generated, unguessable numbers whose purpose is to ensure the freshness of messages. As is common practise, if $Pub(A)$ is the public key of an agent A and M a message, we indicate with $\{M\}_{Pub(A)}$ the cyphertext obtained by encrypting M with the key $Pub(A)$.

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

The NSL protocol is very common in practise, where it is part of a longer message-exchange sequence; usually, after initiator and responder complete a protocol exchange, they will continue communication possibly using the exchanged nonces to establish a session key. The NSL protocol aims at distributing nonces m and n in a secure way, so that only initiator and responder know their value. (*secrecy*). Another aim of the protocol is, for example, to guarantee Bob that m is indeed the nonce sent by Alice (*authentication*).

Formalisation of protocols

Although in the informal explanation of the NSL protocol only two agents in their respective roles are described, the protocol is really a shorthand for a situation of a network of distributed agents, each able to participate in multiple concurrent sessions as both initiator and responder. There is no assurance that they all stick to the protocol, or indeed that communication goes to the intended agent. An attacker might dissemble and pretend to be one or several agents, taking advantage of any leaked keys it possesses in deciphering and preparing the messages it sends. As experience shows, even if protocols are designed carefully, following established criteria [6], they may contain flaws. To be useful protocols in fact involve many concurrent runs among a set of distributed users. Then, for example, the NSL protocol is prone to a "middle-man" attack violating both secrecy and authentication, if, as in the original protocol, the name B is not included in the second message – this weakness was first pointed out by Lowe [47]. Formal analysis of security protocols can both help to prove protocols correct with respect to a chosen model and to discover flaws.

Formalisation of security properties

There is no common agreement on how to formalise the meaning of security. Properties such as secrecy or authentication can be defined in a variety of ways [2, 3, 5, 48, 71, 74]. A property, even the formulation of a property, may be more interesting than another, depending on the protocol and the purpose for which the protocol was designed. To avoid misunderstandings, when analysing a protocol it is important to state precisely the properties that are proved 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 properties holding of finite histories. In this thesis 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 [61] and in the strand-space approach [90] and was already used by Dolev and Yao [28]. A common definition of authentication is the agreement property defined in Lowe [48]:

> "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."

Sometimes one requires in addition that a run of A corresponds to a unique "agreeing" run of B.

There are other security properties one would like to prove about protocols, such as integrity, anonymity, non-repudiation, etc. Some of them, like for example non-interference, cannot be expressed as safety properties. Non interference provides a different definition of secrecy and is often used in process-algebra approaches [5, 72]. It can be expressed by means of a process equivalence [3]:

> "Given a process $P(x)$ with only free variable x, P preserves the 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 thesis. However we do propose a process language for reasoning about protocols which might be helpful in investigating differences between various security-definition styles.

The Dolev Yao model

The assumptions of the Dolev-Yao model are commonly used in modelling security protocols to simplify reasoning about security properties. Originally introduced by Dolev and Yao [28], their model underlies a variety of approaches, e.g., [47, 55, 61, 71, 74, 90]. The basic assumptions of the model are:

- Cryptography is treated as a black box; 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 sent over the communication medium but can also modify, replay and suppress messages, and even participate in the protocol, masquerading as a trusted user.

In the Dolev-Yao model the question

"Is a given protocol secure?"

is known to be decidable for a very special kind of protocols, namely cascade protocols and name-stamp protocols [28]. In that context security is essentially secrecy and the class of protocols for which the security question is decidable is rather restricted – current protocols do not belong to it. Security is undecidable for a broader class of protocols [30].

1.2 Events for security protocols

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

Among others, the method of strand spaces [90] and the inductive method of Paulson [62] have been designed to support such an intensional, event-based, style of reasoning. Strand spaces are based on an explicit causal dependency of events, whereas in Paulson's method the dependency is implicit in the inductive rules, which might express, for instance, that the input of a message depends on its previous output. Both methods have successfully tackled a number of protocols though in an *ad hoc* fashion. Both make an informal spring from a protocol to its representation, as either a strand space or a set of inductive rules. Both methods do not address how to build up their representation of a protocol in a compositional fashion.

1.3 Motivations

The strand-space approach to security protocols and the inductive approach of Paulson have proved to be useful models for the analysis of security protocols. In particular the strand spaces where dependencies among "events" are made more explicit, suggest how security properties can be proved in a relatively easy way. In the Athena tool [80, 81], an automatic-protocol checker based on the strand-space model, the dependencies among events are exploited to avoid the state explosion problem and make Athena one of the most efficient security protocol checkers developed so far. Although proofs of security based on the inductive approach of Paulson seem to require more work than strand-space based proofs, most of the steps in a proof can be carried out automatically using a theorem prover like for example Isabelle [59].

A unifying event-based model

Although successfully applied to analysing numerous security protocols, existing event-based models have been developed in a rather ad-hoc way. The strand-space model, for example, has been invented specifically for the analysis of security protocols. Paulson's approach instead adapts a more general purpose technique to represent security protocols as a set of inductive rules. Although both methods are event-based it has not been clear how they relate to each other and if there are differences that would make one more suited to the analysis of certain protocols than the other. We seek a unifying model in which we can both understand and relate various event-based approaches to security-protocol analysis.

Language for security protocols

Both strand spaces and the inductive method make an informal spring from a protocol to its representation, as either a strand space or a set of inductive rules. One way to bridge this gap is by giving formal semantics to a language for describing security protocols. To be a useful model, the semantics of the language needs to support reasoning about security properties of the protocols programmed in the language. Neither the strand-space model, nor the inductive model are well suited to give formal semantics to a security protocol language directly, nor does the multiset rewriting method of Mitchell et. al [16] – all these methods lack in compositionality.

Compositionality of the model

The event-based approaches developed so far do not address how to build up the representation of a protocol in a compositional fashion. The description of how to represent a protocol from the representation of smaller protocol components could help and improve reasoning about protocols and suggest better ways to automatise it. Compositionality is also essential in giving formal semantics to a security protocol language.

Adequacy of existing approaches

Event-based models appear to have several limitations and therefore their successful application to a broad range of protocols seems rather surprising. The strand-space model, for example, permits only a restricted form of nondeterminism arising only through the choice as to where input comes from. We try to give a formal justification to why some of the ad-hoc methods suffice for analysing several security protocols.

Relation to traditional models for concurrency

There are similarities between the event-based models and more traditional models for concurrent computation. The strands of a security protocol, for example, look like processes and the configurations (bundles) of a strand space like configurations of an event structure. Studying the relations between ad-hoc models and more traditional, well studied models like transition systems, Petri-nets or event structures may help to develop new methods for analysing security protocols and improve the existing approaches.

1.4 Contributions

Petri-nets as unifying model

We show that Petri nets provide a common framework in which to understand both the strand-space and inductive methods, and it seems, although we understand it less well, the multiset rewriting method of Mitchel et al. We establish precise relationships between the net semantics and strand spaces and inductive rules and learn that both methods relate substantially to the same kind of nets. These nets give semantics to the kind of processes that security protocols usually describe. 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 proved 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.

A study of nets with persistent conditions

Our Petri-net model for security protocols makes use of a particular kind of nets in which some conditions are persistent. These nets can be seen as a special kind of contextual nets. We show how nets with persistent conditions and therefore a particular kind of contextual nets unfold, under reasonable assumptions, to more a basic kind of nets. This unfolding appears to be new.

A formal justification to adequacy

The precise relation to our Petri-net model formally backs up the adequacy of strand-space and inductive-rule representations for broad classes of security protocols and properties, showing when nothing is lost in moving to these more restrictive models. For example, a reason for the adequacy of strand spaces lies in the fact that they can sidestep "conflict" if there are enough replicated strands available, which is the case for numerous security protocols.

Correspondence to traditional models for concurrency

We establish precise relationships between the net semantics and more traditional models for concurrency: transition semantics, trace languages, and event structures. Event structures are also related directly to strand spaces.

A security-protocol language with compositional semantics

By moving to a broader class of models we can show how event-based models can be structured in a compositional way and so used to give a formal semantics to security protocols which supports proofs of their correctness. To make the case, and provide semantics to a whole range of protocols once and for all, we study the semantics of SPL (Security Protocol Language). A protocol is a process term of SPL composed out of process terms for the various protocol roles, including process terms for an intruder. The model of a protocol is determined by the semantics of the language and so tight to its syntactic description. First we give a traditional transition-system semantics to SPL which describes the operational behaviour of security protocols in a more intuitive way. We then study an event-based semantics of SPL based on Petri nets which we can formally related to the more intuitive transition semantics. We demonstrate the usefulness of the net semantics in deriving (in contrast to postulating) proof principles for security protocols and apply them to prove security properties of a number of protocols.

Composable strand spaces

In relating strand spaces to the net semantics of SPL we found it useful to extend strand spaces so that we can compose them and observe the conditions under which the extended model reduces to original strand spaces. We also address another issue of compositionality. Because we are only interested in safety properties we can make do with languages of strand-space bundles as models of process behaviour. We show how to compose such languages so that they may be used directly in giving the semantics of security protocols. As a consequence we characterise an interesting congruence relation which lays the ground for equational reasoning between strand spaces.

Security protocol implementations

The transition semantics of SPL is intuitive and apparently easy. It should therefore be not too difficult to implement and construct a compiler for the language. Protocols written in SPL could then be both executed and formally studied. This approach reduces the gap that normally exists between an abstract model of the protocol and its actual implementation. Ideally security properties of the protocol model transfer to the running code, provided the compiler for the language has been implemented correctly.

To check the viability of an implementation of SPL we constructed a prototype compiler and used a tuple-space to build an implementation of the language that works in a distributed setting. We do not explain the details of our implementation in this thesis and refer to [13, 21] instead. We tested our present implementation, called χ-Spaces, with the protocols that we study in this thesis. Since SPL is particularly simple, programs for protocols are short and concise. SPL and therefore χ-Spaces, however, need some extensions to be able to treat industrial-strength protocols. We also started to use χ-Spaces as a testing tool by annotating the protocol roles so that an execution together with a spy stops when a security breach is found. This is done in a rather ad-hoc way and further study is needed. The first results, however, are encouraging.

The main results that we describe in this thesis have been published [13, 21, 22, 23, 24].

1.5 Outline

Chapter 2 summarises existing approaches to security protocol analysis. The strand-space method and the inductive method of Paulson, that we find are more related to this thesis, are described in more detail.

Chapter 3 introduces those concepts from the theory of Petri-nets that are used in this thesis. General nets and their simpler form, basic nets, are described. We define nets with persistent conditions and discuss how they unfold to basic nets.

Chapter 4 introduces SPL, a language for security protocols, its syntax, transition semantics and net semantics. We use the NSL protocol to show how it can be programmed as an SPL process and list the events of the net associated with the process. In this chapter we also discuss how a very general spy can be programmed in the language. The spy process can be added to the process of a protocol to study the behaviour of the protocol in a hostile environment. The chapter concludes with a formal proof of correspondence between the net semantics of SPL and its transition semantics.

Chapter 5 discusses a way of using the net semantics to prove security properties of a protocol. We explain why the net-semantics is more useful than the transition semantics in doing so. General proof principles are derived from the net semantics of SPL and a general result describing when spy events need not be considered in proofs of security. We demonstrate the use of the net semantics for analysing security protocols by showing secrecy and authentication properties for the NSL protocol.

Chapter 6 studies two key-transport protocols based on symmetric-key cryptography. We show how to program the protocols in SPL and how to prove them correct. The first protocol is the ISO 5-pass authentication protocol which makes use of a trusted authority that distributes a session key. The second protocol is a key-translation protocol, the Woo and Lam Π_3 protocol. In this protocol a trusted authority translates the cyphertext containing the key chosen by one agent to one that can be deciphered by the other party. For both protocols authentication and secrecy properties are studied as well as the robustness of the protocols with respect to session-key compromise.

Chapter 7 explains some limitations of strand spaces and shows how to extend strand spaces in order to compose them. Several operations are defined. These operations form a language for strand spaces. We then describe how to compose bundle languages so that they may be used directly in giving the semantics of security protocols. This allows us to characterise a congruence relation between terms of the strand-space language based on the underlying bundle languages. At the end of this chapter we show that, under conditions which are reasonable in security protocols, the original strand spaces are equivalent to their extended version and therefore are not unduly restrictive when they are used as a representation of a wide range of security protocols.

Chapter 8 shows that a strand space is closely related to event structures, a traditional model for concurrency. We then show how both strand spaces and the inductive model relate to SPL and how SPL relates to other models for concurrent computations.

Chapter 9 concludes and presents some ideas for future work.

Chapter 2

Formal methods for security protocols

Security protocols, despite their apparent simplicity, describe complex interactions between a number of distributed agents. Informal reasoning about their security properties is therefore error prone. Formal methods have been successfully applied to security protocol verification. To make formal reasoning simpler, the underlying cryptography is often assumed to be unbreakable. Nevertheless security weaknesses for a number of protocols can be pointed out and the conditions under which security properties hold can be made precise.

In this chapter we recall a number of formal approaches to security protocol verification without, however, trying to be exhaustive. We do not explain each method in detail but point to existing literature instead. Both the strand space-method and the inductive method of Paulson are more related to our work and therefore are explained in more detail.

2.1 The inductive approach

We briefly summarise the inductive modelling approach of Paulson [11, 61, 62, 63] to security protocols making use of the NSL-protocol example.

The protocol is modelled by a set of traces, NSL. Traces are sequences of actions. The action *Says* A B M, means that agent A sent the message M to B. The action *Gets* A M, instead, means that A received the message M. The protocol traces are built up inductively by a set of rules as shown in Figure 2.1. The trace $\alpha\hat{}t$ is the trace t extended with the action α. The inductive definition of Figure 2.1 differs slightly from the one presented in Paulson [61] – it considers input actions "*Gets*" explicitly as done in [62], rather than considering only output actions "*Says*". The given rules are to be thought of as schemata and the parameters A, B as ranging over a set of *agents* participating in the protocol. Similarly n, m range over a set of *nonces*, M over all possible *messages* that can be constructed from a set of values, by encryption and composition of messages into message-tuples. 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 [62] for a formal definition). Each message of the

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

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

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

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

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

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

Figure 2.1: NSL-protocol rules

informal protocol description corresponds to a rule:

(1) This rule extends a trace with the first output message that appears in the protocol description, provided that A's nonce is new and therefore does not already appear in the trace to be extended $n \notin parts(set(t))$.

(2) Extending a trace with the second message of the protocol requires both that B's nonce is new and A's message was previously received by B.

(3) To extend a trace with the last message that A sends to B, in addition to similar preconditions to those of rule (2), the preconditions contain an equality check on nonces. The nonce sent by A in the first message has to correspond to the the nonce returned to A. This check is introduced by adding A's first message as a rule precondition.

$(receive)$ An agent can get a message only if it has previously been sent to him. The rule does not require any other precondition, thus the reception of a message is always possible.

$(fake)$ The last rule models the spy. The set $spy(t)$ contains all the messages the intruder can build up from past traffic. A typical spy can eavesdrop on all messages (all messages in t are included in $spy(t)$), it can build up new messages or extract parts of messages. Such an active spy may also be able to encrypt and decrypt with all available keys, typically all public keys and the private keys of corrupted agents. A precise definition of the set $spy(t)$ can be found in [61, 62].

NSL traces are closed under "stuttering" – once the premises of a rule hold, the rule can be applied any number of times. In the following trace for example the rule (1) has been applied twice:

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

One could think of a tighter model with a distinct receive rule for each different message of the protocol and with premises that enforce events to occur in a sequential fashion. To prove properties about protocols, it seems that the more "generous" model described in Figure 2.1 is sufficient. One reason is that we want protocols to be secure in a hostile environment where a spy can produce stuttering.

Security properties such as secrecy and authentication can be defined on the set of traces of the protocol. The protocol rules are used to prove properties inductively. Since proofs of security properties based on rule induction can be long, machine support, using for example the Isabelle [59] theorem prover, is particularly valuable.

2.2 The strand-space approach

The strand-space model [35, 88, 89, 90, 91] for the analysis of security protocols is based on sequences of actions called strands. In contrast to the inductive model a strand is a sequence of actions of an individual agent and not of a global protocol interaction.

A strand space consists of $\langle s_i \rangle_{i \in I}$, an indexed set of strands. An individual strand s_i, where $i \in I$, is a finite sequence of output or input actions of the kind $\mathit{out}\, M$ or $\mathit{in}\, M$ respectively with M a message built up by encryption and pairing from a set of values. A value whose first appearance in a strand is on an output message, is said to be *originating* on that strand. A value is said to be *uniquely originating* on a strand space if it is originating on only one of its strands. The concept of unique origination captures the idea that a nonce or a value is chosen uniformly at random from a large set and therefore practically unguessable.

The actions of an agent A that initiates an NSL protocol round with B as responder and where n, m are the nonces involved can be described by the following strand:

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

The actions of the corresponding responder B instead can be described by the strand:

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

The NSL protocol as a system composed of initiator and responder runs can be modelled by an indexed set of strands of the kind described above. Since nonces are supposed to be unguessable one needs to take care in constructing the strand space for NSL to keep nonces uniquely originating on the strand space. A spy can be modelled by adding a number of strands that describe the intruder capabilities (see [88, 90, 91] for more details).

A strand space has an associated graph whose events (nodes) identify an action of a strand by strand index and position of the action in that strand. Events associated to output actions are called output events and those associated to input actions input events. Edges are between output and input events concerning the same message and between consecutive events on a same strand. Bundles model protocol runs. A bundle selects from the events of a strand space those that occur in a run of the protocol and displays their causal dependencies. A bundle is a finite and acyclic subgraph of the strand-space graph. Each node in the bundle requires all events that precede it on the same strand (together with the edges that denote the strand precedence). Moreover each input node in the bundle has exactly one incoming edge from an output node. For example, a bundle for the NSL protocol describing a partial run of the protocol between initiator A and responder B:

$$\begin{tikzcd}
\mathit{out}\ \{n, A\}_{Pub(B)} \arrow[r] & \mathit{in}\ \{n, A\}_{Pub(B)} \arrow[d, Rightarrow] \\
 & \mathit{out}\ \{n, m, B\}_{Pub(A)}
\end{tikzcd}$$

(for simplicity we draw actions associated with nodes instead of the nodes themselves).

Security properties are defined on the set of bundles of a strand space that describes a protocol. In proving security properties the dependencies among the events in a bundle can be exploited to reconstruct from a known event the earliest event in the bundle that violates the property. Proofs often turn out to be shorter than those carried out using the inductive approach and therefore they can be done by hand.

Various techniques have been studied to make proofs in the strand-space model even shorter and easier. The work on ideals [88] shows that, regardless of the protocol under consideration, the "standard" attacker has limited abilities. These results can be used to cut down the number of cases that arise when establishing which events belong to a certain bundle to those events concerning trusted protocol participants. Authentication tests [84, 86] can be applied to a number of protocols to establish authentication properties analysing the behaviour of each trusted principal independently. When an authentication test fails, however, the authentication property might still hold for the protocol under consideration. The strand-space approach to security protocols has been automatised to some extent. Athena [80, 81] is a security-protocol checker based on an extension of the strand-space model. In Athena event dependencies help avoiding state space explosion – a system with concurrent events is not modelled with all possible event-interleavings but only with those combinations that respect the event dependencies. Athena has also been used for the automatic generation of protocols that satisfy certain properties and to implement them in Java [82].

Another application of strand spaces studies the conditions under which two protocols can be considered independent [85] – the achievement of a security property by one protocol does not depend on the achievement of a security property of the other protocol. Independent protocols can therefore be composed without thwarting their individual security properties. Strand spaces also aid the design of new protocols [36].

2.3 Other approaches

We recall a number of other approaches to the description and analysis of security protocols.

Multiset rewriting and CAPSL. The multiset rewriting approach to security protocols [16] is based on linear logic [32] and, similarly to the strand-space approach and the inductive approach, it follows the assumptions of Dolev and Yao. A protocol and a spy are described by a number of rewriting rules. To each action of a principal corresponds a rule. Pre- and postconditions of a rule are first order logical formulae that describe both the local state of principals involved in the protocol and the contents of the network through which principals communicate. Existential quantification of formulae and the use of existential quantification similar to that of natural deduction, models values that are freshly chosen during a protocol execution (for example nonces). In the multiset rewriting approach certain logical formulae play a role similar to strands and stand for system components [17]. This insight underlies the connection between linear logic and strand spaces described in [15].

The high level language CAPSL [25] is close to informal protocol descriptions. Its primary goal is to serve as a common specification language that can be translated to input for different verification tools. CAPSL is based on the multiset rewriting model [16]; a CAPSL program is translated to an intermediate language (CIL) from which code for different verification tools is generated (through so called "connectors"). A number of security properties can then be checked automatically. Not all connectors have been formally proved correct and therefore it is not always clear whether or not the verified properties are properties that hold on the multiset rewriting model of the protocol. The question of how to compose different CAPSL descriptions has not been addressed yet either.

Recent work showed how JAVA code can be generated from CIL code and so implement protocols [53]. The Java code from CIL allows communication via sockets to an environment (or network). A demonstration environment that works locally has been programmed. We do not know if the system has already been tested in a distributed setting.

Logic approaches. One of the first approaches to formal reasoning about security protocols is a logic approach. The BAN logic [12] is a belief logic which represents the believes of agents at various stages in a protocol execution as logical formulae. The initial beliefs of the protocol participants are the logical formulae that describe the various protocol steps. With the rules of the logic one derives formulae that express security properties, typically authentication properties for the protocol. The BAN logic does not, unfortunately, always provide an accurate analysis of protocols; several protocols that appear to be secure in the BAN model reveal security breaches when studied in a different model. For example, the security weakness of the Needham Schröder protocol pointed out by Lowe [47] was not revealed by the BAN logic. The logic seems to be more adequate for reasoning about freshness of nonces and keys and to some extent for reasoning about authentication, than for other properties such as confidentiality. Confidentiality in the BAN model is studied in a rather informal way on top of the formulae that can be derived from the initial protocol axioms. There have been several attempts to improve the BAN logic (see for example [49]) resulting, however, in more complicated and less intuitive logics. The degree of adequacy of the BAN logic for the analysis of security protocols has been hidden by its initial lack of a clear operational semantics. Some of the subtleties of the logic have been clarified by Abadi and Tuttle [7], we believe however that the logic still lacks an adequate operationally based semantics. Another attempt to give semantics to the BAN logic is that of Syverson [83] which is based on strand spaces.

Process calculi, types, and model checking. The use of process calculi to model security protocols allows a more operational view of a security protocol which sometimes is more intuitive. The various protocol roles are modelled as processes that exchange messages usually through channels. The formal semantics of the language is then used as basis for the analysis.

Communicating Sequential Processes (CSP) [39], an abstract language for the description of concurrent systems, whose components interact via message passing, has proved itself useful in modelling and analysing security protocols [39, 71, 72, 73, 74, 76, 75, 78]. In the CSP approach, not only the agents that participate in the protocol are described as CSP processes but also the network and the spy. The analysis is carried out on the trace model, the sequences of actions of processes of a CSP system that represents the protocol. Security properties in fact, are often expressible as safety properties and therefore they reduce to properties holding on finite traces. Proving security properties by hand is rather lengthy and tedious in the CSP model. If one restricts the attention to a finite subsystem then model checking techniques can automatise the analysis. The FDR model checker, for example, has been applied in a successful to verify “CSP security protocols” [47, 68]. Model checking techniques are applicable to finite state system but security protocols typically have an infinite number of states – they may involve an arbitrary number of rounds among the participants and generate an arbitrary number of new nonces and keys. Under certain assumptions the results obtained analysing a finite model, can be extended to the whole infinite process describing a protocol. Data independence techniques [70] have been used for that purpose.

Another process algebra that has been used to model and analyse security protocols is the π-calculus [54] and in particular its extension the Spi calculus [5]. The Spi calculus extends the π-calculus with cryptographic primitives. The scoping rules of the π-calculus together with its new name generation mechanism are particularly useful to model the generation of new and unguessable values such as nonces. Security properties can be established by showing that certain equivalences between Spi-calculus processes hold. In this context, there is no need to define a hostile environment explicitly, the spy is an arbitrary Spi calculus process instead. Experience with the model has shown that proofs of security properties that are defined by process equivalences require a lot of work. The work of Abadi [1] has shown that type-checking techniques can help identifying in a much easier way which protocols satisfies certain secrecy properties. A protocol that does not type-check, however, is not said to be flawed. Gordon and Jeffry [34] developed another type system to check for correspondence assertions, a certain kind of authentication properties (see Woo and Lam [98]).

Other process calculi have been used to model security protocols and study some of their security properties – the ambient calculus [14] and the Join calculus [4] for example. A number of other model checkers and state exploration methods have been applied to the verification of security protocols as well. To name a few of them, Murphi [55], SVM [20], SPIN [10] and NRL [51]. The NRL protocol analyser combines theorem proving and model checking – after automatically proving invariants to reduce the state space it does an exhaustive search of the state space.

Petri-net approach. Petri nets have been applied to the verification of cryptographic protocols [57]. We found that the level of detail of the approach in [57] makes the model rather complex and difficult to analyse. We do not see any significant relationship between our approach and that of [57].

Computational-theoretic approaches. Advances have been made in formal reasoning about security protocols where some of the assumptions about the underlying cryptography are lifted. Mitchel et al. proposes an approach based on the π-calculus to study the interactions between a protocol and its cryptographic primitives. Cryptography is not treated as a black box anymore and the spy is an arbitrary polynomial-time process [44, 45, 46]. In [87] Guttman et al. quantify a bound on the divergence between a protocol analysis carried out with the usual Dolev and Yao assumptions on a strand-space model and one where the characteristics of concrete cryptographic primitives are taken into account.

Chapter 3

Petri nets

Petri nets have been first introduced by C. A. Petri in the 60's, and today are a well known model for distributed and concurrent computations [8, 42, 66, 95, 97]. They are a so called "non-interleaving" model, where an event occurrence affects only a neighbourhood of events within a global state. Events that do not affect part of the same neighbourhood are said to be independent and could potentially occur simultaneously.

In this thesis we use Petri nets to give semantics to a language specifically designed for modelling security protocols. In this chapter we introduce the concepts from Petri nets that we use later in the thesis. We first give a rather general definition of Petri nets which we then specialise to a more basic kind of nets. Of particular interest are nets with *persistent conditions*. We choose nets with persistent conditions for modelling and analysing security protocols. It turns out that, under reasonable conditions, nets with persistent conditions are not more expressive than the simpler basic nets; we show how to unfold persistent conditions and yield a basic net out of a net with persistence. Even if nets with persistent conditions arose independently in previous studies, for example in the form of contextual nets or nets with test arcs, we are not aware of a previous result that shows how to unfold them to basic nets.

3.1 General Petri nets

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

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

- a set of *conditions* (or *places*), P,
- a set of *events* (or *transitions*), T,
- a *precondition map pre*, which to each $t \in T$ assigns a multiset $pre(t)$ over P. It is traditional to write ${}^{\cdot}t$ for $pre(t)$.

- a *postcondition* map *post* which to each $t \in T$ assigns an ∞-multiset $post(t)$ over P, traditionally written $t^{\cdot}$.
- a *capacity* function Cap which is an ∞-multiset over P, assigning a nonnegative number or ∞ to each condition p, bounding the multiplicity to which the condition can hold; a capacity of ∞ means the capacity is unbounded.

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

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

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

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

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

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

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

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

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

3.2 Basic nets

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

A *basic Petri net* consists of

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

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

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

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

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

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

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

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

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

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

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

Basic nets are important because they are related to many other models of concurrent computation, in particular, Mazurkiewicz trace languages (languages subject to trace equivalence determined by the independence of actions) and event structures (sets of events with extra relations of causality and conflict) – see [97].

3.3 Coloured nets

We briefly introduce coloured nets following Winskel [95]. Coloured nets have been first introduced by Jensen [42] and are a useful abbreviation of Petri-nets. We consider the case of coloured nets where all multisets have multiplicity either 0 or 1, and therefore can be regarded as sets (see for example [95] for a more general definition). This simpler case will be enough for the purposes of this chapter.

Our description of coloured nets, like many others, uses the notion of places and transitions. These have a higher level nature and stand for sets of conditions and events. Colours are associated to places and transitions so that one can think of a condition as of a place in a certain colour and of an event as of a transition in a certain colour.

A *coloured net* consists of

- a set of *places*, P,
- a set of *transitions*, T,
- a colour function Δ which associates each place p with a set of colours $\Delta(p)$ and each transition t with a set of colours $\Delta(t)$,
- two maps: $\mathit{pre}, \mathit{post} : E \to \mathcal{P}ow(B)$, where

  $$\begin{aligned} E &= \{(t,c) \mid t \in T \text{ and } c \in \Delta(t)\} \\ B &= \{(p,c) \mid p \in P \text{ and } c \in \Delta(p)\} \quad . \end{aligned}$$

  If $e \in E$ then we abbreviate ${}^{\cdot}e = \mathit{pre}(e)$ and $e^{\cdot} = \mathit{post}(e)$. We require ${}^{\cdot}e \cup e^{\cdot} \neq \emptyset$ for all $e \in E$.

Markings for coloured nets consist of sets of conditions in B, which in this case are places possibly instantiated to a particular colour. The token game is the same as for basic nets:

Token game for coloured nets: For $\mathcal{M}, \mathcal{M}' \subseteq B$ and $e \in E$, define

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

As we mentioned, coloured nets are an abbreviation for ordinary Petri-nets. As already shown in [95], the following proposition holds:

Proposition 3.3.1 *A coloured net* $(P, T, \Delta, \mathit{pre}, \mathit{post})$ *determines a basic Petri-net* $(E, B, \mathit{pre}, \mathit{post})$ *where*

$$\begin{aligned} E &= \{(t,c) \mid t \in T \textit{ and } c \in \Delta(t)\} \\ B &= \{(p,c) \mid p \in P \textit{ and } c \in \Delta(p)\} \ . \end{aligned}$$

□

3.4 Nets with persistent conditions

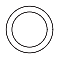
Sometimes we have use for conditions which once established continue to hold and can be used repeatedly. This is true of assertions in traditional logic, for example, where once an assertion is established to be true it can be used again and again in the proof of further assertions. Similarly, if we are to use net events to represent rules of the kind we find in inductive definitions, we need conditions that persist. In this thesis, nets with persistent conditions are used for giving semantics to a language for security protocols. The model of the language abstracts from a network considering it as a pool of persistent messages which can be read simultaneously by more than one agent. It can, for example, be convenient to use persistent conditions in modelling databases or shared memory systems where more than one user can read the same data.

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

To be more precise, we modify the definition of basic net given above by allowing certain conditions to be *persistent*. A net with persistent conditions will still consist of events and conditions related by pre- and postcondition maps which to an event will assign a set of preconditions and a set of postconditions. But, now among the conditions are the persistent conditions forming a subset P. A marking of a net with persistent conditions will be simply a subset of conditions, of which some may be persistent. Nets with persistent conditions have arisen independently several times and have been studied for example in *contextual nets* [56] and as an extension of coloured nets with *test arcs* [18, 43].

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

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



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

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

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

The token game of a net with persistent conditions could be generalised to transitions

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

where A is set or even an ∞-multiset of events. If one permits A to be a multiset then one might allow an event to occur in the net simultaneously more than once. In basic nets this would never be the case because all conditions are marked with multiplicity 1. In nets with persistent conditions however, one might want to permit simultaneous occurrences of those events whose pre- and postconditions are all persistent.

3.5 Nets with persistent conditions and basic nets

In this section we show how to unfold a net with persistent conditions to a basic net. We first construct a coloured net from the net with persistent conditions and then, through a well known unfolding, a basic net. A run of the net with persistent conditions relates to a run of the unfolded, basic net provided that any event that marks a persistent condition does not occur more than once in the run.

It is known how a coloured net with test arcs can be transformed to an equivalent coloured net without test arcs (see [18]). That construction, however, introduces infinite multiplicities in the unfolded net. Test arcs have a similar function to persistent conditions. If an event has a precondition connected with a test arc then it can occur only if the condition is marked; when the event fires, however, that condition is not consumed. If one tries to unfold nets with persistent condition as suggested in [18] then conditions with infinite capacity replace those with persistence. A general Petri-net is the result of the unfolding. In this section we show a much stronger result; it tells how persistent conditions can be unfolded to yield a basic net where all multiplicities are either 0 or 1.

In this section we make use of the following notation: Write

$$^{\cdot}b = \{e \mid b \in e^{\cdot}\} \cup \{*\}$$

for all the events that mark the condition b. We add $*$ to indicate that b can be included in an initial marking and therefore need not be marked by an event. Write

$$b^{\cdot} = \{e \mid b \in {}^{\cdot}e\}$$

for the set of events that have b as one of their preconditions.

3.5.1 Unfolding persistent conditions

Consider a net with persistent conditions

$$(B, P, E, pre, post)$$

it unfolds to the coloured net

$$(B, E, \Delta, pre, post)$$

where Δ is a colouring function for events and conditions. The colour of an event that has persistent preconditions is a tuple consisting of all its persistent preconditions and of a choice of events that marks them. We colour an event that does not have any persistent preconditions with the default colour δ. More precisely, for every event $e \in E$

$$\Delta(e) = \begin{cases} \{\delta\} & \text{if } {}^{\cdot}e \cap P = \emptyset \\ \prod_{b \in {}^{\cdot}e \cap P}(\{b\} \times {}^{\cdot}b) & \text{otherwise} \end{cases} .$$

The colour of a persistent condition consists of a set of pairs of events that can mark the condition with the events that require the condition marked. The colour of ordinary conditions instead is the default colour δ. More precisely, for every condition $b \in B$

$$\Delta(b) = \begin{cases} \{\delta\} & \text{if } b \in B \setminus P \\ {}^{\cdot}b \times b^{\cdot} & \text{otherwise} \end{cases} .$$

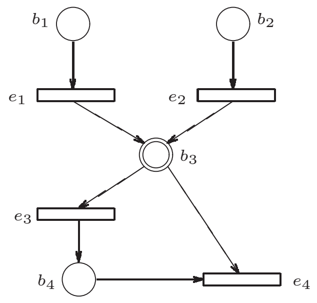
The precondition map of the coloured net is defined for pairs (e, c) of events e and colours $c \in \Delta(e)$ as follows:

$$pre(e, c) = \{(b, \delta) \mid b \in {}^{\cdot}e \setminus P\} \;\cup\; \{(b, (e', e)) \mid (b, e') \in c\} \ .$$

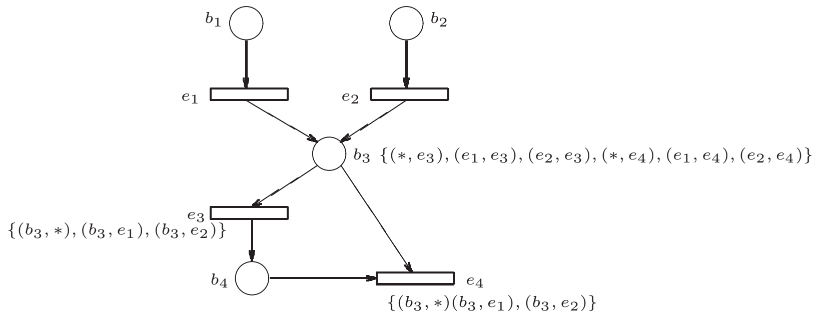
The postcondition map for pairs (e, c) where e is an event and $c \in \Delta(e)$ is defined as follows:

$$\begin{aligned} post(e, c) = {} & \{(b, \delta) \mid b \in e^{\cdot} \setminus P\} \cup \\ & \{(b, (e, e')) \mid b \in e^{\cdot} \cap P \text{ and } e' \in b^{\cdot}\} \cup \\ & \{(b, (e', e)) \mid (b, e') \in c\} \ . \end{aligned}$$

As an example of the construction consider the following net with persistent conditions:



It unfolds to the following coloured net, where non-default colours are listed in curly brackets next to events and conditions:


b_1
b_2
e_1
e_2
b_3 $\{(*, e_3), (e_1, e_3), (e_2, e_3), (*, e_4), (e_1, e_4), (e_2, e_4)\}$
e_3
$\{(b_3, *), (b_3, e_1), (b_3, e_2)\}$
b_4
e_4
$\{(b_3, *)(b_3, e_1), (b_3, e_2)\}$

As shown in Winskel's [95] a coloured net determines a Petri-net. In this case the coloured net obtained from a net with persistent conditions yields the basic net

$$(B', E', pre, post)$$

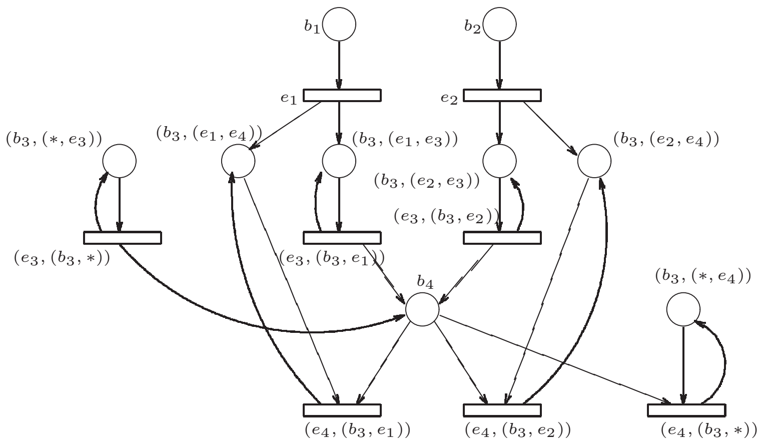
with conditions

$$B' = \bigcup_{b \in B} \{b\} \times \Delta(b)$$

and with events

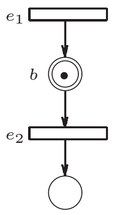
$$E' = \bigcup_{e \in E} \{e\} \times \Delta(e) \quad .$$

If one applies this unfolding to our example one obtains the basic net:

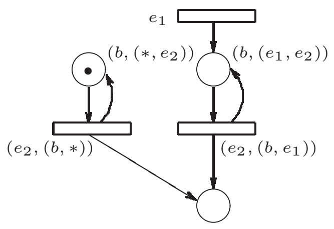


The construction that unfolds persistent conditions into ordinary conditions keeps track of how the conditions can be marked and of what events can consume a condition.

Persistent conditions in a net can be part of the initial marking and therefore do not necessarily need an event to mark them. Conditions of the kind $(b,(*,e))$ are introduced so that the behaviour of a net that has the persistent condition b in its initial marking is still related to the unfolded net where the initial marking contains $(b,(*,e))$. For example consider the net



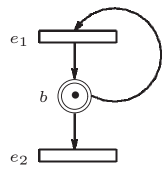
and its unfolding:



If we marked initially condition $(b,(e_1,e_2))$ instead of $(b,(*,e_2))$ the event e_1 would not be able to occur unless $(e_2,(b,e_1))$ occurs first. In the net with persistent conditions however e_1 can always occur.

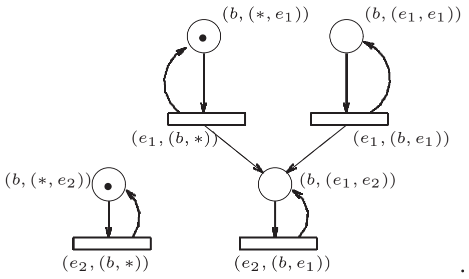
3.5.2 Relating nets with persistent conditions to basic nets

The construction that we described in the previous section, which shows how to unfold a net with persistent conditions into a basic net, is correct when the runs of the net it yields are substantially the same to the runs of the original net. Not all nets with persistent conditions unfold well if one applies the described construction. Nets unfold well if they do not have runs where events with persistent postconditions appear more than once in the same run. The following example illustrates the reasons for this restriction. Consider the net:



.

The event e_1 can occur more than once in a run of the net, without the event e_2 necessarily having to occur. In the unfolding of the net, however, after the event $(e_1, (b, *))$ occurs once, it can occur again only after the event $(e_2, (b, e_1))$ has occurred.



.

Let $\mathcal{M} \subseteq B$ and $\bar{\mathcal{M}} \subseteq B'$ be two markings. Define $\mathcal{M} \sim \bar{\mathcal{M}}$ if all the following conditions hold:

1. $\mathcal{M} \setminus P = \{b \mid (b, \delta) \in \bar{\mathcal{M}}\}$,
2. if $b \in \mathcal{M} \cap P$ then $\exists e \in {}^{\cdot}b \,.\, \forall e' \in b^{\cdot} \,.\, (b, (e, e')) \in \bar{\mathcal{M}}$,
3. if $(b, c) \in \bar{\mathcal{M}}$ and $c \neq \delta$ then $b \in \mathcal{M} \cap P$.

Write $\mathcal{M} \prec \bar{\mathcal{M}}$ whenever the first and the third condition above hold but not necessarily the second condition.

Theorem 3.5.1 *Let N be a net with persistent conditions and N' be the basic net obtained from it by the described construction.*

1. *To every finite run*
$$\mathcal{M}_0 \xrightarrow{e_1} \cdots \xrightarrow{e_{i-1}} \mathcal{M}_i \xrightarrow{e_i} \cdots$$
in N in which events that carry persistent postconditions occur at most once, there corresponds a run
$$\bar{\mathcal{M}}_0 \xrightarrow{(e_1,c_1)} \cdots \xrightarrow{(e_{i-1},c_{i-1})} \bar{\mathcal{M}}_i \xrightarrow{(e_i,c_i)} \cdots$$
in N' such that at every stage i in the run $c_i \in \Delta(e_i)$ and $\mathcal{M}_i \sim \bar{\mathcal{M}}_i$.

2. *To every finite run*
$$\bar{\mathcal{M}}_0 \xrightarrow{(e_1,c_1)} \cdots \xrightarrow{(e_{i-1},c_{i-1})} \bar{\mathcal{M}}_i \xrightarrow{(e_i,c_1)} \cdots$$

in N' there corresponds a run

$$\mathcal{M}_0 \xrightarrow{e_1} \cdots \xrightarrow{e_{i-1}} \mathcal{M}_i \xrightarrow{e_i} \cdots$$

in N such that at every stage i in the run $\mathcal{M}_i \prec \bar{\mathcal{M}}_i$.

Proof.

1. By induction on the length of a run in N. Let $\mathcal{M}_0$ be a marking of conditions in N and define

$$\bar{\mathcal{M}}_0 = \{(b,\delta) \mid b \in \mathcal{M}_0 \setminus P\} \cup \bigcup_{b \in \mathcal{M}_0 \cap P} \{b\} \times (\{*\} \times b^{\cdot}) \quad .$$

Clearly $\bar{\mathcal{M}}_0$ is a marking of conditions in N' and $\mathcal{M}_0 \sim \bar{\mathcal{M}}_0$. Suppose that to a run in N

$$\mathcal{M}_0 \xrightarrow{e_1} \cdots \xrightarrow{e_{i-1}} \mathcal{M}_i$$

corresponds in N' the run

$$\bar{\mathcal{M}}_0 \xrightarrow{(e_1,c_1)} \cdots \xrightarrow{(e_{i-1},c_{i-1})} \bar{\mathcal{M}}_i$$

such that $\mathcal{M}_j \sim \bar{\mathcal{M}}_j$ for all $0 \le j \le i$. If

$$\mathcal{M}_i \xrightarrow{e_{i+1}} \mathcal{M}_{i+1}$$

then ${}^{\cdot}e_{i+1} \subseteq \mathcal{M}_i$ and therefore, since by the induction hypothesis $\mathcal{M}_i \sim \bar{\mathcal{M}}_i$, it follows that

$${}^{\cdot}e_{i+1} \setminus P \subseteq \{b \mid (b,\delta) \in \bar{\mathcal{M}}_i\}$$

and for all $b \in {}^{\cdot}e_{i+1} \cap P$ there exists $e_j \in {}^{\cdot}b$ such that

$$(b,(e_j,e_{i+1})) \in \bar{\mathcal{M}}_i \quad .$$

This yields a colour $c_{i+1} \in \Delta(e_{i+1})$ and therefore an event (e_{i+1},c_{i+1}) in the net N' such that

$${}^{\cdot}(e_{i+1},c_{i+1}) \subseteq \bar{\mathcal{M}}_i \quad .$$

More precisely, one distinguishes two cases:

If ${}^{\cdot}e_{i+1} \cap P = \emptyset$ then

$$\Delta(e_{i+1}) = \{\delta\} \quad \text{and} \quad {}^{\cdot}(e_{i+1},\delta) = \{(b,\delta) \mid b \in {}^{\cdot}e_{i+1}\} \ .$$

Since ${}^{\cdot}e_{i+1} \subseteq \{b \mid (b,\delta) \in \bar{\mathcal{M}}_i\}$ clearly

$${}^{\cdot}(e_{i+1},\delta) \subseteq \bar{\mathcal{M}}_i \ .$$

If instead ${}^{\cdot}e_{i+1} \cap P \neq \emptyset$ then let

$$c_{i+1} \in \Pi_{b \in {}^{\cdot}e_{i+1} \cap P}(\{b\} \times A)$$

where $A \subseteq {}^{\cdot}b$ is the set of events e_j such that $(b,(e_j,e_{i+1})) \in \bar{\mathcal{M}}_i$. It is easy to check that

$${}^{\cdot}(e_{i+1},c_{i+1}) \subseteq \bar{\mathcal{M}}_i \ .$$

The event (e_{i+1}, c_{i+1}) is enabled at the marking $\bar{\mathcal{M}}_i$ if it does not cause contact, in other words if

$$(\bar{\mathcal{M}}_i \setminus {}^{\cdot}(e_{i+1}, c_{i+1})) \cap (e_{i+1}, c_{i+1})^{\cdot} = \emptyset \quad .$$

Suppose the contrary. Assume that there exists a condition b and an event e such that $(b, (e_{i+1}, e)) \in (e_{i+1}, c_{i+1})^{\cdot}$. Then $b \in e_{i+1}^{\cdot} \cap P$. Assume also that $(b, (e_{i+1}, e)) \in \bar{\mathcal{M}}_i$. The event e_{i+1} is an event of the net N, therefore $(b, (e_{i+1}, e)) \notin \bar{\mathcal{M}}_0$ (otherwise e_{i+1} would be $*$). It follows from the token game for basic nets that there is an event occurrence (e_j, c_j) with $j \leq i$ such that $(b, (e_i, e)) \in (e_j, c_j)^{\cdot}$ and therefore a preceding event occurrence in the run of N such that $b \in e_j^{\cdot} \cap P$. The assumption that in the run of N under consideration an event with persistent postconditions occurs at most once is contradicted. One can reach a contradiction in a similar way for the case where one assumes $(b, \delta) \in (e_{i+1}, c_{i+1})^{\cdot}$ and $(b, \delta) \in \bar{\mathcal{M}}_i$. Therefore

$$\bar{\mathcal{M}}_i \xrightarrow{(e_{i+1}, c_{i+1})} \bar{\mathcal{M}}_{i+1}$$

where $\bar{\mathcal{M}}_{i+1}$ is the marking obtained with the token game for basic nets. The markings

$$\begin{aligned}
\mathcal{M}_{i+1} &= (\mathcal{M}_i \setminus ({}^{\cdot}e_{i+1} \setminus P)) \cup e_{i+1}^{\cdot} \quad \text{and} \\
\bar{\mathcal{M}}_{i+1} &= (\bar{\mathcal{M}}_i \setminus \{(b, \delta) \mid b \in {}^{\cdot}e_{i+1} \setminus P\}) \\
&\quad \cup \{(b, \delta) \mid b \in e_{i+1}^{\cdot} \setminus P\} \\
&\quad \cup \{(b, (e_{i+1}, e)) \mid b \in e_{i+1}^{\cdot} \cap P \text{ and } e \in b^{\cdot}\}
\end{aligned}$$

are such that $\mathcal{M}_{i+1} \sim \bar{\mathcal{M}}_{i+1}$.

2. Consider a marking $\bar{\mathcal{M}}_0$ of conditions in N' and define

$$\mathcal{M}_0 = \{b \mid (b, c) \in \bar{\mathcal{M}}_0\} \quad .$$

Clearly $\mathcal{M}_0 \prec \bar{\mathcal{M}}_0$. Suppose that to the run in N'

$$\bar{\mathcal{M}}_0 \xrightarrow{(e_1, c_1)} \cdots \xrightarrow{(e_{i-1}, c_{i-1})} \bar{\mathcal{M}}_i$$

corresponds the run

$$\mathcal{M}_0 \xrightarrow{e_1} \cdots \xrightarrow{e_{i-1}} \mathcal{M}_i$$

in N such that $\mathcal{M}_j \prec \bar{\mathcal{M}}_j$ for all $0 \leq j \leq i$. If

$$\bar{\mathcal{M}}_i \xrightarrow{(e_{i+1}, c_{i+1})} \bar{\mathcal{M}}_{i+1}$$

then ${}^{\cdot}(e_{i+1}, c_{i+1}) \subseteq \bar{\mathcal{M}}_i$. From the induction hypothesis $\mathcal{M}_i \prec \bar{\mathcal{M}}_i$ and therefore ${}^{\cdot}e_{i+1} \subseteq \mathcal{M}_i$.

The event e_{i+1} is enabled at the marking $\mathcal{M}_i$ if

$$(\mathcal{M}_i \setminus ({}^{\cdot}e_{i+1} \cup P)) \cap e_{i+1}^{\cdot} = \emptyset \quad .$$

Suppose there is contact, instead. Since N is a net with persistent conditions, contact can happen only on non-persistent. Suppose therefore that there is a condition b such that $b \in e_{i+1}{}^{\cdot} \setminus P$ and $b \in \mathcal{M}_i \setminus {}^{\cdot}e_{i+1}$. Clearly $(b, \delta) \in (e_{i+1}, c_{i+1})^{\cdot}$ and $(b, \delta) \in \bar{\mathcal{M}}_i \setminus^{\cdot} (e_{i+1}, c_{i+1})$ since $\mathcal{M}_i \prec \bar{\mathcal{M}}_i$. This, however, is not possible since the event (e_{i+1}, c_{i+1}) is enabled.

Therefore

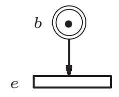
$$\mathcal{M}_i \xrightarrow{e_{i+1}} \mathcal{M}_{i+1}$$

where $\mathcal{M}_{i+1}$ is the marking obtained with the token game on nets with persistent conditions. In a similar way to point 1. one can show that $\mathcal{M}_{i+1} \prec \bar{\mathcal{M}}_{i+1}$.

□

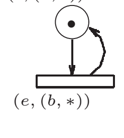
For simplicity we studied the relation between the runs of a net with persistent condition and the runs of the unfolded net only when one event at a time can fire. Petri-nets can describe true concurrent behaviour and so one might want to generalise the above result to runs in which events can occur simultaneously. Provided some care is taken, the result can be generalised straightforwardly:

- One can generalise to transitions of the kind $\mathcal{M} \xrightarrow{A} \mathcal{M}'$ where A is a set but *not* a multiset; simultaneous or concurrent occurrences of events are allowed, provided they are not occurrences of the same event. The marking of the following net, for example, permits the event e to occur simultaneously any number of times:



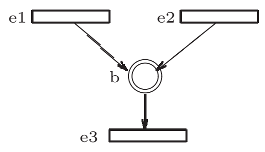
.

In its unfolding, however, the event e can occur any number of times but only once at a time:

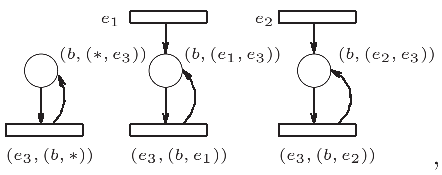


.

- If one does generalise to transitions $\mathcal{M} \xrightarrow{A} \mathcal{M}'$ where A is a set then one needs to impose another restriction: all events in the net with persistent conditions carry at least one non-persistent condition. Consider the following net with persistence

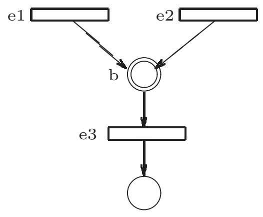


and suppose that both the events e_1 and e_2 occur in one of its runs. In the unfolded net

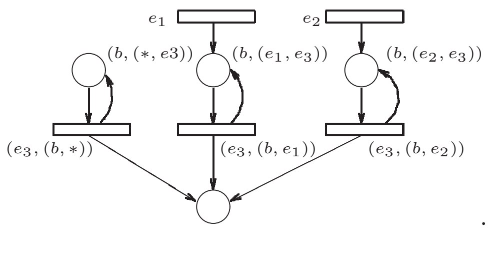


,

the events $(e_3,(b,e_1))$ and $(e_3,(b,e_2))$ can both occur simultaneously. One would therefore obtain a corresponding run in the net with persistence that contains two simultaneous occurrences of the event e_3. Transitions $\mathcal{M} \xrightarrow{A} \mathcal{M}'$ where A is a set can't, however, express simultaneous occurrences of e_3 – one would need a multiset instead. If the net with persistent conditions is such that every event has at least one non-persistent condition then contact on that condition prevents the previous situation. Consider for example



and its unfolding



.

Chapter 4

A Security Protocol Language

The few lines of an informal protocol description appear very simple at first sight. It is not uncommon to be deceived by this apparent simplicity in thinking that the protocol satisfies a number of desired security properties, when instead it can take very little for an attacker to undermine security. Only a rigorous way of reasoning about the protocol and its properties can give enough guarantees about its correctness and show the conditions under which security is not compromised.

In order to be more explicit about the activities of participants in a protocol and those of a possible attacker, and to express these compositionally, we design an economical process language for the purpose. The language **SPL** (Security Protocol Language) that we introduce resembles Linda [31] in some ways: simple primitives are provided so that distributed agents can interact via a common space to which messages are sent and from which messages are received via pattern matching.

Security protocols often make use of cryptography to achieve the desired security goals, therefore messages of the SPL language may contain cyphertext. In SPL an underlying cryptographic system is assumed, however without fixing a particular one. Even if SPL abstracts from the the details of concrete cryptographic algorithms the language provides means of representing public key and symmetric key cryptography and it could easily be extended to represent other operations such as for example hashing. The main assumption that is made about cryptography is that it is unbreakable and that random numbers are unguessable.

The space of messages through which communication takes place is, in some cases, a quite abstract model of a complex network in which communication can be point to point. However, when studying security properties that are meant to hold for all possible executions of a protocol, the simple "space model" turns out to be sufficient. This is certainly the case for security protocols that operate in hostile environments, where malicious entities have the power to redirect messages. In SPL we chose to abstract even further and consider all messages in the network (the space) to be persistent – once sent messages remain available on the network forever. If a malicious entity has access to the network so that it is able to redirect the communication and make an arbitrary number of copies of any message that appears on the network then a space with message persistence conveniently incorporates this form of hostility.

The SPL language is a process language and close to an asynchronous π-Calculus [54], similar to that adopted in [5], though in its treatment of new names its transition semantics will be closer to that in [64] (it separates concerns of freshness from concerns of scope which are combined in the π-Calculus restriction).

In order to express security proprieties of a protocol programmed in SPL in a precise way we give a formal semantics to the language. A traditional transition-semantics allows several security properties to be formalised. However, an obvious proof strategy based on establishing an invariant property of the transitions that constitute a protocol run, is badly supported by the SPL transition-semantics. A different semantics for the language, an event-based semantics in which the events of a protocol and their dependencies are made explicit, appears more appropriate. It turns out that the events of SPL and their pre- and postconditions form a Petri-net [66, 95]. The SPL net is strongly related to t he more traditional transition system of the language.

This chapter introduces both syntax and semantics of SPL. In Section 4.1 we introduce its syntax together with an informal meaning of the main language constructs and some conventions that make SPL programs more concise. As a first examples of SPL programs we show the processes for the Needham-Schröder-Lowe protocol [47] and for a powerful spy. The system obtained composing protocol and attacker models an interesting situation of a security protocol operating in a rather hostile environment and for which security questions don't find an immediate and obvious answer. A traditional transition semantics for SPL is described in Section 4.2. The same section gives reasons for the inadequacy of the transition model which lead to the net-semantics of Section 4.3. As an example of the reachable events of a process in the SPL net, the NSL events and the spy events are described graphically.

4.1 The syntax of SPL

In this section we show the syntax of SPL and give an informal explanation of the main language constructs. Some conventions are introduced to help programming and to make programs more concise. As an example of how SPL can be used to formalise security protocols we show how to program the Needham-Schröder-Lowe (NSL) protocol in SPL – the informal protocol description of Section 1.1 is taken to a process term giving precise account of the activities of the participants in the protocol. In addition we give the SPL process for a rather powerful and general attacker which describes the hostile environment in which security protocols often operate.

4.1.1 Syntactic sets

We start by giving the syntactic sets of the language:

- An infinite set of names $\mathbf{N}$, with elements $n, m, A, \cdots$. Names range over nonces as well as agent names, and can also include other values. We often use capital letters for names of agents and reserve small letters for nonces and other values.
- Variables over names $x, y, \cdots, X, Y, \cdots$.
- Variables over keys $\chi, \chi', \chi_1, \cdots$.
- Variables over messages $\psi, \psi', \psi_1, \cdots$.

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

Figure 4.1: Syntax of **SPL**

- Indices $i, j \in$ **Indexes** with which to index components of parallel compositions.

The other syntactic sets of the language are described by the grammar shown in Figure 4.1. We use "vector" notation; we assume that vectors of variables consist of a possibly empty list of distinct variables (for example, the vector $\vec{x}$ abbreviates some possibly empty list $x_1, \cdots, x_l$) wheres vectors of name expressions and messages have at least one element and may contain repetitions.

We use $Pub(v)$, $Priv(v)$ for the public, private keys of v, and we use $Key(\vec{v})$ for the symmetric key shared by agents with names in $\vec{v}$. Keys can be used in building up encrypted messages. A message can be a name or key expression, the composition of two messages M, M', an encryption $\{M\}_k$ representing the message M encrypted using the key k, or a message variable.

4.1.2 Free variables

In the process terms $out\, new(\vec{x})\, M.\, p$ and $in\, pat\, \vec{x}\vec{\chi}\vec{\psi}\, M.\, p$, the annotations $new(\vec{x})$ and $pat\vec{x}\vec{\chi}\vec{\psi}$ are binders for the variables in the lists $\vec{x}$ and $\vec{x}\vec{\chi}\vec{\psi}$ respectively. The occurrence of a variable x, χ or ψ in a process term is said to be *bound* if it occurs in the scope of a "$new(\vec{x})$" or "$pat\, \vec{x}\vec{\chi}\vec{\psi}$" with $x \in \vec{x}$ or $\psi \in \vec{\psi}$. A variable occurring in a process term is said to be *free* if it is not bound. Formally one can define the set of free variables $fv(p)$ of a process term p on the structure of the term. Let $fv(M)$, the free variables of a message M, to be the set of variables which appear in M (this set can be easily defined by induction on the structure of messages) and define:

Definition 4.1.1 (Free variables of a process term)

$$\begin{array}{lcl} fv(out\, new(\vec{x})\, M.p) & = & (fv(p) \cup fv(M)) \backslash \{\vec{x}\} \\ fv(in\, pat\vec{x}\vec{\chi}\vec{\psi}\, M.p) & = & (fv(p) \cup fv(M)) \backslash \{\vec{x}, \vec{\chi}, \vec{\psi}\} \\ fv(\|_{i \in I} p_i) & = & \bigcup_{i \in I} fv(p_i) \quad . \end{array}$$

□

As usual, we say that a process without free variables is *closed*, as is a message without variables.

We use standard notation for substitution into the free variables of an expression. For example $p[\vec{x}/\vec{n}]$ stands for the process term obtained from p substituting each free occurrence of a variable in the list $\vec{x}$ with the corresponding value in the list $\vec{n}$. We will only be concerned with the substitution of names or closed (variable-free) messages, obviating the problems of variable capture.

4.1.3 Informal meaning of processes

As shown in the grammar of Figure 4.1, SPL processes are built up either by prefixing a smaller process with an action or by composing several processes in parallel. The empty parallel composition is the smallest component and forms the basic case for building up process terms inductively. SPL actions are very simple; they allow processes to communicate through a space of messages by sending a message onto the space or receiving a message from the space whenever a pattern is matched successfully. In addition, new name generation is incorporated in the output action.

Process terms are informally explained as follows:

$out\, new(\vec{x})\, M.p$ This process chooses fresh, distinct names $\vec{n} = n_1, \cdots, n_l$ and binds them to the variables $\vec{x} = x_1, \cdots, x_l$. The message $M[\vec{n}/\vec{x}]$ is output to the network and the process resumes as $p[\vec{n}/\vec{x}]$. The communication is *asynchronous* in the sense that the action of output does not await input. The *new* construct is like that of Pitts and Stark [64] and abstracts out an important property of a value chosen randomly from some large set: such a value is likely to be new.

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

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

4.1.4 Some conventions

It simplifies the writing of process expressions making them more concise, if we adopt some conventions.

1. We simply write $out\, M.p$ when the list of "*new*" variables is empty.

2. We allow ourselves to write
$$\cdots in\ M\, .\, p \cdots$$
in an expression, to be understood as meaning the expression
$$\cdots in\, pat\, \vec{x}\vec{\chi}\vec{\psi}\ M\, .\, p \cdots$$
where the pattern variables $\vec{x}\vec{\chi}\vec{\psi}$ are precisely those variables left free in M by the surrounding expression. For example
$$in\ \psi\, .\, out\, new(x)\ \psi, x\, .\, in\ x\, .\, nil$$
For the first input, the variable ψ is free in the whole expression, so by convention is a pattern variables. On the other hand, in the second input the variable x is bound by the outer $out\, new(x) \cdots$ and so by the convention is not a pattern variable, and has to be that value sent out earlier.

3. Often we won't write the nil process explicitly, so, for example, omitting its mention at the end of process term.

4. A parallel composition can be written in infix form via the notation

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

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

4.1.5 The size of a process

Often definitions and proofs by structural induction won't suffice, and we would like to proceed by induction on the "size" of closed process expressions, i.e. on the number of prefix and parallel composition operations in the process expression. But because of infinite parallel compositions, expressions may not contain just a finite number of operations, so we define $size(p)$ of a process term p to be an ordinal as follows:

Definition 4.1.2 The *size* of a closed process term is an ordinal given by the structural induction:

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

□

The size of a process term does not change if some of its variables are substituted by names or messages.

Proposition 4.1.3 *Let $\vec{x}$, $\vec{\chi}$, and $\vec{\psi}$ be vectors of name and message variables respectively and let $\vec{n}$, $\vec{k}$, and $\vec{M}$ be vectors of names, keys, and messages respectively. For every process term p*

$$size(p) = size(p[\vec{n}/\vec{x}, \vec{k}/\vec{\chi}, \vec{M}/\vec{\psi}]) \quad .$$

Proof. By induction on the structure of process terms p and making use of the definition of *size* and substitution. □

4.1.6 The algebra of messages

Messages are built up as described in the grammar of Figure 4.1:

$$M, M' := v \mid k \mid M, M' \mid \{M\}_k \mid \psi, \psi', \ldots$$

where v stands for any value expression, k stands for any key expression, and $\psi, \psi', \ldots$ are message variables. The set of messages we are interested in is therefore given by the free algebra constructed from value and key expressions and message variables by concatenation of messages and encryption of messages with keys and such that the operation of message concatenation is associative. No other equations on messages are added. We don't make use of parentheses when writing a message term that is the concatenation of a number of shorter messages and we say that two messages M and N are *equal* and write $M = N$ if they are syntactically the same term. In particular the following property on message terms holds:

Proposition 4.1.4 (Strong encryption.) *For all messages M, N and keys k, k', if $\{M\}_k = \{N\}_{k'}$ then $M = N$ and $k = k'$.* □

This means that if two cyphertexts are the same then they were obtained using the same cleartext and the same key. In other words, a cyphertext can be decrypted only with the right key. In practise, encryption schemes may not satisfy this property, like for example the symmetric encryption scheme CBC (see [33]) that uses the DES block cypher (see [26]) or a public key encryption scheme like RSA (see [67]). In good encryption schemes, however, it is hard to find messages and keys that violate strong encryption. The message algebra of SPL is making an assumption, an idealisation about the underlying cryptography, the *strong encryption assumption*. This assumption is common in other models like the Spi calculus [5], the inductive approach of Paulson [61], strand spaces [90] and model checking approaches (e.g. [20]).

Pairing messages through message concatenation does not give any nested structure to a message tuple so that when matched against a pattern there can be a choice to how to decompose a tuple. For example consider the message given by the tuple

$$A, B, n$$

and consider the following process that inputs a tuple from the space, destructs it in two subcomponents and sends the two separate components onto the space:

$$in\ \psi, \psi' \,.\, out\ \psi \,.\, out\ \psi' \quad .$$

The input action of this process carries a pattern that can match the tuple A, B, n in two different ways: one where the variable ψ is bound to the agent name A and the variable ψ' is bound to the pair B, n, and another one where ψ is bound to A, B and ψ' is bound to the name n. The semantics of SPL does not resolve this choice and therefore nondeterministic behaviour can arise in input actions when concatenated messages are matched.

4.1.7 A submessage relation

We define a submessage relation among messages belonging to the set of messages described by the grammar in Figure 4.1 and we write $M \sqsubseteq M'$ meaning that message M is a subexpression of message M'. More precisely:

Definition 4.1.5 The relation $\sqsubseteq$ is the smallest binary relation on messages defined by the following rules:

$$M \sqsubseteq M\ (1)\ , \quad \frac{M \sqsubseteq N}{M \sqsubseteq N, N'}\ (2)\ , \quad \frac{M \sqsubseteq N}{M \sqsubseteq N', N}\ (3)\ , \quad \frac{M \sqsubseteq N}{M \sqsubseteq \{N\}_k}\ (4) \quad .$$

□

As expected, the submessage relation that we just defined satisfies the following properties:

Proposition 4.1.6 *For all messages M, M', N and for all keys k the relation $\sqsubseteq$ is such that*

1. $M, M' \sqsubseteq N \;\Rightarrow\; M \sqsubseteq N$ *and* $M' \sqsubseteq N$*,*
2. $\{M\}_k \sqsubseteq N \;\Rightarrow\; M \sqsubseteq N$

Proof. We make use of the inductive definition of $\sqsubseteq$ and proceed by rule induction. Suppose $M, M' \sqsubseteq N$ has been derived applying one of the rules in Definition 4.1.5, we distinguish the following cases:

- If axiom (1) was applied then $N = M, M'$. Since $M \sqsubseteq M$ and $M' \sqsubseteq M'$ we derive $M \sqsubseteq N$ and $M' \sqsubseteq N$ using rule (2).
- If rule (2) was applied then $N = N', N''$ for some messages N', N'' and $M, M' \sqsubseteq N'$. By the induction hypothesis $M \sqsubseteq N'$ and $M' \sqsubseteq N'$. We can now apply rule (2) and obtain $M \sqsubseteq N', N''$ and $M' \sqsubseteq N', N''$ as desired.
- The case of rule (3) is analogous to the one of rule (2).
- If rule (3) was applied then $N = \{N'\}_k$ for some message N' and $M, M' \sqsubseteq N'$. By the induction hypothesis $M \sqsubseteq N'$ and $M' \sqsubseteq N'$. Applying rule (4) we obtain $M \sqsubseteq \{N'\}_k$ and $M' \sqsubseteq \{N'\}_k$ as we wanted.

Along the same lines the proof of 2. □

The size of a message is a natural number and is defined by structural induction as follows:

$$\begin{array}{lcl} size(v) & = & 1 \\ size(k) & = & 1 \\ size(M, M') & = & size(M) + size(M') \\ size(\{M\}_k) & = & size(M) + 1 \quad . \end{array}$$

Let M, N be messages such that $M \sqsubseteq N$. Their size is $size(M) \leq size(N)$.

Proposition 4.1.7 *The relation $\sqsubseteq$ is a partial order among messages.*

Proof. The submessage relation is reflexive by definition. We need to show that it is antisymmetric and transitive.

- Let M, N be messages such that $M \sqsubseteq N$ and $N \sqsubseteq M$. Suppose that $M \sqsubseteq N$ follows from rule (2), then $N = N_1, N_2$ where N_1 and N_2 are messages and $M \sqsubseteq N_1$. From Proposition 4.1.6 it follows that $N_1 \sqsubseteq M$, a contradiction since

  $$size(M) \leq size(N_1, N_2) < size(N_1) \leq size(M) \quad .$$

  One reaches a similar contradiction when $M \sqsubseteq N$ follows from rule (3) and (4). Only rule (1) is admitted and therefore $M = N$.
- Let M, N, T be messages such that $M \sqsubseteq N$ and $N \sqsubseteq T$. By well founded induction on the size of N and making use of Proposition 4.1.6 one can show that $M \sqsubseteq T$.

□

4.1.8 Encryptions and Signatures in SPL

The language SPL includes constructs for both public and symmetric key cryptography. If the underlying encryption scheme is a symmetric-key scheme then $\{M\}_{Key(A,B)}$ denotes an encryption of the message M using $Key(A,B)$, a key shared between agents A and B. We make another idealisation and assume that a symmetric key encryption is non-malleable – an attacker can't, given a cyphertext, produce another cyphertext so that both have related cleartexts. In practise non-malleability is usually achieved by adding a message authentication scheme (MAC) to the encryption schema (for example the CBC encryption scheme with the DES cypher; although very common in practise this scheme does not ensure a very high degree of security [33]). If a public-key encryption scheme is used then $\{M\}_{Pub(A)}$ denotes an encryption of M using the key $Pub(A)$, a public key of agent A. We assume that public-key encryption are non-malleable too. Non-malleable public-key cryptography has been studies for example in [27]. One writes $\{M\}_{Priv(A)}$ to denote the signature of M with the private key $Priv(A)$. Usually the hash code of a message, instead of the message itself, is encrypted with the private key to produce a signature. One can easily extend the syntax of SPL to include hashing and MAC's and treat them in a more explicit way.

The input patterns of SPL provide a convenient way of treating decryption. Let k be a key then, due to the *strong encryption* (Property 4.1.4), the process

$$in\ \{\psi\}_k \, . \, \cdots$$

can match and receive only messages of the from $\{M\}_k$, extracting the cleartext M and binding it to the variable ψ so that it can be used later in the process term. A decryption is performed even if the process term does not explicitly mention how decryption happens. In our model, where cryptography is assumed to be unbreakable, a decryption succeeds only if the right key for deciphering is available. One assumes that:

> Whenever a cyphertext appears on an input pattern, then the right "decryption key" is available to the agent performing that input.

If, for example, k is a public key $Pub(A)$ then the process

$$in\ \{\psi\}_{Pub(A)} \, . \, \cdots$$

can match and receive the cyphertext $\{n, B, A\}_{Pub(A)}$. It also extracts the cleartext n, B, A binding it to ψ. The next action publishes the cleartext on the network. The agent executing this process is in possession of $Priv(A)$, the private key of agent A.

Input patterns not only can denote decryption, they can also stand for signature verification. If, for example, k is the private key $Priv(A)$, then the process

$$in\ \{\psi\}_{Priv(A)} \, . \, \cdots$$

can match and receive a message of the form $\{M\}_{Priv(A)}$ and ensures that the message is signed with the private key of A. Also in this case the way the signature is verified is not made explicit. As before, the agent executing that input process will be in possession of the public key $Pub(A)$ so that it can verify the signature in the prescribed way.

We mentioned in the introduction that we implemented SPL. In giving an implementation of the language the availability of keys becomes an issue: it is unrealistic to assume that all keys that appear on a process term executed by an agent are available. Keys need to be generated and if necessary distributed before they can be used to encrypt, decrypt, sign messages or verify signatures (see [13, 21]).

One can express key-generation in an SPL process by generating a new name and use the name as part of a key-expression, for example as follows:

$$\mathit{out}\,\mathit{new}(x)\ \{\mathit{Key}(x)\}_{\mathit{Pub}(A)} \quad .$$

If m was chosen as a new name on the previous output action then the key expression $\mathit{Key}(m)$ is new as well and stands for a freshly generated key.

4.1.9 NSL as a process

As an illustration, we program the NSL protocol in our language, and so formalise the description given in the introductory chapter of this thesis. While programming NSL we are forced to formalise aspects that are implicit in the informal description, such as the creation of new nonces, the decryption of messages and the matching of nonces. We assume given a set of agent names, **Agents**, of agents participating in the protocol. We program a situation in which agents participating in the NSL protocol can play two roles, as initiator and responder with any other agent. Let $A, B \in \mathbf{Agents}$.

Abbreviate with $\mathit{Init}(A, B)$ the program of initiator A communicating with B:

$$\begin{aligned} \mathit{Init}(A,B) \quad \equiv \quad & \mathit{out}\,\mathit{new}(x)\ \{x, A\}_{\mathit{Pub}(B)}. \\ & \mathit{in}\ \{x, y, B\}_{\mathit{Pub}(A)}. \\ & \mathit{out}\ \{y\}_{\mathit{Pub}(B)} \quad . \end{aligned}$$

The initiator A first generates a new nonce, encrypts the nonce together with its own identifier under the public key of B, the intended responder, and sends it to B. The first action of the initiator's process,

$$\mathit{out}\,\mathit{new}(x)\ \{x, A\}_{\mathit{Pub}(B)} . \cdots \ ,$$

makes all this explicit binding a freshly chosen name denoting a nonce to x. Whenever an encrypted message appears on an output action, one assumes that the agent executing the action is in possession of the key, here $\mathit{Pub}(B)$, and can perform the encryption. In the second action

$$\cdots \mathit{in}\ \{x, y, B\}_{\mathit{Pub}(A)} . \cdots$$

the variable x is bound by an outer $\mathit{new}\,x$ and so by convention is not a pattern variable, and has to be that nonce sent out earlier. The variable y instead is a pattern variable. A message is received only when it matches the pattern of the input action: it is a cyphertext obtained encrypting a text with the public key of the initiator A and the text is a triple whose first element is the nonce sent out earlier by A, the second element is a name, and the third element is the name B. As we mentioned earlier, the input pattern $\{x, y, B\}_{\mathit{Pub}(A)}$ should be thought of as a decryption – one assumes that the agent that executes the action is in possession of the right key to decrypt, in this case $Priv(A)$, and therefore can recover a cleartext of the from n, m, B with n, m names. The last action performed by the initiator in a protocol session is

$$\cdots out\ \{y\}_{Pub(B)}\ .$$

The initiator sends to B an encryption with B's public key. It is an encryption of the nonce which was apparently received from B and bound to the variable y in the previous input action.

Abbreviate with $Resp(B)$ the program of responder B:

$$\begin{aligned} Resp(B) \quad \equiv \quad & in\ \{x, Z\}_{Pub(B)}. \\ & out\,new(y)\ \{x, y, B\}_{Pub(Z)}. \\ & in\ \{y\}_{Pub(B)} \quad . \end{aligned}$$

The agent B responds to an initiator that produces a message encrypted with the public key of B and that contains a nonce and the name of an agent. The responder takes the agent name obtained from the first decryption and bounds it to the variable Z. The responder treats that name as the name of the initiator with whom to carry out the rest of the protocol session.

4.1.10 The process for a spy

The Dolev-Yao model (see [28]) describes a powerful attacker that attempts to tamper the security of a protocol in several ways. It is more interesting to study the behaviour of a security protocol together with such powerful attacker rather than studying its behaviour in isolation. The attacker, or "Spy" can be described with an SPL process term. The spy can eavesdrop all communication that is transmitted over then network and manipulate messages in several ways:

1. It can compose different messages into a message tuple

   $$Spy_1 \equiv in\ \psi_1\,.\,in\ \psi_2\,.\,out\ \psi_1, \psi_2 \quad .$$

2. It can decompose a composite message into more components

   $$Spy_2 \equiv in\ \psi_1, \psi_2\,.\,out\ \psi_1\,.\,out\ \psi_2 \quad .$$

   The first input action matches any message that is a tuple with at least two components.

3. It can encrypt any message with the keys that are available. In case of public key encryption one can assume that all public keys of all agents are known. Therefore the spy knows the public key as soon as it knows an agent name:

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

   In case of shared key encryption one can not assume in general that the spy knows the shared key of other trusted agents. Therefore writing

   $$in\ x\,.\,in\ y\,.\,in\ \psi\,.\,out\ \{\psi\}_{Key(x,y)}$$

   would give unwanted power to the spy. The spy can only encrypt if it can eavesdrop and therefore get hold of a shared key from the network:

   $$Spy_4 \equiv in\ Key(x, y)\,.\,in\ \psi\,.\,out\ \{\psi\}_{Key(x,y)} \quad .$$

4. An attacker can decrypt messages with available keys. As before one can not assume that the spy knows all secret keys of the agents participating in a protocol, but it might get hold of one if transmitted in cleartext over the network. For public key cryptography this can be described as:

$$Spy_5 \equiv in\ Priv(x)\,.\,in\ \{\psi\}_{Pub(x)}\,.\,out\ \psi$$

and for shared key cryptography as:

$$Spy_6 \equiv in\ Key(x,y)\,.\,in\ \{\psi\}_{Key(x,y)}\,.\,out\ \psi \quad .$$

5. A spy can sign with the private keys that are eavesdropped from the network:

$$Spy_7 \equiv in\ Priv(x)\,.\,in\ \psi\,.\,out\ \{\psi\}_{Priv(x)} \quad .$$

6. Other capabilities can be added to a spy. For example the spy can verify any signature as soon as it knows the name of the agent signing the message

$$Spy_8 \equiv in\ x\,.\,in\ \{\psi\}_{Priv(x)}\,.\,out\ \psi$$

or it can create new (random) values

$$Spy_9 \equiv out\, new(\vec{n})\ \vec{n} \quad .$$

Not much power is added to a spy when one of those two capabilities is explicitly included in the process of the spy. It may, however, be useful to give explicit signature verification capabilities to the spy as a way to give the spy access to the signed message. As for new name generation, the initial message set from which a protocol run starts may already contain names that can be used by the spy as if they were new.

There is no need to add message duplication and suppression as explicit spy operations. Messages in SPL are considered as persistent so that once a message is sent it stays available on the network. There is therefore no need to duplicate messages. In this thesis we focus on security properties that can be described as properties of all possible finite behaviours of a protocol – included are behaviours where some messages are never received and therefore can be understood as suppressed by a spy. Choosing a different program for the spy means restricting or augmenting its power, for example to passive eavesdropping or active falsification.

The attacker is described with the parallel composition of various components, each describing a different capability. The spy uses the network, the space of messages, as a buffer to store intermediate results and so construct complicated messages or decompose them into basic values. The various components of a spy are replicated so that an operation can be applied over and over again to a message. In some cases, depending on the protocol to be analysed, one might be interested only in encryption and public key cryptography rather than for example signatures or shared key cryptography. For the NSL protocol, for example, a spy might have the following components:

$$Spy_{NSL} \equiv !\ \|_{i\in\{1,2,3,5\}}\ Spy_i \quad .$$

4.1.11 NSL in a malicious environment

The NSL protocol system together with an attacker is described by putting all the components together. Components are replicated, to model multiple concurrent runs of the protocol. Let $\mathcal{A}$ be the set of agents participating in the system. Here we describe a situation where each agent can engage in the protocol both as initiator and as responder with any other agent:

$$\begin{array}{lcl} P_{init} & \equiv & \|_{(A,B)\in\mathcal{A}\times\mathcal{A}} \; ! \; Init(A,B) \\ P_{resp} & \equiv & \|_{A\in\mathcal{A}} \; ! \; Resp(A) \\ P_{spy} & \equiv & Spy_{NSL} \\ \\ NSL & \equiv & \|_{i\in\{resp,init,spy\}} \, P_i \quad . \end{array}$$

4.2 Transition semantics

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

4.2.1 Configurations, actions, and transitions

A *configuration* consists of a triple $\langle p, s, t\rangle$ where p is a closed process term, s is a subset of names $\mathbf{N}$, and t is a subset of closed (i.e., variable-free) messages. We say that a configuration is *proper* iff the names in p and t are included in s. A closed process p when inputting a message or generating a new name acts in the context of t, the messages that have been output, and s, the names used so far.

Actions are either input actions or output actions, possibly tagged with indices to keep track of parallel component at which they occur:

$$\alpha ::= out\, new(\vec{n})\; M \mid in\, M \mid i : \alpha$$

where M is a closed message, $\vec{n}$ are names and i is an index drawn from **Indexes**. New names can be generated and included in a message that is sent out – output actions record the new names that are generated. We write $out\, M$ for an output action, outputting a message M, where no new names are generated.

The way configurations evolve is expressed by *transitions*

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

given by the following rules:

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

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

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

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

(parallel composition)

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

where p'_i is p'_j for $i = j$, else p_i.

Proper configurations evolve through transitions into configurations that are proper as well.

Proposition 4.2.1 *Let α be an action. Suppose that $\langle p, s, t\rangle$ and $\langle p', s', t'\rangle$ are configurations, and that $\langle p, s, t\rangle$ is proper. If $\langle p, s, t\rangle \xrightarrow{\alpha} \langle p', s', t'\rangle$, then $\langle p', s', t'\rangle$ is also proper.*

Proof. Easy rule induction on the transition rules. □

4.2.2 Security properties and transition semantics

The transition semantics allows us to state formally many security properties. In this thesis we focus on security properties that are expressible as safety properties. For a process term p they can therefore be expressed on the set of finite sequences of configurations and transitions describing runs of the protocol

$$\langle p, s_0, t_0\rangle \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_w} \langle p_w, s_w, t_w\rangle \xrightarrow{\alpha_{w+1}} \cdots$$

starting from a proper configuration $\langle p, s_0, t_0\rangle$.

A common security property of a protocol is *secrecy* of data that is exchanged over the network – in every run of the protocol the secret is never leaked to undesired, corrupted or even malicious agents. In our setting to show that a message is secret it often suffices to show that the message never appears in the clear and by itself on the network, the space of messages. In fact if the spy gets hold of a secret it can publish it as a stand alone message. For the NSL protocol, for example, one shows that for every protocol run the nonce of the responder never appears in cleartext on the network, provided private keys are not leaked from the start:

Secrecy of the responder's nonce: Let $A_0, B_0 \in \mathcal{A}$ be two names of agents participating in the protocol. For all runs

$$\langle NSL, s_0, t_0\rangle \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_w} \langle p_w, s_w, t_w\rangle \xrightarrow{\alpha_{w+1}} \cdots$$

where $\langle NSL, s_0, t_0\rangle$ is proper and where $Priv(A_0)$ and $Priv(B_0)$ do not appear as content of any message in t_0, if at some stage w in the run

$$\alpha_w = resp : B_0 : i_0 : out\, new(n_0)\ \{m_0, n_0, B_0\}_{Pub(A_0)}$$

where i_0 is an index and n_0, m_0 are names, then at all stages v in the run $n_0 \notin t_v$.

Another interesting security property is *authentication* among the participants in a protocol. Making use of the SPL-transition semantics one can describe authentication properties following Lowe's style (see [47]): Whenever an initiator A_0 performs a run to which apparently agent B_0 responds, then agent B_0 responded with matching actions apparently to A_0. The NSL protocol, for example, satisfies the following agreement property:

An authentication guarantee for the initiator: Let $A_0, B_0 \in \mathcal{A}$ be two names of agents participating in the protocol. For all runs

$$\langle NSL, s_0, t_0\rangle \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_w} \langle p_w, s_w, t_w\rangle \xrightarrow{\alpha_{w+1}} \cdots$$

where $\langle NSL, s_0, t_0\rangle$ is proper and where $Priv(A_0)$ and $Priv(B_0)$ do not appear as content of any message in t_0, if at some stage w in the run

$$\alpha_w = init : (A_0, B_0) : j_0 : in\ \{m_0, n_0, B_0\}_{Pub(A_0)}$$

where j_0 is an index and n_0, m_0 are names, then at some previous stage v in the run

$$\alpha_v = resp : B_0 : i_0 : out\, new\, n_0\ \{m_0, n_0, B_0\}_{Pub(A_0)}$$

where i_0 is an index.

The transition semantics, however, does not support directly local reasoning of the kind one might wish to apply in the analysis of security protocols. To give an idea of the difficulty, imagine one wanted to prove the secrecy property for the nonce of an NSL responder. A reasonable way to prove such a property is to find a stronger invariant, a property which can be shown to be preserved by all the actions of the process. Equivalently, one can assume that there is an earliest action α_v in a run which violates the invariant, and derive a contradiction by showing that this action must depend on a previous action, which itself violates the invariant.

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

4.3 Event-based semantics

In this section we introduce an alternative semantics for SPL which makes events of a protocol and their dependencies more explicit. We must first address the issue of what constitutes an event of a security protocol. Here, we follow the lead from Petri nets (see Chapter 3, [66, 95]), and define events in terms of how they affect conditions. It turns out that the events of SPL processes together with the conditions and the pre- and postcondition maps between events and conditions, which are implicit in our definition of events, from a basic Petri-net where some conditions are persistent (see Chapter 3).

4.3.1 Conditions

Conditions are to represent some form of local state and we discern conditions of three kinds: *control*, *output* and *name* conditions.

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

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

where $i \in$ **Indexes**. A condition in **C** stands for the point of control in a (single-thread) process. When C is a subset of control conditions we will write $i : C$ to mean $\{i : b \mid b \in C\}$. We define the *initial control conditions* of a closed process term p, to be the subset $Ic(p)$ of **C**, given by the following structural induction:

$$\begin{array}{lcl} Ic(out\, new(\vec{x})\ M\,.\,p) & = & \{out\, new(\vec{x})\ M\,.\,p\} \\ Ic(in\, pat\, \vec{x}\vec{\chi}\vec{\psi}\ M\,.\,p) & = & \{in\, pat\, \vec{x}\vec{\chi}\vec{\psi}\ M\,.\,p\} \\ Ic(\|_{i\in I} p_i) & = & \bigcup_{i\in I} i : Ic(p_i) \end{array}$$

where the last case also includes the base case *nil*, when the indexing set is empty. We sometimes write $index(b)$ for the index of a control condition b.

Proposition 4.3.1 *If p is a closed SPL-process then for all $b, b' \in Ic(p)$ such that $b \neq b'$ the $index(b)$ is never prefix of $index(b')$.*

Proof. Obvious induction on the structure of closed process terms. □

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

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

4.3.2 Events

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

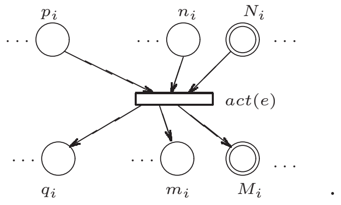
$$\mathcal{P}ow(\mathbf{C}) \times \mathcal{P}ow(\mathbf{O}) \times \mathcal{P}ow(\mathbf{N}) \times \mathcal{P}ow(\mathbf{C}) \times \mathcal{P}ow(\mathbf{O}) \times \mathcal{P}ow(\mathbf{N}) \quad .$$

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

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

where ${}^{c}e$ is the set of **C**-preconditions of e, e^c is the set of **C**-postconditions of e, etc. Write ${}^{\cdot}e$ for ${}^{c}e \cup^{o} e \cup^{n} e$, all preconditions of e, and $e^{\cdot}$ for all postconditions $e^c \cup e^o \cup e^n$. Earlier in the transition semantics we used actions α to specify the nature of transitions. An event e is associated with a unique action $act(e)$.

Sometimes it is convenient to represent events graphically:



Conditions are drawn as circles and persistent conditions are highlighted as double circles. The square represents the event itself which is connected to its preconditions by incoming edges and to its postconditions by outgoing edges.

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

Output events

$$\mathbf{Out}(out\,new(\vec{x})\,M.p;\vec{n}) \quad ,$$

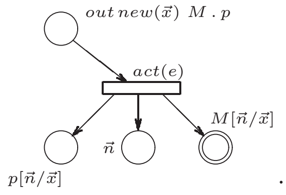
where $\vec{n} = n_1, \cdots, n_l$ are *distinct* names to match the variables $\vec{x} = x_1, \cdots, x_l$, consist of an event e with these pre- and postconditions:

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

The *action* of an output event is

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

Output can be graphically represented as following:



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

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

Input events

$$\mathbf{In}(in\,pat\,\vec{x}\vec{\chi}\vec{\psi}\,M.p;\vec{n},\vec{k},\vec{L}) \quad ,$$

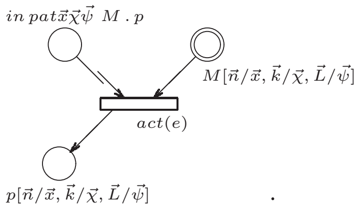
where $\vec{n}$ is a list of names to match $\vec{x}$, $\vec{k}$ a list of closed key expressions to match $\vec{\chi}$, and $\vec{L}$ is a list of closed messages to match $\vec{\psi}$, consist of an event e with these pre- and postconditions:

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

The action of an input event is

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

Graphically input events are drawn as following:


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

.

Indexed events

$$i : e \quad ,$$

where $e \in$ **Events**, $i \in$ **Indices**, and

$$\begin{array}{lllllllll} {}^c(i:e) & = & i:{}^ce\ , & {}^o(i:e) & = & {}^oe\ , & {}^n(i:e) & = & {}^ne\ , \\ (i:e)^c & = & i:e^c\ , & (i:e)^o & = & e^o\ , & (i:e)^n & = & e^n \quad . \end{array}$$

The action of an indexed event is $act(i:e) = i:act(e)$. When E is a subset of events we will generally use $i:E$ to mean $\{i:e \mid e \in E\}$.

4.3.3 A net from SPL

In defining the set of conditions and, inductively, the set of events, we have in fact defined a (rather large) net from the syntax of SPL. The SPL-net has:

- conditions $\mathbf{C} \cup \mathbf{O} \cup \mathbf{N}$,
- events **Events**,
- precondition map pre : **Events** $\rightarrow \mathcal{P}ow(\mathbf{C} \cup \mathbf{O} \cup \mathbf{N})$ such that $pre(e) = {}^{\cdot}e$ for every event $e \in$ **Events**,
- and postcondition map $post$: **Events** $\rightarrow \mathcal{P}ow(\mathbf{C}\cup\mathbf{O}\cup\mathbf{N})$ such that $post(e) = e^{\cdot}$ for every event $e \in$ **Events**.

Its markings $\mathcal{M}$ will be subsets of conditions and so of the form

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

where $c \subseteq \mathbf{C}$, $s \subseteq \mathbf{N}$, and $t \subseteq \mathbf{O}$. By assumption the set of conditions $\mathbf{O}$ are persistent so the net is a basic net with persistent conditions (see Chapter 3).

Definition 4.3.2 (*Token game for the SPL net*). Letting $c \cup s \cup t$ and $c' \cup s' \cup t'$ be two markings,

$$c \cup s \cup t \xrightarrow{e} c' \cup s' \cup t' \ \text{ iff}$$

i) ${}^{\cdot}e \subseteq c \cup s \cup t \quad \& \quad e^c \cap c = \emptyset \quad \& \quad e^n \cap s = \emptyset$ and

ii) $c' = (c \setminus {}^ce) \cup e^c \quad \& \quad s' = s \cup e^n \quad \& \quad t' = t \cup e^o \quad .$

□

In particular, observe that the occurrence of e begins the holding of its name post-conditions e^n - these names have to be distinct from those already in use to avoid contact.

4.3.4 The net of an SPL process

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

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

$$
\begin{aligned}
&Ev(out\, new(\vec{x})\, M.p) = \\
&\qquad \{\mathbf{Out}(out\, new(\vec{x})\, M.p; \vec{n}) \mid \vec{n} \text{ distinct names}\} \\
&\quad \cup \bigcup \{Ev(p[\vec{n}/\vec{x}]) \mid \vec{n} \text{ distinct names}\}
\end{aligned}
$$

$$
\begin{aligned}
&Ev(in\, pat\vec{x}\vec{\chi}\vec{\psi}\, M.p) = \\
&\qquad \{\mathbf{In}(in\, pat\vec{x}\vec{\chi}\vec{\psi}\, M.p; \vec{n}, \vec{k}, \vec{L}) \mid \vec{n} \text{ names }, \vec{k} \text{ closed keys}, \vec{L} \text{ closed messages}\} \\
&\quad \cup \bigcup \{Ev(p[\vec{n}/\vec{x}, \vec{k}/\vec{\chi}, \vec{L}/\vec{\psi}]) \mid \vec{n} \text{ names }, \vec{k} \text{ closed keys}, \vec{L} \text{ closed messages}\}
\end{aligned}
$$

$$
Ev(\|_{i \in I} p_i) = \bigcup_{i \in I} i : Ev(p_i)
$$

If an event belongs to the set of events of a closed process term then so do all the events in **Events** that share the same control condition.

Proposition 4.3.3 *Let e, e' be events in* **Events** *and p a closed process term. If $e \in Ev(p)$ and ${}^c e' = {}^c e$ then $e' \in Ev(p)$.*

Proof. By induction on the size of a closed process and by definition of **Events** and $Ev(p)$. $\square$

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

The control conditions of a process term are all control conditions that appear as preconditions on the events of a process term.

Definition 4.3.4 *Let p be a closed process term. Define its* control-conditions *by ${}^c p = \bigcup \{{}^c e \mid e \in Ev(p)\}$.* $\square$

As expected, the control conditions of a closed process include all initial control conditions of the process, and all control postconditions of the events of the process.

Proposition 4.3.5 *Let p be a closed process term.*

1. *$Ic(p) \subseteq {}^c p$,*

2. *if $e \in Ev(p)$ then $e^c \subseteq {}^c p$.*

Proof. The proof of both properties is based on an induction on the size of closed processes. The size of a close process is defined in Section 4.1.5.

1. With an easy induction on the size of a closed process and making use of the definition of initial control conditions and events of a closed process one shows that for every control condition $c \in Ic(p)$ there exists an event $e \in Ev(p)$ such that $c = {}^ce$. Then from the definition of ${}^cp$, the control conditions of a closed process, it follows that $Ic(p) \subseteq {}^cp$.

2. Inductively on the size of a closed process distinguishing the following cases:

   Suppose $e \in Ev(out\,new(\vec{x})\ M\,.\,p)$ then either
   $$e \in Ev(p[\vec{n}/\vec{x}]) \text{ or } e = \mathbf{Out}(out\,new\vec{x}M.p;\vec{n})$$
   where $\vec{n}$ is a list of names. Consider the first case, then from the induction hypothesis it follows that $e^c \subseteq {}^cp[\vec{n}/\vec{x}]$. Moreover ${}^cp[\vec{n}/\vec{x}] \subseteq {}^cp$ since $Ev(p[\vec{n}/\vec{x}]) \subseteq Ev(p)$. Therefore $e^c \subseteq {}^cp$ as desired.

   The case where $e \in Ev(in\,pat\,\vec{x}\vec{\chi}\vec{\psi}\ M\,.\,p)$ is analogous to the previous one.

   Suppose $e \in Ev(\|_{i\in I}p_i)$ then there exists an index $j \in I$ such that $e \in Ev(p_j)$ and therefore it follows from the induction hypothesis that $e^c \subseteq {}^cp_j$. Moreover ${}^cp_j \subseteq {}^c\|_{i\in I}p_i$ since $Ev(p_j) \subseteq Ev(\|_{i\in I}p_i)$ and therefore $e^c \subseteq {}^c\|_{i\in I}p_i$ as desired.

□

If the control preconditions of an event are included in the control conditions of a closed process term then that event is reachable in the behaviour of that process.

Proposition 4.3.6 *Let p be a closed process term and $e \in$ **Events**:*
$$ {}^ce \subseteq {}^cp \textit{ iff } e \in Ev(p) \quad . $$

Proof. The "if" part follows obviously from the definition of the control conditions of a closed process.

The "only if" part: Let ${}^ce \subseteq {}^cp$, then there exists an event $e' \in Ev(p)$ such that ${}^ce' = {}^ce$. From Proposition 4.3.3 it follows that $e \in Ev(p)$. □

Theorem 4.3.7 *Let $\mathcal{M} \cap \mathbf{C} \subseteq p^c$. Let $e \in$ **Events**. Then, $\mathcal{M} \xrightarrow{e} \mathcal{M}'$ in the SPL-net iff $e \in Ev(p)$ & $\mathcal{M} \xrightarrow{e} \mathcal{M}'$ in $Net(p)$.*

Proof. The "if" part is clear. The "only if" part follows from Proposition 4.3.6. □

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

Corollary 4.3.8 *Let p be a closed process term and let*
$$\mathcal{M}_0 \xrightarrow{e_1} \cdots \xrightarrow{e_w} \mathcal{M}_w \xrightarrow{e_{w+1}} \cdots$$
be a sequence of event transitions in the SPL net such that $\mathcal{M}_0 \cap \mathbf{C} \subseteq {}^cp$. At every stage $w > 0$ in the run $e_w \in Ev(p)$ and therefore $\mathcal{M}_{w-1} \xrightarrow{e_w} \mathcal{M}_w$ is in $Net(p)$.

Proof. By induction on the sequence of event transitions we show that at every stage w in the run $\mathcal{M}_w \cap \mathbf{C} \subseteq {}^{c}p$ and $e_w \in Ev(p)$. At stage 0 we assumed $\mathcal{M}_0 \cap \mathbf{C} \subseteq {}^{c}p$ then from Theorem 4.3.7 it follows that $e_0 \in Ev(p)$. Suppose that at some stage w in the sequence of event transitions $\mathcal{M}_w \cap \mathbf{C} \subseteq {}^{c}p$ and $e_w \in Ev(p)$. From the token game on the SPL net (Definition 4.3.2) it follows that

$$ {}^{c}e_w \subseteq {}^{c}p \text{ and } \mathcal{M}_w \cap \mathbf{C} = ((\mathcal{M}_{w-1} \cap \mathbf{C}) \setminus {}^{c}e_w) \cup e_w^c $$

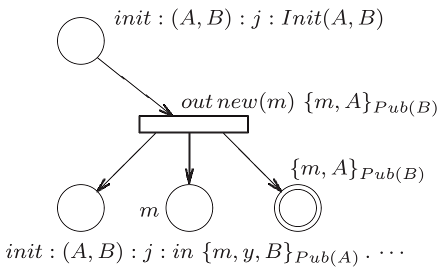
Therefore $\mathcal{M}_w \cap \mathbf{C} \subseteq {}^{c}p \cup {}^{c}e_w$. By the induction hypothesis $e_w \in Ev(p)$, and by Proposition 4.3.5 $e_w^c \subseteq {}^{c}p$. As desired we obtain $\mathcal{M}_w \cap \mathbf{C} \subseteq {}^{c}p$ and from Theorem 4.3.7 it follows that $e_w \in Ev(p)$. □

4.3.5 The NSL events

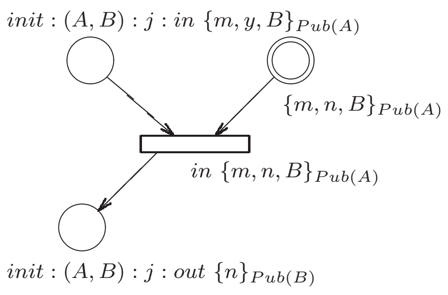
As an example we show the events of the NSL system that we described in Section 4.1.11. We distinguish between events for the initiators, the responders and the spy. The set $\mathcal{A}$ is the set of names of agents that are participating in the protocol. We choose to join spy events together at shared control conditions, to give a better idea of the dependencies among them. We classify the events $Ev(NSL)$ involved in the NSL protocol as following:

Initiator events for every agent names $A, B \in \mathcal{A}$, names $m, n \in \mathbf{N}$, and session indices $j \in \omega$.

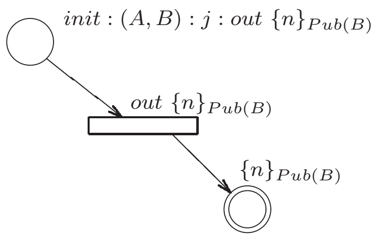
$init : A, B : j : \mathbf{Out}(Init(A, B); m)$


$init : (A, B) : j : Init(A, B)$
$out\, new(m)\ \{m, A\}_{Pub(B)}$
$\{m, A\}_{Pub(B)}$
m
$init : (A, B) : j : in\ \{m, y, B\}_{Pub(A)} \,.\, \cdots$

$init : A, B : j : \mathbf{In}(in\ \{m, y, B\}_{Pub(A)} \,.\, out\ \{y\}_{Pub(B)}; n)$

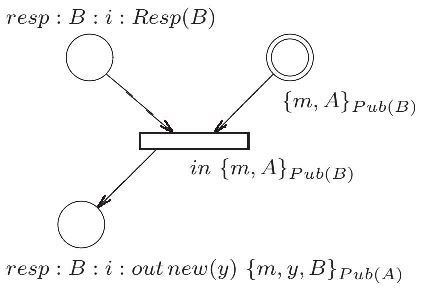

$init : (A, B) : j : in\ \{m, y, B\}_{Pub(A)}$
$\{m, n, B\}_{Pub(A)}$
$in\ \{m, n, B\}_{Pub(A)}$
$init : (A, B) : j : out\ \{n\}_{Pub(B)}$

$init : A, B : j : \mathbf{Out}(out\ \{n\}_{Pub(B)})$

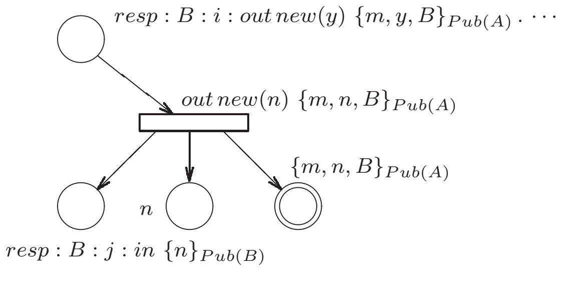


Responder events for every agent names $A, B \in \mathcal{A}$, names $m, n \in \mathbf{N}$, and session indices $i \in \omega$

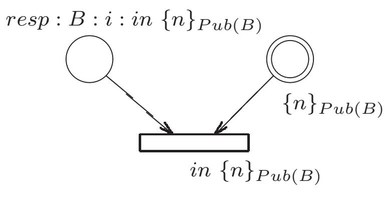
$resp : B : i : \mathbf{In}(Resp(B); m, A)$



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

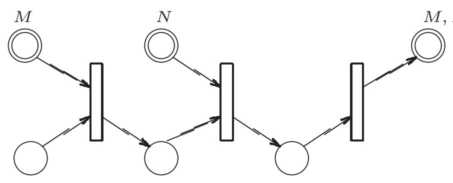

$resp : B : i : out\, new(y)\ \{m, y, B\}_{Pub(A)} \,.\, \cdots$
$out\, new(n)\ \{m, n, B\}_{Pub(A)}$
$\{m, n, B\}_{Pub(A)}$
n
$resp : B : j : in\ \{n\}_{Pub(B)}$

$resp : B : i : \mathbf{In}(in\ \{n\}_{Pub(B)})$

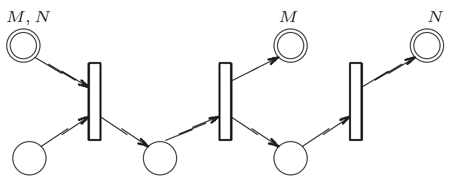


Spy events for every M, N messages and names n, m. For simplicity we only display spy events graphically joining them together at control conditions.

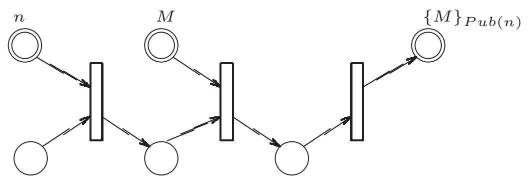
Composition $Spy_1 \equiv in\ \psi\,.\,in\ \psi'\,.\,out\ \psi, \psi'$:



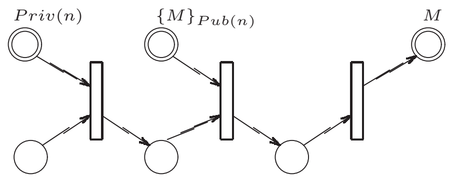
Decomposition $Spy_2 \equiv in\ \psi, \psi'\,.\,out\ \psi\,.\,out\ \psi'$:



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



Decryption $Spy_5 \equiv in\ Priv(x)\,.\,in\ \{\psi\}_{Pub(x)}\,.\,out\ \psi$:



Other spy events whenever included in the system need to be considered. They all are of a similar kind of those described here.

4.4 Relating transition and event-based semantics

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

Theorem 4.4.1

i) If

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

then

$$Ic(p) \cup s \cup t \xrightarrow{e} Ic(p') \cup s' \cup t'$$

for some $e \in \mathbf{Events}$ *with* $act(e) = \alpha$.

ii) If

$$Ic(p) \cup s \cup t \xrightarrow{e} \mathcal{M}'$$

in the SPL-net, then

$$\langle p, s, t\rangle \xrightarrow{act(e)} \langle p', s', t'\rangle \quad and \quad \mathcal{M}' = Ic(p') \cup s' \cup t' \ ,$$

for some closed process term p'*, for some* $s' \subseteq \mathbf{N}$*, and* $t' \subseteq \mathbf{O}$*.*

Proof. Both $i)$ and $ii)$ are proved by induction on the size of p.

i) Consider the possible forms of a closed process p and assume that

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

Case $p \equiv out\,new(\vec{x})\ M\,.\,q$: There must be, by the operational transition rules, distinct names $\vec{n} = n_1, \cdots n_l$, not in s, for which $\alpha = out\,new(\vec{n})\ M[\vec{n}/\vec{x}]$ and $p' \equiv q[\vec{n}/\vec{x}]$. The initial conditions $Ic(p)$ form the singleton set $\{p\}$, therefore the output event

$$e = \mathbf{Out}(out\,new\vec{x})\ M\,.\,q; \vec{n})$$

is enabled at the marking $\{p\} \cup s \cup t$, its action is α, and

$$Ic(p) \cup s \cup t \xrightarrow{e} Ic(q[\vec{n}/\vec{x}]) \cup s' \cup t' \ .$$

Case $p \equiv in\,pat\,\vec{x}\vec{\chi}\vec{\psi}\ M\,.\,q$: This case is similar to the one for output.

Case $p \equiv \|_{i\in I} p_i$: By the operational rules, there must be

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

with $\alpha = j : \beta$ and $p' \equiv \|_{i\in I} p'_i$, where $p'_i = p_i$ whenever $i \neq j$. Inductively,

$$Ic(p_j) \cup s \cup t \xrightarrow{e} Ic(p'_j) \cup s' \cup t'$$

for some event e such that $act(e) = \beta$. It is now easy to check that

$$Ic(p) \cup s \cup t \xrightarrow{j:e} Ic(\|_{i\in I} p'_i) \cup s' \cup t' \ .$$

ii) Assume that

$$Ic(p) \cup s \cup t \xrightarrow{e} c' \cup s' \cup t'$$

for $c' \subseteq \mathbf{C}$, $s' \subseteq \mathbf{N}$, and $t' \subseteq \mathbf{O}$. Consider the possible forms of the closed process term p.

Case $p \equiv out\,new(\vec{x})\ M\,.\,q$: Note that $Ic(p) = \{p\}$. By the definition of **Events**, the only possible events with concession at $\{p\} \cup s \cup t$, are the ones of the form

$$e = \mathbf{Out}(out\,new(\vec{x})\ M\,.\,q;\vec{n})\ ,$$

for some choice of distinct names $\vec{n}$ not in s. The occurrence of e would, by the token game (Definition 4.3.2), make $c' = Ic(q[\vec{n}/\vec{x}])$, $s' = s \cup \{\vec{n}\}$ and $t' = t \cup \{M[\vec{n}/\vec{x}]\}$. Clearly, from the transition semantics,

$$\langle p, s, t\rangle \xrightarrow{act(e)} \langle q[\vec{n}/\vec{x}], s', t'\rangle \quad .$$

Case $p \equiv in\,pat\,\vec{x}\vec{\chi}\vec{\psi}\ M\,.\,q$: This case is similar to the one for output.

Case $p \equiv \|_{i\in I} p_i$: Note that $Ic(p) = \bigcup_{i\in I} i : Ic(p_i)$. From the token game (Definition 4.3.2) and by the definition of **Events**, the event e can only have the form $e = j : e'$, where

$$Ic(p_j) \cup s \cup t \xrightarrow{e'} c'_j \cup s' \cup t'$$

and

$$c' = \bigcup_{i\neq j} i : Ic(p_i) \cup j : c'_j \quad .$$

Inductively,

$$\langle p_j, s, t\rangle \xrightarrow{act(e')} \langle p'_j, s', t'\rangle \text{ and } c'_j = Ic(p'_j)$$

for some closed process p'_j. Thus, according to the transition semantics,

$$\langle p, s, t\rangle \xrightarrow{act(e)} \langle \|_{i\in I} p'_i, s', t'\rangle$$

where $p'_i = p_i$ whenever $i \neq j$. Hence, $c' = Ic(\|_{i\in I} p'_i)$.

□

It turns out that all markings reachable in the behaviour of the process have the form

$$\mathcal{M} = Ic(p) \cup s \cup t\ ,$$

for some close process term p, names s, and network conditions t. One can adopt the following definition:

Definition 4.4.2 *Let* $e \in \mathbf{Events}$. *Let* p *be a closed process,* $s \subseteq \mathbf{N}$, *and* $t \subseteq \mathbf{O}$. *Write* $\langle p,\ s,\ t\rangle \xrightarrow{e} \langle p',\ s',\ t'\rangle$ *iff* $Ic(p) \cup s \cup t \xrightarrow{e} Ic(p') \cup s' \cup t'$ *in the SPL-net.*

There is no contact at control conditions, throughout the reachable behaviour of the net:

Proposition 4.4.3 *Let p be a closed process term. Let $e \in$ **Events**. Then,*

$$ {}^{c}e \subseteq Ic(p) \Rightarrow e^{c} \cap Ic(p) = \emptyset $$

Proof. By induction on the size of a closed process. □

Proposition 4.4.4 *If*

$$\langle p_0, s_0, t_0 \rangle \xrightarrow{e_1} \cdots \xrightarrow{e_w} \langle p_w, s_w, t_w \rangle \xrightarrow{e_{w+1}} \cdots ,$$

is a run in the SPL-net then for any two event occurrences with $u \neq w$ in the run ${}^{c}e_u \neq {}^{c}e_w$. As a consequence there are no two different stages carrying the same event.

Proof. We first prove the following property:

> Let u, w be two stages in the run such that $u \leq w$. For all $b \in Ic(p_u)$ and for all $b' \in Ic(p_w)$ if $size(b) < size(b')$ then $index(b')$ is not prefix of $index(b)$.

Suppose the contrary and let w be the closest stage to u for which there is $b' \in Ic(p_w)$ such that $size(b) < size(b')$ and $index(b')$ is prefix of $index(b)$. Obviously $b \neq b'$ and therefore $b' \notin Ic(p_u)$ (Proposition 4.3.1). From the token game it follows that there exists a stage h such that $u < v \leq w$ and $b' \in e_v^c$. Clearly if $b'' \in^c e_v$ then $b'' \in Ic(p_{v-1})$ and $index(b'')$ is prefix of $index(b')$, so $index(b'')$ is prefix of $index(b)$. Moreover $size(b') < size(b'')$ and therefore $size(b) < size(b'')$. If $v - 1 = u$ then from Proposition 4.3.1 it follows that $b'' = b$ yielding a contradiction. If instead $u < v - 1$ then we found a stage closer to u than w violating the desired property, which yields another contradiction.

Let $u < w$ and suppose that ${}^{c}e_u = {}^{c}e_w$. Let $b' \in {}^{c}e_u$ and $b \in e_u^c$. Then we know that $size(b) < size(b')$ and $index(b')$ is prefix of $index(b)$. It follows from the token game that $b \in Ic(p_u)$ and $b' \in Ic(p_{w-1})$. This obviously contradicts the property above. □

Chapter 5

Reasoning about security in the net semantics

To demonstrate the viability of the net semantics as a tool for proving security properties, we use the semantics to derive general proof principles. The principles capture the kind of dependency reasoning found in the strand spaces [88, 90] and inductive methods [61, 62]. To illustrate the principles in action, we apply them to establish secrecy and authentication guarantee of the NSL protocol. The proofs are simplified by a result about the occurrence of spy events in a protocol run – it tells precise conditions under which occurrence of spy events can be excluded. The result is based on the notion of *surroundings* of a message inside of another message. It is a general result that holds for all SPL-protocols and with a similar purpose to the results on strand-space ideals introduced in [88]. We introduce a diagrammatic style of reasoning which we find helpful both in proving some security properties and in showing where other security properties fail. Diagrams like ours have been used in [89] to conveniently describe actions in a protocol run and perhaps to display an attack.

5.1 General proof principles

From the net semantics we can derive several principles useful in proving authentication and secrecy of security protocols.

Convention 5.1.1 In the remaining part of this chapter and in the next chapter reasoning is based on runs in the net of an SPL process. From now on we assume that all runs are proper runs: all configurations $\langle p, s, t\rangle$ in a proper run are such that all names appearing on the process p or as part of messages in t are in s.

5.1.1 Well-foundedness

Every run in the net of an SPL process consists of a sequence of transitions starting from an initial configuration. Clearly, only a finite number of stages in the run separates a configuration from any earlier configuration. Therefore the following principle holds:

Proposition 5.1.2 *(Well-foundedness) Given a property $\mathcal{P}$ on configurations, if a run*

$$\langle p_0, s_0, t_0 \rangle \xrightarrow{e_1} \cdots \xrightarrow{e_w} \langle p_w, s_w, t_w \rangle \xrightarrow{e_{w+1}} \cdots ,$$

contains a configurations s.t. $\mathcal{P}(p_0, s_0, t_0)$ and $\neg\mathcal{P}(p_w, s_w, t_w)$, then there is a stage v, $0 < v \leq w$, such that $\mathcal{P}(p_u, s_u, t_u)$ for all $u < v$ and $\neg\mathcal{P}(p_v, s_v, t_v)$. □

5.1.2 Freshness

During their execution processes can choose new (fresh) names. In the SPL semantics output events mark freshly chosen names. The token game for the SPL net ensures that once a condition is marked it can't be marked again unless it is first consumed. The name conditions that come into play when new names are chosen are never consumed therefore once a name is marked it can't be marked again in the same run. We say that a name $m \in \mathbf{N}$ is *fresh on an event* e if $m \in e^n$ and we write $Fresh(m, e)$.

Proposition 5.1.3 *(Freshness) Within a run*

$$\langle p_0, s_0, t_0 \rangle \xrightarrow{e_1} \cdots \xrightarrow{e_w} \langle p_w, s_w, t_w \rangle \xrightarrow{e_{w+1}} \cdots ,$$

the following properties hold:

i) If $n \in s_w$ then either $n \in s_0$ or there is a previous stage $v \leq w$ such that $Fresh(n, e_v)$.

ii) Given a name n there is at most one stage w such that $Fresh(n, e_w)$.

iii) If $Fresh(n, e_w)$ then for all $v < w$ the name n does not appear in $\langle p_v, s_v, t_v \rangle$.

Proof.

i) Obvious by the token game on the SPL net (Definition 4.3.2).

ii) Suppose the contrary. Given a name $n \in \mathbf{N}$, there exists run stages w, v with $w \neq v$ such that $Fresh(n, e_w)$ and $Fresh(n, e_v)$. The name n is such that $n \in s_w$ and $n \notin s_{v-1}$. On the other hand if $w < v$ then $s_w \subseteq s_{v-1}$ (the token game can only increase the set of name conditions), a contradiction. Similarly for $v < w$.

iii) Suppose that $Fresh(n, e_w)$ and that there exists $v < w$ such that n appears in $\langle p_v, s_v, t_v \rangle$. This configuration is proper (Convention 5.1.1 and Proposition 4.2.1), therefore $n \in s_v$. From point $i)$, either $n \in s_0$ and in that case e_w can't occur since it would cause contact and not have concession (see token game) or there exists a previous stage $u < v$ such that $Fresh(n, e_u)$. Since $u \neq w$ it contradicts point $ii)$.

□

5.1.3 Precedence

The events that occur in a run respect the causal dependencies of the SPL net. From the token game it follows that an event can occur only in a configuration that contains all the preconditions of the event. The token game describes how configurations are formed starting from an initial configuration so that at a certain stage in the run a condition is marked either because it belonged to the initial configuration or because an earlier event carried it as postcondition. We distinguish two kinds of precedence. The first one, control precedence, is due to casual dependency given by control conditions and the second one, output-input precedence, is due to causal dependency given by output conditions.

Proposition 5.1.4 *(Control precedence) Within a run*

$$\langle p_0, s_0, t_0\rangle \xrightarrow{e_1} \cdots \xrightarrow{e_w} \langle p_w, s_w, t_w\rangle \xrightarrow{e_{w+1}} \cdots ,$$

if $b \in {}^c e_w$ at stage w then either $b \in Ic(p_0)$ or there is an earlier stage v, $v < w$, such that $b \in {e_v}^c$.

Proof. Clear by the token game (Definition 4.3.2). □

Proposition 5.1.5 *(Output-input precedence) In a run*

$$\langle p_0, s_0, t_0\rangle \xrightarrow{e_1} \cdots \xrightarrow{e_w} \langle p_w, s_w, t_w\rangle \xrightarrow{e_{w+1}} \cdots ,$$

if $M \in {}^o e_w$ at stage w then either $M \in t_0$ or there is an earlier stage v, $v < w$, such that $M \in {e_v}^o$.

Proof. Clear by the token game (Definition 4.3.2). □

We won't make much use of output-input precedence in this thesis. It remains a useful principle in some situations, however. In our examples we make use of the token game and a property on message surroundings instead.

5.2 On the power of the spy

A common step in the proof of a security property consists in determining the first event in a run of the protocol that violates a certain property of a message appearing on the network. Let, for example, m be a nonce and consider the property

$$\forall M \in t_u \,.\, m \sqsubseteq M \;\Rightarrow\; \{n, m, B\}_{Pub(A)} \sqsubseteq M \,. \tag{5.1}$$

Suppose the property holds at all stages u such that $u < w$ but does not hold at stage w in a run. The event e_w is therefore an output event such that

$$m \sqsubseteq e_w^o \quad \text{but} \quad \{n, m, B\}_{Pub(A)} \not\sqsubseteq e_w^o \,.$$

The question is: *Which event, among those of the protocol and the spy, could be the event e_w?*

Properties like the one above often are expressed by means of a submessage relation. Their proof usually requires a case analysis on the events of the protocol to determine which events violate them. It is sometimes possible to formulate a stronger property for which it is easier to find the events that violate the property. For example:

$$\text{At stage } u \text{ nonce } m \text{ can appear on the network only inside } \{n, m, B\}_{Pub(A)}, \tag{5.2}$$

is a stronger property than (5.1). If the network contains the message

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

then (5.1) holds but not (5.2).

5.2.1 Message surroundings

First we introduce the notion of surroundings of a message within a larger message to conveniently express properties like Property 5.2. Then, we characterise a class of properties of the surroundings of messages in a protocol run, that can never be violated by spy events.

Given messages M and N the surroundings of N in M are the smallest submessages of M containing N under one level of encryption. So for example the surroundings of $Key(A)$ in

$$(A, \{B, Key(A)\}_k, \{Key(A)\}_{k'})$$

are $\{B, Key(A)\}_k$ and $\{Key(A)\}_{k'}$. If N is a submessage of M but does not appear under an encryption in M then we take the surroundings of N in M to be N itself. For example the surroundings of $Key(A)$ in

$$(A, \{B, Key(A)\}_k, Key(A))$$

are $\{B, Key(A)\}_k$ and $Key(A)$. More precisely:

Definition 5.2.1 *(Surroundings of a message).* Let M and N be two messages. Define $\sigma(N, M)$ the surroundings of N in M inductively as follows:

$$\sigma(N, v) = \begin{cases} \{\, v \,\} & \text{if } N = v \\ \emptyset & \text{otherwise} \end{cases}$$

$$\sigma(N, k) = \begin{cases} \{\, k \,\} & \text{if } N = k \\ \emptyset & \text{otherwise} \end{cases}$$

$$\sigma(N, (M, M')) = \begin{cases} \{\, (M, M') \,\} & \text{if } N = M, M' \\ \sigma(N, M) \cup \sigma(N, M') & \text{otherwise} \end{cases}$$

$$\sigma(N, \{M\}_k) = \begin{cases} \{\, \{M\}_k \,\} & \text{if } N \in \sigma(N, M) \quad \text{or} \quad N = \{M\}_k \\ \sigma(N, M) & \text{otherwise} \end{cases}$$

$$\sigma(N, \psi) = \begin{cases} \{\, \psi \,\} & \text{if } N = \psi \\ \emptyset & \text{otherwise} \end{cases}$$

□

Clearly the surroundings of message M inside message N are sets of messages of the following form:

Proposition 5.2.2 *Let M, N be messages. There exist messages $N_1, \dots N_l$ and keys $k_1, \dots k_l$ such that*

$$\sigma(M, N) \subseteq \{M, \{N_1\}_{k_1}, \dots, \{N_l\}_{k_l}\} \quad .$$

□

As a shorthand write $\sigma(N, t) = \bigcup_{M \in t} \sigma(N, M)$. Using the surroundings of a message one can, for example, rewrite Property 5.2 as follows:

$$\sigma(m, t_u) \subseteq \{\{n, m, B\}_{Pub(A)}\} \quad .$$

If this property holds at some stage w in the run but fails at the next stage $w + 1$ then $w + 1$ is the first stage in the run where it fails, thus e_{w+1} is the first event in the run that violates it.

Proposition 5.2.3 *Within a run*

$$\langle p_0, s_0, t_0 \rangle \xrightarrow{e_1} \cdots \xrightarrow{e_w} \langle p_w, s_w, t_w \rangle \xrightarrow{e_{w+1}} \cdots ,$$

at every stage $\sigma(N, t_w) \subseteq \sigma(N, t_{w+1})$.

Proof. Message conditions are persistent, therefore from the token game it follows that at every stage w in a run $t_w \subseteq t_{w+1}$. □

5.2.2 A limitation on the power of the spy

Consider a spy with all the capabilities that we described in Section 4.1.10:

$$p_{spy} \equiv \,!\, \|_{i \in \{1, \dots 9\}} Spy_i$$

and if k is a key expression then define $\overline{k}$ as follows:

$$\overline{Pub(v)} = Priv(v), \quad \overline{Priv(v)} = Pub(v), \quad \overline{Key(\vec{v})} = Key(\vec{v}) \ .$$

Suppose that at a certain stage in the run of a protocol the surroundings of a secret message on the network are a number of cyphertexts. Suppose that the spy does not have access to any of the decryption keys and so cannot decipher any of the surroundings of the secret. In that situation (configuration) it is not too surprising that none of the events of the spy can change the surroundings of the secret.

Theorem 5.2.4 *Let N be a closed name or key expression. Consider a run*

$$\langle p_0 \parallel p_{spy}, s_0, t_0 \rangle \xrightarrow{e_1} \cdots \xrightarrow{e_w} \langle p_w, s_w, t_w \rangle \xrightarrow{e_{w+1}} \cdots \quad .$$

If at stage w for closed messages $M_1, \dots, M_l$ and closed symmetric and public keys $k_1, \dots, k_l$

$$\emptyset \subset \sigma(N, t_w) \subseteq \{\{M_1\}_{k_1}, \dots, \{M_l\}_{k_l}\} \text{ and } \sigma(N, t_{w+1}) \not\subseteq \{\{M_1\}_{k_1}, \dots, \{M_l\}_{k_l}\}$$

and at every stage v in the run $\overline{k_1} \notin t_v, \dots, \overline{k_l} \notin t_v$ then

$$e_{w+1} \notin spy : Ev(p_{spy}) \ ,$$

meaning that the event e_{w+1} is not a spy event.

Proof. Suppose that $e_{w+1} \in spy : Ev(p_{spy})$. The event e_{w+1} is the first output event in the run such that

$$\sigma(N, e^o_{w+1}) \not\subseteq \{\{M_1\}_{k_1}, \ldots, \{M_l\}_{k_l}\} \quad .$$

We consider all the possible shapes for the spy output events and prove that e_{w+1} can't have the shape of any spy event. Let i be a session index, M, M' messages and k a key. Distinguish the following cases:

Composition:

$$e_{w+1} = spy : i : 1 : \mathbf{Out}(out\ \psi_1, \psi_2; M, M') \ .$$

By control precedence, and the token game $M \in t_w$ and $M' \in t_w$ and therefore (since N is a name or a key and not a message tuple)

$$\begin{aligned} \sigma(N, t_{w+1}) &= \sigma(N, t_w) \cup \sigma(N, M) \cup \sigma(N, M') \\ &= \sigma(N, t_w) \\ &\subseteq \{\{M_1\}_{k_1}, \ldots, \{M_l\}_{k_l}\} \ . \end{aligned}$$

Decomposition:

$$e_{w+1} = spy : i : 2 : \mathbf{Out}(out\ \psi \,.\, out\ \psi'; M) \quad \text{or} \quad e_{w+1} = spy : i : 2 : \mathbf{Out}(out\ \psi; M) \ .$$

Similar to the previous case.

Encryption and *signature*:

$$e_{w+1} = spy : i : u : \mathbf{Out}(out\ \{\psi\}_k; M)$$

for $u = 3$ or $u = 4$ (encryption) or $u = 7$ (signature). By control precedence, and the token game $M \in t_w$ and therefore

$$\sigma(N, M) \subseteq \sigma(N, t_w) \subseteq \{\{M_1\}_{k_1}, \ldots, \{M_l\}_{k_l}\} \ .$$

From the definition of surroundings it follows that $\sigma(N, \{M\}_k) = \sigma(N, M)$, thus

$$\sigma(N, t_{w+1}) = \sigma(N, t_w) \subseteq \{\{M_1\}_{k_1}, \ldots, \{M_l\}_{k_l}\}$$

(since N is a name or a key and not an encryption or signature).
Decryption:

$$e_{w+1} = spy : i : u : \mathbf{Out}(out\ \psi; M)$$

for $u = 5$, $u = 6$ or $u = 8$. By control precedence, and the token game $\{M\}_k \in t_w$ and $\overline{k} \in t_w$. Therefore

$$\sigma(N, \{M\}_k) \subseteq \sigma(N, t_w) \subseteq \{\{M_1\}_{k_1}, \ldots, \{M_l\}_{k_l}\} \ .$$

By definition of surroundings, either

$$\sigma(N, \{M\}_k) = \{\{M\}_k\}$$

and from the assumptions of the theorem it follows that $\overline{k} \notin t_w$ or

$$\sigma(N, \{M\}_k) = \sigma(N, M) \not\subseteq \{\{M_1\}_{k_1}, \ldots, \{M_l\}_{k_l}\} \ .$$

In both cases one reaches a contradiction.
Signature verification:

$$e_{w+1} = spy : i : 8 : \mathbf{Out}(out\ \psi; M)\ .$$

By control precedence, and the token game $\{M\}_k \in t_w$ with k some private key and therefore

$$\sigma(N, \{M\}_k)\ \subseteq\ \sigma(N, t_w)\ \subseteq\ \{\{M_1\}_{k_1}, \ldots, \{M_l\}_{k_l}\}\ .$$

This, however, is not possible since by assumption k can't be a private key.
New name generation:

$$e_{w+1} = spy : i : p : \mathbf{Out}(out\, new(x)\, \psi; n, M)\ .$$

In this case $N = n$, however, by freshness $\sigma(N, t_w) = \emptyset$ which contradicts the assumption that $\sigma(N, t_w) \neq \emptyset$. □

The spy has limitations also regarding the possibility to forge digital signatures. If the spy does not have access to the private key of some agent then it can't produce a message signed with that key. Therefore if at some stage in a run the surroundings of a message reveal a signature that was not present at the previous stage then that signature was not produced by a spy event. A similar case is that of encrypting a message with a symmetric key that is not known to the spy – recall that in our idealised setting we assumed symmetric-key encryptions to be non-malleable.

Theorem 5.2.5 *Let N be a closed name or key expression. Consider a run*

$$\langle p_0 \parallel p_{spy}, s_0, t_0\rangle \xrightarrow{e_1} \cdots \xrightarrow{e_w} \langle p_w, s_w, t_w\rangle \xrightarrow{e_{w+1}} \cdots \quad .$$

If at stage w for a closed message M and a closed private or symmetric key k

$$\{M\}_k \in \sigma(N, t_{w+1}) \setminus \sigma(N, t_w)$$

and at every stage v in the run $k \notin t_v$ then $e_{w+1} \notin spy : Ev(p_{spy})$.

Proof. Suppose that $e_{w+1} \in spy : Ev(p_{spy})$. The event e_{w+1} is the first output event in the run such that

$$\{M\}_{Priv(n)} \in \sigma(N, e^o_{w+1})\ .$$

We consider all the possible shapes for the spy output events and prove that e_{w+1} can't have the shape of any spy event. Let i be a session index, M' message, and k key. Similarly to what we showed in the previous proof for the cases of composition and decomposition of messages, $\sigma(N, t_{w+1}) = \sigma(N, t_w)$. Moreover the case of new name generation is clearly excluded. Consider the remaining cases:

Encryption and *signature*:

$$e_{w+1} = spy : i : u : \mathbf{Out}(out\ \{\psi\}_{k'}; M')$$

for $u = 3$ or $u = 4$ (encryption) or $u = 7$ (signature). By control precedence and the token game $M' \in t_w$ and $k' \in t_w$. Obviously

$$\sigma(N, t_{w+1}) = \sigma(N, t_w) \cup \sigma(N, \{M'\}_{k'})$$

where $\{M\}_k \in \sigma(N, \{M'\}_{k'})$. From the definition of surroundings it follows that either

$$M' = M \quad \text{and} \quad k' = k$$

therefore $k \in t_w$ or

$$\{M\}_k \in \sigma(N, M')$$

and therefore $\{M\}_k \in \sigma(N, t_w)$.

Decrypting and Signature verification:

$$e_{w+1} = spy : i : 5 : \mathbf{Out}(out\ \psi; M')$$

for $u = 5$ or $u = 6$. Obviously

$$\sigma(N, t_{w+1}) \;=\; \sigma(N, t_w) \cup \sigma(N, M')$$

where $\{M\}_k \in \sigma(N, M')$. From the definition of surroundings (and the assumption that N is a name or a key expression) it follows that

$$\{M\}_k \in \sigma(N, \{M'\}_{k'})$$

for any key k'. By control precedence, and the token game $\{M'\}_{k'} \in t_w$ for some key k', thus $\{M\}_k \in \sigma(N, t_w)$.

□

The two previous results can be generalised, with some care, to arbitrary closed messages. For simplicity we restricted to messages N that are closed name expressions or key expressions, which is enough for the examples that are studied in this thesis.

Our results, based on the notion of surroundings are similar the results based on the notion of honest ideals [88] in the strand-space model, as well as Schneider's rank functions [77] in the CSP model of a security protocol. Honest ideals and rank functions are known to be closely related [38]. An ideal in the strand-space model expresses the set of all messages containing a particular submessage when encryption is restricted to a certain set of keys. A honest ideal for a bundle is an ideal such that no penetrator node in the bundle contributes to the ideal, except for a lucky guess of a nonce or a key. The main theorem in [88] gives general conditions under which an ideal is honest. Our approach and in particular Theorem 5.2.4 has a dual flavor. Instead of giving conditions under which a penetrator can not contribute to an ideal, we give conditions under which a penetrator can not extend the surroundings of a message.

5.3 Diagrammatic style of reasoning

We often describe precedence among events in a run in a diagrammatic way. If we know that in a run an event e has to precede the event e' then we draw

$$e \longrightarrow e' \quad .$$

The precedence relation among e and e' is due to control precedence, output-input precedence, freshness or any other precedence among events that can be established.

We sometimes write a message on a link between two events – if $e^o = \{M\}$ and ${}^o e' = \{M\}$ where M is a message and if e precedes e' in the run under consideration then we sometimes draw

$$\begin{tikzcd} e \arrow[r, "M"] & e' \end{tikzcd} \quad .$$

Note that this does not necessarily mean that the message sent by e is directly received by e'. There could be other events that consume and mark that message in between, for example events from a spy. Sometimes a message is sent on the network and we don't care about the event receiving it and sometimes a message is received and we don't care what event produced it. We describe this situation drawing a $\bullet$. For example

$$\begin{tikzcd} \bullet \arrow[r, "M"] & e \end{tikzcd}$$

stands for a run in which the event e received the message M without saying anything about the origin of message M in that run. We can build more complex diagrams to show the different dependencies among events in a run. For example

$$\begin{tikzcd} e \arrow[r] \arrow[d] & e' \\ e'' \arrow[r, "M"] & \bullet \end{tikzcd} \quad .$$

In the run under consideration the event e precedes both the events e and e''. The diagram however does not say whether e' precedes or follows e'' in the run. The diagrams that we draw are a graphic representation of some of the causal dependencies among events in a run. In proofs we may add events and links between events as we establish them. This kind of diagrams has already been introduced to describe bundles of a strand space [89]. They are also a convenient way of displaying the events in a run leading to an attack.

5.4 Security properties of NSL

We use the net semantics to formalise security properties for NSL. Typically security properties are properties of all runs of the protocol. If t_0 is the network state given by the set of messages on the network from which a protocol execution starts then for the NSL protocol we are interested in proper runs of the kind

$$\langle NSL, s_0, t_0 \rangle \xrightarrow{e_1} \cdots \xrightarrow{e_w} \langle p_w, s_w, t_w \rangle \xrightarrow{e_{w+1}} \cdots \quad .$$

In the remaining part of this section, a run of this kind is meant whenever we refer to a run of NSL.

Each security theorem comes with precise conditions under which it holds. A common pattern can be recognised throughout proofs of security theorems:

> If a property of configurations holds at some stage in the run and does not hold in a later stage then there is a first event e in between the two stages that makes the property fail.

A crucial step in our proof strategy is to determine which event, among those in the protocol net, is the event e. The proof principles of Section 5.1, the token game for nets with persistent conditions and Theorem 5.2.4 are used in determining what event e is. Our proof strategy is closely related to that of strand-space arguments (see for example [90, 88]).

5.4.1 Secrecy

A value is secret or confidential withing a group of agents if it is not disclosed to others outside the group. The leakage of a secret can happen for example if an agent belonging to the group with access to the secret value spontaneously decides to publish it or to hand it over to strangers, with perhaps a malicious intention. We are not interested in considering this possibility in our analysis, which is hard to prevent. We focus our attention to the security of the protocol instead, and assume that principals follow the protocol. The malicious environment is represented by a spy, a stranger to which secrets might be leaked. For example when the spy gets hold of the right decryption key of an agent that sends an encrypted value over the network, the spy can recover that value and publish it in cleartext on the network. One can therefore model confidentiality of a value by asking that the value never appears in cleartext on the network.

In the NSL protocol both initiator and responder choose and exchange fresh nonces. Once exchanged, the two nonces could be used to establish a common session key for further secure communication. It is therefore desirable that the protocol does not leak any of the two nonces but keeps them secret instead.

The first secrecy theorem for NSL regards the private keys of principals. If private keys are not corrupted from the start and principals behave according to the protocol then the keys are not leaked during a protocol run. If one assumes $Priv(A_0) \not\sqsubseteq t_0$, where $A_0 \in \mathcal{A}$, then the initial network state clearly does not expose the private key of A_0 to any danger of corruption – that key is not appearing anywhere as part of messages in t_0. We will make use of the following theorem in establishing more security properties for NSL.

Theorem 5.4.1 *Given a run of NSL and $A_0 \in \mathcal{A}$, if $Priv(A_0) \not\sqsubseteq t_0$ then at each stage w in the run $Priv(A_0) \not\sqsubseteq t_w$.*

Proof. Suppose there is a run of NSL in which $Priv(A_0)$ appears on a message sent over the network. This means, since $Priv(A_0) \not\sqsubseteq t_0$, that there is a stage $w > 0$ in the run such that

$$Priv(A_0) \not\sqsubseteq t_{w-1} \quad \text{and} \quad Priv(A_0) \sqsubseteq t_w \ .$$

The event e_w is an event in the set

$$Ev(NSL) = init : Ev(p_{init}) \ \cup \ resp : Ev(p_{resp}) \ \cup \ spy : Ev(p_{spy})$$

and, by the token game of nets with persistent conditions, is such that

$$Priv(A_0) \sqsubseteq e_w^o \ .$$

As can easily be checked, the shape of every initiator or responder event

$$e \in init : Ev(p_{init}) \cup resp : Ev(p_{resp})$$

is such that

$$Priv(A_0) \not\sqsubseteq e^o .$$

The event e_w can therefore only be a spy event. If $e_w \in spy : Ev(p_{spy})$, however, by control precedence and the token game one would find an earlier stage u in the run, $u < w$, such that $Priv(A_0) \sqsubseteq t_u$ and therefore reach a contradiction. We show that e_w can't be the spy output event of a decryption, the other cases are similar. Suppose that e_w carries action

$$act(e_w) = spy : i : 5 : out\, M$$

where i is a round index and M is a message such that $Priv(A_0) \sqsubseteq M$. By control precedence there exists an event e_{u+1} such that $0 \leq u < w$ and

$$act(e_{u+1}) = spy : i : 5 : in\, \{M\}_{Pub(B_0)}$$

for some agent name $B_0 \in \mathcal{A}$. Clearly $Priv(A_0) \sqsubseteq \{M\}_{Pub(B_0)}$. From the token game it follows that $\{M\}_{Pub(B_0)} \in t_u$ and therefore $Priv(A_0) \sqsubseteq t_u$. □

The second secrecy property that we prove for NSL, is the secrecy of a nonce generated by the initiator. NSL-protocol exchanges keep initiator nonces secret provided private keys are not corrupted initially. We are interested in proving the secrecy of a new nonce that has been chosen on an initiator event, and thus in runs which contain such initiator events.

Theorem 5.4.2 *Given a run of NSL and $A_0, B_0 \in \mathcal{A}$, if $Priv(A_0), Priv(B_0) \not\sqsubseteq t_0$ and the run contains an initiator event a_1 labelled with action*

$$act(a_1) \quad = \quad init : (A_0, B_0) : i_0 : out\, new\, m_0\ \{m_0, A_0\}_{Pub(B_0)} \ ,$$

where i_0 is a session index and m_0 a name, then at every stage w in the run $m_0 \notin t_w$.

Proof. We show a stronger property. Consider the property of configurations $\langle p, s, t\rangle$ and names n

$$Q(p, s, t, n) \ \Leftrightarrow \ \sigma(m_0, t) \subseteq \{\{m_0, A_0\}_{Pub(B_0)}, \{m_0, n, B_0\}_{Pub(A_0)}\} \ .$$

If we can show that

there exists $n \in \mathbf{N}$ such that at every stage w in the run $Q(p_w, s_w, t_w, n)$

then clearly $m_0 \notin t_w$ for every stage w in the run. Suppose the contrary. Let n_0 be a name such that $n_0 \neq m_0$ and suppose that at some stage in the run the property Q does not hold. By freshness clearly $Q(NSL, s_0, t_0, n_0)$. Let v, by Well-foundedness, be the first stage in the run such that $\neg Q(p_v, s_v, t_v, n_0)$. From the Freshness Principle 5.1.3 it follows that

$$\begin{tikzcd} a_1 \arrow[r] & e_v \end{tikzcd}$$

and from the token game $\{m_0, A_0\}_{Pub(B_0)} \in \sigma(m_0, t_{v-1})$ (messages on the network are persistent). The event e_v is an event in

$$Ev(NSL) = init : Ev(p_{init}) \ \cup \ resp : Ev(p_{resp}) \ \cup \ spy : Ev(p_{spy})$$

and, from the token game of nets with persistent conditions, is such that

$$\sigma(m_0, e_v^n) \not\subseteq \{\{m_0, A_0\}_{Pub(B_0)}, \{m_0, n_0, B_0\}_{Pub(A_0)}\} \ . \tag{5.3}$$

Clearly e_v can only be an output event since $e^n = \emptyset$ for all input events e. We examine the possible output events of $Ev(NSL)$ and conclude that $e_v \notin Ev(NSL)$, reaching a contradiction.

Initiator output events. We distinguish two cases:
Case 1:

$$act(e_v) = init : (A, B) : j : out\, new\, m\ \{m, A\}_{Pub(B)}$$

where $A, B \in \mathcal{A}$, and so $A, B \in s_0$ and where m is a name and j is an index. Property (5.3) and the definition of message surroundings imply that $m_0 \sqsubseteq \{m, A\}_{Pub(B)}$. If $m = m_0$ then one reaches a contradiction to Property (5.3) because from the Freshness Principle 5.1.3 it follows that $e_v^n = \{\{m_0, A_0\}_{Pub(B_0)}\}$. Since $A \in s_0$ freshness also implies that $A \neq m_0$. Therefore e_v can't be an initiator event with the above action.
Case 2:

$$act(e_v) = init : (A, B) : j : out\ \{n\}_{Pub(B)}$$

where $A, B \in \mathcal{A}$, and so $A, B \in s_0$, and where n is a name and j is an index. As before, from the definition of message surroundings and Property (5.3) it follows that $n = m_0$. By Control-precedence 5.1.4, there exists an event e_u in the run such that

$$\begin{tikzcd} e_u \arrow[r] & e_v \end{tikzcd}$$

and

$$act(e_u) = init : (A, B) : j : in\ \{m, m_0, B\}_{Pub(A)}$$

for some name m. By the token game

$$\{m, m_0, B\}_{Pub(A)} \in t_{u-1}$$

where $m_0 \neq n_0$ and so $\neg Q(p_{u-1}, s_{u-1}, t_{u-1}, n_0)$ which is a contradiction since $u < v$.
Responder output events. There is only one possible case:

$$act(e_v) = resp : B : j : out\, new\, n\ \{m, n, B\}_{Pub(A)}$$

where $A, B \in \mathcal{A}$, and so $A, B \in s_0$, m, n are names and j an index. From the freshness principle it follows that $n \neq m_0$ and $B \neq m_0$. Therefore, since Property (5.3) holds and by the definition of message surroundings, $m = m_0$ and either $B \neq B_0$ or $A \neq A_0$. By Control-precedence 5.1.4, there exists an event e_u in the run such that

$$\begin{tikzcd} e_u \arrow[r] & e_v \end{tikzcd}$$

and

$$act(e_u) = resp : B : j : in\ \{m_0, A\}_{Pub(B)} \ .$$

By the token game

$$\{m_0, A\}_{Pub(B)} \in t_{u-1}$$

and $\neg Q(p_{u-1}, s_{u-1}, t_{u-1}, n_0)$ since either $B \neq B_0$ or $A \neq A_0$. A contradiction follows because $u < v$.

Spy output events. An assumption of the theorem is that the private keys of the two agents A_0 and B_0 are not leaked, meaning that $Priv(A_0), Priv(B_0) \not\sqsubseteq t_0$. At every stage w in the run $Priv(A_0), Priv(B_0) \notin t_w$ (Theorem 5.4.1). Since $\sigma(m_0, t_{v-1}) \neq \emptyset$, $Q(p_{v-1}, s_{v-1}, t_{v-1}, n_0)$, and $\neg Q(p_v, s_v, t_v, n_0)$, Theorem 5.2.4 can be applied and one concludes that e_v is not a spy event. $\square$

It is easy to see how a spy, as the one shown in Section 4.1.10, can get hold of the secret nonce if either $Priv(A_0)$ or $Priv(B_0)$ are corrupted. For example suppose a run of NSL containing the events

$$\begin{aligned} act(a_1) &= init : (A_0, B_0) : j_0 : out\, new\, m_0\ \{m_0, A_0\}_{Pub(B_0)} \\ act(b_2) &= resp : B_0 : i_0 : out\, new\, n_0\ \{m_0, n_0, B_0\}_{Pub(A_0)} \end{aligned}$$

and suppose $Priv(A_0) \in t_0$. One can easily extend the run to an NSL run such that at some stage v in the run $m_0 \in t_v$. For example

$$\begin{tikzcd}
a_1 \arrow[r, "\{m_0,A_0\}_{Pub(B_0)}"] & b_1 \arrow[d] & s_1 \arrow[d] & & \\
& b_2 \arrow[r, "\{m_0,n_0,B_0\}_{Pub(A_0)}"] & s_2 \arrow[d] & & \\
& & s_3 \arrow[r, "m_0{,}n_0{,}B_0"] & s_4 \arrow[d] & \\
& & & s5 \arrow[r, "m_0"] & \bullet
\end{tikzcd}$$

where b_1 and b_2 are the obvious responder events and $s_1, \ldots, s_5$ are spy event with actions

$$\begin{aligned} act(s_1) &= spy : 5 : l_0 : in\ Priv(A_0) \\ act(s_2) &= spy : 5 : l_0 : in\ \{m_0, n_0, B_0\}_{Pub(A_0)} \\ act(s_3) &= spy : 5 : l_0 : out\ m_0, n_0, B_0 \\ act(s_4) &= spy : 2 : r_0 : in\ m_0, n_0, B_0 \\ act(s_5) &= spy : 2 : r_0 : out\ m_0 \qquad . \end{aligned}$$

The third secrecy property that we prove for NSL, is the secrecy of a nonce generated by the responder. As for initiator nonces, NSL-protocol exchanges keep responder nonces secret provided private keys are not corrupted initially. We are interested in those NSL runs that contain an NSL-responder event on which a new nonce is chosen and then sent out.

Theorem 5.4.3 *Given a run of NSL and $A_0, B_0 \in \mathcal{A}$, if $Priv(A_0), Priv(B_0) \not\sqsubseteq t_0$ and if the run contains a responder event b_2 labelled with action*

$$act(b_2) = resp : B_0 : i_0 : out\, new\, n_0\ \{m_0, n_0, B_0\}_{Pub(A_0)}$$

for some round index i_0 then at every stage w in the run $n_0 \notin t_w$.

Proof. We show a stronger property. Consider the following property on configurations $\langle p, s, t\rangle$

$$Q(p,s,t) \;\Leftrightarrow\; \sigma(n_0,t) \subseteq \{\{n_0\}_{Pub(B_0)}, \{m_0,n_0,B_0\}_{Pub(A_0)}\}\ .$$

From the freshness principle it follows that $Q(NSL, s_0, t_0)$. Suppose that at some stage in the run the property does not hold. Let v, by Well-foundedness, be the first stage in the run such that $\neg Q(p_v, s_v, t_v)$. From the Freshness Principle 5.1.3 it follows that

$$\begin{tikzcd} b_2 \arrow[r] & e_v \end{tikzcd}$$

and from the token game $\{m_0,n_0,B_0\}_{Pub(A_0)} \in \sigma(n_0, t_{v-1})$ (messages on the network are persistent). The event e_v is an event in

$$Ev(NSL) = init : Ev(p_{init}) \;\cup\; resp : Ev(p_{resp}) \;\cup\; spy : Ev(p_{spy})\ .$$

From the token game of nets with persistent conditions the event e_v is such that

$$\sigma(n_0, e_v^n) \not\subseteq \{\{n_0\}_{Pub(B_0)}, \{m_0,n_0,B_0\}_{Pub(A_0)}\}\ . \tag{5.4}$$

Clearly e_v can only be an output event since $e^n = \emptyset$ for all input events e. We examine the possible output events of $Ev(NSL)$ and conclude that $e_v \notin Ev(NSL)$, reaching a contradiction.
Initiator output events. We distinguish two cases:
Case 1:

$$act(e_v) = init : (A,B) : j : out\,new\,m\ \{m,A\}_{Pub(B)}$$

where $A, B \in \mathcal{A}$, and so $A, B \in s_0$, m a name and j an index. Because of Property (5.4) and the definition of surroundings, $n_0 \sqsubseteq \{m,A\}_{Pub(B)}$. From the freshness principle it follows that $m \neq n_0$ and since $A \in s_0$ it also follows that $A \neq n_0$. Therefore e_v can't be an initiator event of carrying the above action.
Case 2:

$$act(e_v) = init : (A,B) : j : out\ \{n\}_{Pub(B)}$$

where $A, B \in \mathcal{A}$, and so $A, B \in s_0$, and where n is a name and j is an index. As before, by the definition of message surroundings and the Property (5.4), $n = n_0$ and $B \neq B_0$. By Control-precedence 5.1.4, there exists an event e_u in the run such that

$$\begin{tikzcd} e_u \arrow[r] & e_v \end{tikzcd}$$

and

$$act(e_u) = init : (A,B) : j : in\ \{m,n_0,B\}_{Pub(A)}$$

for some name m. From the token game it follows that

$$\{m,n_0,B\}_{Pub(A)} \in t_{u-1}\ .$$

Therefore $\neg Q(p_{u-1}, s_{u-1}, t_{u-1})$, since $B \neq B_0$. A contradiction since $u < v$.
Responder output events. There is only one possible case:

$$act(e_v) = resp : B : j : out\,new\,n\ \{m,n,B\}_{Pub(A)}$$

where $A, B \in \mathcal{A}$, and so $A, B \in s_0$, and where m, n are names and j is an index. From the freshness principle it follows that $n \neq n_0$ and $B \neq n_0$. Property (5.4) holds and by the definition of surroundings, $m = n_0$. By Control-precedence 5.1.4, there exists an event e_u in the run such that

$$\begin{tikzcd} e_u \arrow[r] & e_v \end{tikzcd}$$

and

$$act(e_u) = resp : B : j : in\ \{n_0, A\}_{Pub(B)}\ .$$

By the token game,

$$\{n_0, A\}_{Pub(B)} \in t_{u-1}$$

and $\neg Q(p_{u-1}, s_{u-1} t_{u-1})$. A contradiction follows because $u < v$.
Spy output events. As an assumption the private keys of the two agents A_0 and B_0 are not leaked, meaning that $Priv(A_0), Priv(B_0) \not\sqsubseteq t_0$. Therefore at every stage w in the run $Priv(A_0), Priv(B_0) \notin t_w$ (Theorem 5.4.1). Since $\sigma(n_0, t_{v-1}) \neq \emptyset$, $Q(p_{v-1}, s_{v-1}, t_{v-1})$ and $\neg Q(p_v, s_v, t_v)$, Theorem 5.2.4 can be applied and one concludes that e_v is not a spy event. □

5.4.2 Authentication

An NSL protocol exchange between an initiator and a responder achieves mutual authentication. The authentication properties that hold for NSL are of a strong kind. They are agreement properties (see Lowe [48]) saying that, for example, to a protocol session completed by the initiator there corresponds a protocol session of the responder which agrees on the exchanged nonces. The initiator can then be sure it interacted with the right agent. This kind of properties however do not exclude the possibility of a spy as a middleman which simply forwards messages. For the NSL protocol this can indeed be the case, and it is arguable whether or not this is an attack. This stronger kind of authentication has been studied by Roscoe in [69].

The first authentication theorem is a guarantee for the responder. We are interested in those runs of NSL that contain responder events for a completed protocol session. It turns out that the following theorem needs the assumption that the private key of the initiator is not corrupted.

Theorem 5.4.4 *If a run of NSL contains responder events b_2, b_3 with actions*

$$\begin{aligned} act(b_2) &= resp : B_0 : i_0 : out\,new\,n_0\ \{m_0, n_0, B_0\}_{Pub(A_0)} \\ act(b_3) &= resp : B_0 : i_0 : in\ \{n_0\}_{Pub(B_0)} \end{aligned}$$

for some round index i_0 and $Priv(A_0) \not\sqsubseteq t_0$, then the run contains initiator events a_1, a_2, a_3 with actions

$$\begin{aligned} act(a_1) &= init : (A_0, B_0) : j_0 : out\,new\,m_0\ \{m_0, A_0\}_{Pub(B_0)} \\ act(a_2) &= init : (A_0, B_0) : j_0 : in\ \{m_0, n_0, B_0\}_{Pub(A_0)} \\ act(a_3) &= init : (A_0, B_0) : j_0 : out\ \{n_0\}_{Pub(B_0)} \end{aligned}$$

where j_0 is a round index and such that $\begin{tikzcd} a_3 \arrow[r] & b_3 \end{tikzcd}$.

Proof. By control precedence we obtain

$$\begin{tikzcd} b_2 \arrow[d] \\ b_3 \ . \end{tikzcd}$$

Consider the property of configurations $\langle p, s, t\rangle$

$$Q(p,s,t) \ \Leftrightarrow\ \sigma(n_0,t) \subseteq \{\{m_0,n_0,B_0\}_{Pub(A_0)}\}$$

By freshness the property Q holds immediately after b_2, but clearly not immediately before b_3. By well-foundedness there is an earliest stage v in the run such that $\neg Q(p_v,s_v,t_v)$. The event e_v is an output event such that

$$\sigma(n_0,e_v^o) \not\subseteq \{\{m_0,n_0,B_0\}_{Pub(A_0)}\} \tag{5.5}$$

and it follows b_2 but precedes b_3 in the run

$$\begin{tikzcd} b_2 \arrow[d] \arrow[dr] & \\ b_3 & e_v \arrow[l] \ . \end{tikzcd}$$

We inspect the output events of NSL to determine which action the event e_v carries.
Initiator output events. We distinguish two cases:
Case 1:

$$act(e_v) = init:(A,B):j_0:out\,new\,m\ \{m,A\}_{Pub(B)}$$

where $A,B \in \mathcal{A}$, and so $A,B \in s_0$, and where m is a name and j_0 is an index. Property 5.5 holds and by the definition of message surroundings, $n_0 \sqsubseteq \{m,A\}_{Pub(B)}$. The Freshness Principle 5.1.3 implies that $m \neq n_0$ and since $A \in s_0$ it also implies that $A \neq n_0$. Therefore e_v can't be an initiator event of carrying the above action.
Case 2:

$$act(e_v) = init:(A,B):j_0:out\ \{n\}_{Pub(B)}$$

where $A,B \in \mathcal{A}$, and so $A,B \in s_0$, and where n is a name and j_0 is an index. As before, by the definition of message surroundings and the Property (5.5), $n = n_0$. By Control-precedence 5.1.4, there exists an event e_u in the run such that

$$\begin{tikzcd} e_u \arrow[r] & e_v \end{tikzcd}$$

and

$$act(e_u) = init:(A,B):j_0:in\ \{m,n_0,B\}_{Pub(A)}$$

for some name m. This, however, is only possible if $B = B_0$, $A = A_0$, and $m = m_0$, since $u < v$. Therefore $Q(p_{u-1},s_{u-1},t_{u-1})$.
Responder output events. There is only one case possible:

$$act(e_v) = resp:B:j_0:out\,new\,n\ \{m,n,B\}_{Pub(A)}$$

where $A, B \in \mathcal{A}$, and so $A, B \in s_0$, and where m, n are names and j_0 is an index. From the freshness principle it follows that $n \neq n_0$ and $B \neq n_0$. Property (5.5) holds and by the definition of message surroundings, $m = n_0$. By Control-precedence 5.1.4, there exists an event e_u in the run such that

$$\begin{tikzcd} e_u \arrow[r] & e_v \end{tikzcd}$$

and

$$act(e_u) = resp : B : j_0 : in\ \{n_0, A\}_{Pub(B)}\ .$$

By the token game,

$$\{n_0, A\}_{Pub(B)} \in t_{u-1}$$

and $\neg Q(p_{u-1}, s_{u-1} t_{u-1})$. A contradiction follows, since $u < v$.

Spy output events. An assumption of the theorem is that the private key of agent A_0 is not leaked, meaning that $Priv(A_0) \not\sqsubseteq t_0$. Then that key is not corrupted by the protocol – at every stage w in the run $Priv(A_0) \notin t_w$ (Theorem 5.4.1). Observe that due to message persistence $\{m_0, n_0, B_0\}_{Pub(A_0)} \in \sigma(n_0, t_{v-1})$. Since $Q(p_{v-1}, s_{v-1}, t_{v-1})$ and $\neg Q(p_v, s_v, t_v)$, Theorem 5.2.4 can applies and one concludes that e_v is not a spy event.

From the above inspection of NSL events it follows that e_v is an initiator event with action

$$act(e_v) = init : (A_0, B_0) : j_0 : out\ \{n_0\}_{Pub(B_0)}$$

and that, in the run, it is preceded by another initiator event e_u with action

$$act(e_u) = init : (A_0, B_0) : j_0 : out\ \{m_0, n_0, B_0\}_{Pub(A_0)}\ .$$

Let a_3 be the event e_v and a_2 the event e_u, yielding the diagram:

$$\begin{tikzcd} b_2 \arrow[d] \arrow[dr] & a_2 \arrow[d] \\ b_3 & a_3 \arrow[l, "\{n_0\}_{Pub(A_0)}"] \ . \end{tikzcd}$$

Since $Fresh(b_2, n_0)$ the event b_2 precedes a_2:

$$\begin{tikzcd} b_2 \arrow[d] \arrow[r, "\{m_0,n_0,B_0\}_{Pub(B_0)}"] & a_2 \arrow[d] \\ b_3 & a_3 \arrow[l, "\{n_0\}_{Pub(A_0)}"] \ . \end{tikzcd}$$

By control precedence the run contains an event a_1 labelled with the desired action:

$$\begin{tikzcd} & a_1 \arrow[d] \\ b_2 \arrow[d] \arrow[r, "\{m_0,n_0,B_0\}_{Pub(B_0)}"] & a_2 \arrow[d] \\ b_3 & a_3 \arrow[l, "\{n_0\}_{Pub(A_0)}"] \ . \end{tikzcd}$$

By control precedence there is an event b_1 carrying action

$$act(b_1) = resp : B_0 : i_0 : in\ \{m_0, A_0\}_{Pub(B_0)}$$

and yielding the diagram:

$$\begin{tikzcd}
b_1 \arrow[d] & a_1 \arrow[d] \\
b_2 \arrow[r, "\{m_0,n_0,B_0\}_{Pub(A_0)}"] \arrow[d] & a_2 \arrow[d] \\
b_3 & a_3 \arrow[l, "\{n_0\}_{Pub(B_0)}"] \ .
\end{tikzcd}$$

Since $Fresh(a_1, m_0)$ the event a_1 precedes b_1 and one obtains the following diagram which completes the proof:

$$\begin{tikzcd}
b_1 \arrow[d] & a_1 \arrow[l, "\{m_0,A_0\}_{Pub(B_0)}"'] \arrow[d] \\
b_2 \arrow[r, "\{m_0,n_0,B_0\}_{Pub(A_0)}"] \arrow[d] & a_2 \arrow[d] \\
b_3 & a_3 \arrow[l, "\{n_0\}_{Pub(B_0)}"] \ .
\end{tikzcd}$$

□

We have seen the form of authentication that NSL guarantees to the responder, now we study a similar guarantee for the initiator. This time, both private keys of initiator and responder should not be corrupted. We show later how authentication fails when one of the two keys becomes corrupted at some stage a the protocol run.

Theorem 5.4.5 *If a run of NSL contains an initiator event a_2 with action*

$$act(a_2) \quad = \quad init : (A, B) : i_0 : in\ \{m_0, n_0, B_0\}_{Pub(A_0)}$$

for some round index i_0 and $Priv(A_0), Priv(B_0) \not\sqsubseteq t_0$, then the run contains responder events b_1, b_2 with actions

$$\begin{aligned}
act(b_1) &= resp : B_0 : j_0 : in\ \{m_0, A_0\}_{Pub(B_0)} \\
act(b_2) &= resp : B_0 : j_0 : out\, new\, n_0\ \{m_0, n_0, B_0\}_{Pub(A_0)} \quad .
\end{aligned}$$

where j_0 is a round index and such that $b_2 \longrightarrow a_2$.

Proof. By control precedence we obtain

$$\begin{tikzcd}
a_1 \arrow[d] \\
a_2
\end{tikzcd}$$

where

$$act(a_1) = init : (A, B) : i_0 : out\, new\, m_0\ \{m_0, A_0\}_{Pub(B_0)}\ .$$

Consider the property of configurations $\langle p, s, t\rangle$

$$Q(p, s, t)\ \Leftrightarrow\ \sigma(m_0, t) \subseteq \{\{m_0, A_0\}_{Pub(B_0)}\} \cup \{\{m_0, n, B_0\}_{Pub(A_0)} \mid n \in \mathbf{N} \backslash \{m_0, n_0\}\}\ .$$

By freshness the property Q holds immediately after a_1 but not immediately before a_2. By well-foundedness there is an earliest stage v in the run such that $\neg Q(p_v, s_v, t_v)$. The event e_v is an output event such that

$$\sigma(m_0, e_v^o) \not\subseteq \{\{m_0, A_0\}_{Pub(B_0)}\} \cup \{\{m_0, n, B_0\}_{Pub(A_0)} \mid n \in \mathbf{N} \setminus \{m_0, n_0\}\} \tag{5.6}$$

and it follows a_1 but precedes a_2 in the run:

$$\begin{tikzcd} a_1 \arrow[d] \arrow[dr] & \\ a_2 & e_v \arrow[l] \end{tikzcd}$$

We inspect the output events of NSL to determine which event e_v is.
Initiator output events. We distinguish two cases:
Case 1:

$$act(e_v) = init : (A, B) : j_0 : out\, new\, m\ \{m, A\}_{Pub(B)}$$

where $A, B \in \mathcal{A}$, and so $A, B \in s_0$, and where m is a name and j_0 is an index. Property 5.6 holds and by the definition of message surroundings, $m_0 \sqsubseteq \{m, A\}_{Pub(B)}$. Freshness 5.1.3 implies that $m \neq m_0$ and since $A \in s_0$ it also implies that $A \neq m_0$. Therefore e_v can't be an initiator event carrying the above action.
Case 2:

$$act(e_v) = init : (A, B) : j_0 : out\ \{n\}_{Pub(B)}$$

where $A, B \in \mathcal{A}$, and so $A, B \in s_0$, and where n is a name and j_0 is an index. As before, from the definition of surroundings and Property (5.6) it follows that $n = m_0$. By Control-precedence 5.1.4, there exists an event e_u in the run such that

$$\begin{tikzcd} e_u \arrow[r] & e_v \end{tikzcd}$$

and

$$act(e_u) = init : (A, B) : j_0 : in\ \{m, m_0, B\}_{Pub(A)}$$

for some name m. This, however, is not possible since $u < v$ and $Q(p_{u-1}, s_{u-1}, t_{u-1})$.
Responder output events. There is only one possible case:

$$act(e_v) = resp : B : j_0 : out\, new\, n\ \{m, n, B\}_{Pub(A)}$$

where $A, B \in \mathcal{A}$, and so $A, B \in s_0$, and where m, n are names and j_0 is an index. From the freshness principle it follows that $n \neq m_0$ and $B \neq m_0$. Therefore, since Property (5.6) holds and by the definition of message surroundings, $m = m_0$. By Control-precedence 5.1.4, there exists an event e_u in the run such that

$$\begin{tikzcd} e_u \arrow[r] & e_v \end{tikzcd}$$

and

$$act(e_u) = resp : B : j_0 : in\ \{m_0, A\}_{Pub(B)}\ .$$

By the token game,

$$\{m_0, A\}_{Pub(B)} \in t_{u-1}$$

and since $Q(p_{u-1}, s_{u-1}t_{u-1})$ it follows that $A = A_0$ and $B = B_0$. Moreover, $n = n_0$ since $\neg Q(p_v, s_v, t_v)$.

Spy output events. An assumption of the theorem is that the private keys of A_0 and B_0 are not leaked, meaning that $Priv(A_0), Priv(B_0) \not\sqsubseteq t_0$. At every stage w in the run $Priv(A_0), Priv(B_0) \notin t_w$ (Theorem 5.4.1). Observe that due to message persistence $\{m_0, A_0\}_{Pub(B_0)} \in \sigma(m_0, t_{v-1})$. Since $Q(p_{v-1}, s_{v-1}, t_{v-1})$ and $\neg Q(p_v, s_v, t_v)$, Theorem 5.2.4 can be applied and one can conclude that e_v can not be a spy event.

From the inspection of NSL events above, it follows that e_v is a responder event with action

$$act(e_v) = resp : B : j_0 : out\,new\,n_0\ \{m_0, n_0, B_0\}_{Pub(A_0)}$$

for some round index j_0. Let therefore $b_2 = e_v$ and by control precedence the run contains a responder event b_1 with action

$$act(b_1) = resp : B : j_0 : in\ \{m_0, A_0\}_{Pub(B_0)}$$

which precedes b_2:

$$\begin{tikzcd}
a_1 \arrow[d] & & b_1 \arrow[d] \\
a_2 & & b_2 \arrow[ll, "\{m_0,n_0,B_0\}_{Pub(A_0)}"'] \ .
\end{tikzcd}$$

By freshness:

$$\begin{tikzcd}
a_1 \arrow[d] \arrow[rr, "\{m_0,A_0\}_{Pub(B_0)}"] & & b_1 \arrow[d] \\
a_2 & & b_2 \arrow[ll, "\{m_0,n_0,B_0\}_{Pub(A_0)}"'] \ .
\end{tikzcd}$$

□

Sometimes it is convenient to use already established security results as lemmas for other security theorems. We have seen, for example, how the secrecy of the private keys is used in proving secrecy of the nonces. The authentication theorem for the initiator could be proved in a more concise way if one used the secrecy property established for initiator nonces. An alternative proof goes as follows:

Proof. (*An alternative proof for the initiator's authentication guarantee*)
The proof is along the same lines of the previous one. We choose, however, a simpler property of configurations and make use of the secrecy Theorem 5.4.2.

By control precedence we obtain

$$\begin{tikzcd}
a_1 \arrow[d] \\
a_2
\end{tikzcd}$$

where

$$act(a_1) = init : (A_0, B_0) : i_0 : out\, new\, m_0\ \{m_0, A_0\}_{Pub(B_0)}\ .$$

Consider the property of configurations $\langle p, s, t\rangle$

$$Q(p, s, t) \ \Leftrightarrow\ \sigma(m_0, t) \subseteq \{\{m_0, A_0\}_{Pub(B_0)}\}\ .$$

By freshness the property Q holds immediately after a_1 but not immediately before a_2. By well-foundedness there is an earliest stage v in the run such that $\neg Q(p_v, s_v, t_v)$. The event e_v is an output event such that

$$\sigma(m_0, e_v^o) \not\subseteq \{\{m_0, A_0\}_{Pub(B_0)}\}\ .$$

In proving the secrecy Theorem 5.4.2 for the initiator's nonce we showed that there is a name $n \in \mathbf{N}$ such that for every stage w in an NSL run containing event a_1

$$\sigma(t_w, m_0) \subseteq \{\{m_0, A_0\}_{Pub(A_0)}, \{m_0, n, B_0\}_{Pub(B_0)}\}\ ,$$

provided that $Priv(A_0), Priv(B_0) \notin t_0$. The event a_2 is also included in the run, and so $n = n_0$. Therefore

$$\sigma(m_0, e_v^o) = \{\{m_0, n_0, B_0\}_{Pub(A_0)}\}\ .$$

Clearly e_v follows a_1 but precedes a_2 in the run.

$$\begin{tikzcd}
a_1 \arrow[d] \arrow[rrd] & & \\
a_2 & & e_v \arrow[ll]
\end{tikzcd}\ .$$

We inspect the output events of NSL to determine which event e_v is. It is immediate to see that e_v can't be an initiator event. The spy output events can be excluded for a reason similar to the one we saw in the previous proof. It remains to check the responder output events. There is only one possible case:

$$act(e_v) = resp : B : j_0 : out\, new\, n_0\ \{m_0, n_0, B_0\}_{Pub(A_0)}$$

for some round index j_0. Let $b_2 = e_v$. By control precedence, the run contains a responder event b_1 which precedes b_2 and carries the action

$$act(b_1) = resp : B : j_0 : in\ \{m_0, A_0\}_{Pub(B_0)}\ .$$

Therefore, using the freshness principle as in the previous proof, one obtains

$$\begin{tikzcd}
a_1 \arrow[r, "\{m_0, A_0\}_{Pub(B_0)}"] \arrow[d] & b_1 \arrow[d] \\
a_2 & b_2 \arrow[l, "\{m_0, n_0, B_0\}_{Pub(A_0)}"']
\end{tikzcd}\ .$$

□

It is worth observing that to guarantee authentication to a responder it is only required that the private key of the initiator is not corrupted. Even if the responder uses a corrupted private key for decryption, it does not affect the level of authentication achieved. The authentication property that we showed for the initiator, however, fails when the private key of the initiator is corrupted. [1] Assume that $Priv(B_0) \notin t_0$ but $Priv(A_0) \in t_0$ and assume that the run contains initiator events a_1 and a_2. When we try to redo the proof of authentication we encounter the following difficulty: The property on message surroundings that we are considering is of the form

$$\sigma(m_0, t) \subseteq \{\{m_0, A_0\}_{Pub(B_0)}, \{m_0, n, B_0\}_{Pub(A_0)}, \dots\}$$

and therefore when examining the spy events, Theorem 5.2.4 can only be applied if both $Priv(A_0)$ and $Priv(B_0)$ are not corrupted. The spy is therefore able to change the surroundings of m_0 when it listens to a message $\{m_0, n, B_0\}_{Pub(A_0)}$ and can produce fake surroundings for m using the available public keys. Suppose that n_0 is a name that the spy knows in addition to agent names A_0, B_0 and the public key $Pub(A_0)$ – suppose, for example, that these names and key are in cleartext on the network. The following attack is possible:

$$\begin{tikzcd}
a_1 \arrow[rr, "\{m_0,A_0\}_{Pub(B_0)}"] \arrow[d] & & b_1 \arrow[d] \\
a_2 & \langle spy \rangle \arrow[l, "\{m_0,n_0,B_0\}_{Pub(A_0)}"'] & b_2 \arrow[l, "\{m_0,n,B_0\}_{Pub(A_0)}"']
\end{tikzcd}$$

where $\langle spy \rangle$ is the following diagram:

$$\begin{tikzcd}
 & s_1 \arrow[d] & \bullet \arrow[l, "Priv(A_0)"'] & & \\
b_2 \arrow[r, "\{m_0,n,B_0\}_{Pub(A_0)}"] & s_2 \arrow[d] & & & \\
 & s_3 \arrow[r, dotted, "m_0"] & s_4 \arrow[d] & & \\
 & \bullet \arrow[r, dotted, "{n_0,B_0}"] & s_5 \arrow[d] & s_7 \arrow[d] & \bullet \arrow[l, "Pub(A_0)"'] \\
 & & s_6 \arrow[r, "{m_0,n_0,B_0}"] & s_8 \arrow[d] & \\
 & & & s_9 \arrow[r, "\{m_0,n_0,B_0\}_{Pub(A_0)}"] & a_2
\end{tikzcd}$$

The dotted lines involve more steps from the spy, these however are elementary and just involve composing or decomposing message tuples.

[1] This observation is not new. It can for example be found in [90].

The attack shows how the initiator can be tricked into believing that a certain nonce is the nonce of B_0 even if, in reality, it is some other value chosen by the spy. The authentication property that we can show when both private keys are not corrupted is clearly violated. A weaker form of authentication still holds even if $Priv(A_0)$ is corrupted. It says that to a run of the initiator there corresponds a run of the responder. The two runs, however, not necessarily agree on the chosen values. For some applications this weaker form of authentication might be enough.

The nonces that initiator and responder exchange during an NSL session could be used to generate a common session key. It would be a security breach if nonces that are exchanged during a protocol session appear in any other protocol session, especially in a session which involves different agents. The authentication properties for NSL that have been studied earlier in the chapter ensure non-injective agreement [48] – they do not guarantee that to an initiator run corresponds a unique responder run that agrees on nonces. The following theorems show that injective agreement holds for NSL runs.

Theorem 5.4.6 *Let m_0, n_0 be names. In every run of NSL there exists at most one initiator event a_2 carrying an action*

$$act(a_2) \quad = \quad init : (A, B) : i : in\ \{m_0, n_0, B\}_{Pub(A)}$$

or an action

$$act(a_2) \quad = \quad init : (A, B) : i : in\ \{n_0, m_0, B\}_{Pub(A)}$$

where A, B are agent names such that $Priv(A), Priv(B) \not\sqsubseteq t_0$ and i a round index.

Proof. Suppose that there exists a run of NSL containing two events a_2 and a_2' carrying actions

$$\begin{array}{rcl} act(a_2) & = & init : (A, B) : i : in\ \{m_0, n_0, B\}_{Pub(A)} \\ act(a_2') & = & init : (A', B') : i' : in\ \{m_0, n_0, B'\}_{Pub(A')}\ . \end{array}$$

The events a_2 and a_2' need to be distinct (Proposition 4.4.4). By control precedence and the token game, there exist two distinct events a_1 and a_1' in the run such that $Fresh(m_0, a_1)$ and $Fresh(m_0, a_1')$. This, however, is a contradiction to the freshness principle.

Suppose instead, that there exists a run of NSL containing two events a_2 and a_2' with actions

$$\begin{array}{rcl} act(a_2) & = & init : (A, B) : i : in\ \{m_0, n_0, B\}_{Pub(A)} \\ act(a_2') & = & init : (A', B') : i' : in\ \{n_0, m_0, B'\}_{Pub(A')} \end{array}$$

and where $Priv(A), Priv(B), Priv(A'), Priv(B') \not\sqsubseteq t_0$. By control precedence there exist two initiator events a_1, a_1' in the run such that $Fresh(m_0, a_1)$ and $Fresh(n_0, a_1')$. On the other hand, from Theorem 5.4.5 it follows that the run contains two responder events b_2, b_2' such that $Fresh(n_0, b_2)$ and $Fresh(m_0, b_2')$. From the freshness principle follows a contradiction since responder and initiator events are different events. □

Similarly one can show that:

Theorem 5.4.7 *Let m_0, n_0 be names. In every run of NSL there exists at most one responder event b_2 carrying an action*

$$act(b_2) \quad = \quad resp : B : i : out\, new\, n_0 \ \{m_0, n_0, B\}_{Pub(A)}$$

or an action

$$act(b_2) \quad = \quad resp : B : i : out\, new\, m_0 \ \{n_0, m_0, B\}_{Pub(A)}$$

where A, B are agent names, i a round index and where $Priv(B) \not\sqsubseteq t_0$. □

One could also try to extend the analysis and study the consequences of nonces getting corrupted at some stage during a protocol run. A corrupted session key should, if possible, not compromise the security of later sessions. The issue of session-key compromise is studied in more detail for the ISO protocol in the next chapter.

Chapter 6

Examples

The protocols that we study in this chapter are examples of how security protocols can be written and analysed in SPL. Protocol verification is based on the SPL net.

A class of security protocols concerns the establishment of secret shared keys between a number of agents. Agents can use the established keys for a variety of purposes – typically as session keys for further secure communication, for example to ensure confidentiality and authentication of transmitted data. One can distinguish between two main approaches to secret-key establishment [52]. The *key-transport* approach, where one of the agents or a trusted third party chooses the key and sends it to the other agents. In the *key-agreement* approach, instead, agents derive a common key from information that they already have. The examples that we study in this chapter are protocols of the first kind. Their objective is the distribution of a secret key in a way that the key is shared only among the legitimate parties (key authentication) and, perhaps, ensure the key's freshness to prevent re-play attacks [52]. Key-transport protocols usually make use of cryptography to achieve their objectives. The NSL protocol that we studied earlier is a key-transport protocol that makes use of public-key cryptography. Symmetric-key cryptography can be used for key distribution as an alternative to public-key cryptography. Symmetric-key cryptography can be the preferred choice for time-critical applications since public-key encryption algorithms are rather slow (up to 1000 times slower than conventional symmetric-key cryptography [79]).

The ISO 5-pass authentication protocol [40, 19] and the Π_3 key-translation protocol [100, 99, 19] that we study in this chapter are both based on a symmetric-key cryptographic system.

6.1 The ISO 5-pass authentication protocol

Two agents, Alice and Bob want to establish a shared session key that can be used for encrypting confidential data. If they both share a key with a trusted third party, a server, they can ask the server to issue and send the session key to them. The ISO 5-pass protocol [40, 19] is a symmetric key authentication mechanism that allows to do so.

6.1.1 Informal description

At an informal level the protocol is described by the following sequence of actions:

$$\begin{array}{lll}
(1) & A \to B: & ra, T1 \\
(2) & B \to S: & rbs, ra, A, T2 \\
(3) & S \to B: & T5, \{rbs, Key(ab), A, T4\}_{K(B,S)}, \{ra, Key(ab), B, T3\}_{K(A,S)} \\
(4) & B \to A: & T7, \{ra, Key(ab), B, T3\}_{K(A,S)}, \{rba, ra, T6\}_{Key(ab)} \\
(5) & A \to B: & T9, \{ra, rba, T8\}_{Key(ab)} \quad .
\end{array}$$

Agents that engage in the protocol can take two roles; the initiator, here A and the responder, here B. The server S is a fixed trusted entity. When A initiates the protocol (message (1)) it sends to B a random number ra, a challenge for the server. This challenge, when encrypted with the key shared by A and S, aims at providing authentication [1] of S to A and freshness of the cyphertext in which it is contained. After receiving the request from A, the responder B interacts with the server to request a session key – together with A's challenge and identifier, it sends a random number rbs, his own challenge for S (message (2)). Upon receiving the message from B, the server sets up a session key $Key(ab)$ for the two principals and prepares two encryptions: one in which the session key $Key(ab)$ and B's challenge rbs are encrypted with the key S shares with B and another in which the session key $Key(ab)$ and A's challenge ra are encrypted with the key S shares with A. This compound message is sent to the responder (3). The responder decrypts the first part and checks whether it got back the same random number he chose in step (2). If so, B concludes that the message received is fresh and was indeed sent by S. To make sure that the session key is shared only with A, the responder exchanges two more messages with the initiator. In (4) B forwards to A part of the message received from S and an encryption using the session key and containing rba, a challenge for A. After receiving the message, the initiator decrypts the first part, extracts the new key and then checks whether it got back the challenge sent in (1). If so, A decrypts the second part of the message and authenticates B checking for ra. If all checks are successful the initiator concludes that it got a good session key which is shared only with B. In step (5) the initiator acknowledges the responder so that B can authenticate A and equally conclude that the session key is shared only with A.

The protocol should guarantee that session keys stay secret to initiator, responder and the server that issued them, and that there is mutual authentication between initiator and responder.

6.1.2 Programming the ISO-5 pass authentication in SPL

In the above informal description of the protocol, T1-T9 stand for any text that one might want to include in the communication between the principals. To keep the messages shorter and make the proofs of correctness of the protocol more readable

[1] As mentioned earlier, we assume symmetric-key encryptions to providing message authentication – to achieve this a MAC is used together with encryption.

we omit the texts T1-T9 entirely. We assume given a set of agent names $\mathcal{A}$. Let $A, B, S \in \mathcal{A}$. We can program the three different protocol roles as SPL-processes as follows:

$$\begin{array}{lcl}
Init(A,B) & \equiv & out\, new(x)\ x. \\
 & & in\ \{x, \chi, B\}_{Key(A,S)}, \{y, x\}_{\chi}. \\
 & & out\ \{x, y\}_{\chi} \\
 & & \\
Resp(A,B) & \equiv & in\ x. \\
 & & out\, new(z)\ z, x, A. \\
 & & in\ \{z, \chi, A\}_{Key(B,S)}, \psi. \\
 & & out\, new(y)\ \psi, \{y, x\}_{Key(B,S)}. \\
 & & in\ \{x, y\}_{\chi} \\
 & & \\
Server(A,B) & \equiv & in\ z, x, A. \\
 & & out\, new(z')\ \{z, Key(z'), A\}_{Key(B,S)}, \{x, Key(z'), Y\}_{Key(A,S)}\ .
\end{array}$$

The generation of a new key can be described conveniently with new name generation. If ab is a new name then $Key(ab)$ is a new term that we use to denote a new key. Therefore the session key-generation of the server is expressed by:

$$out\, new(z')\ \{\cdots Key(z') \cdots\}_{Key(B,S)}, \cdots \quad .$$

Suppose that $\mathcal{A}$ is the set of agents that can participate in the protocol and suppose that each agent can be both initiator and responder with any other agent. This rather general protocol system can be described by the following parallel composition of processes. We don't, however, want to study the protocol in isolation, instead we analyse its executions in a hostile environment. Therefore, as we did for NSL, we add a spy. The process Spy has all the components describe in Section 4.1.10. The ISO protocol system is described by the following SPL-process term:

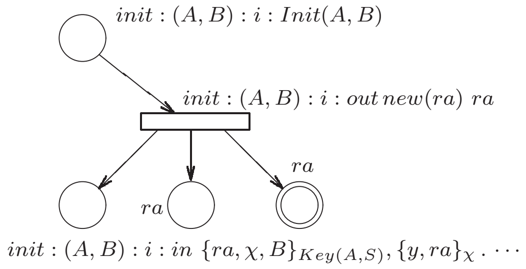
$$\begin{array}{lcl}
P_{init} & \equiv & \|_{A,B \in \mathcal{A} \times \mathcal{A}}\ !\ Init(A,B) \\
P_{resp} & \equiv & \|_{A,B \in \mathcal{A} \times \mathcal{A}}\ !\ Resp(A,B) \\
P_{server} & \equiv & \|_{A,B \in \mathcal{A} \times \mathcal{A}}\ !\ Server(A,B) \\
P_{spy} & \equiv & Spy \\
 & & \\
ISO & \equiv & \|_{i \in \{resp, init, server, spy\}}\ P_i \quad .
\end{array}$$

6.1.3 The events of the protocol

We classify the events $Ev(ISO)$ of the ISO protocol:

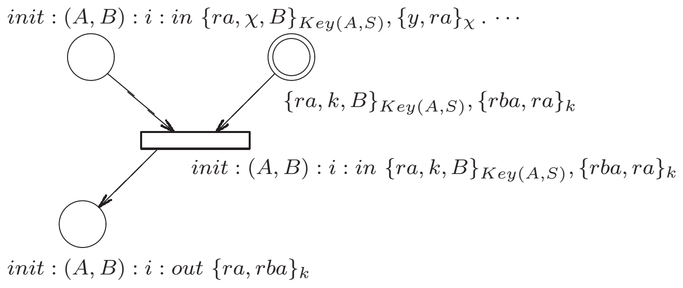
Initiator events for every agent name $A, B \in \mathcal{A}$, names $ra, rba \in \mathbf{N}$, key k, and indices $i \in \omega$.

$init : (A, B) : i : \mathbf{Out}(Init(A, B); ra)$



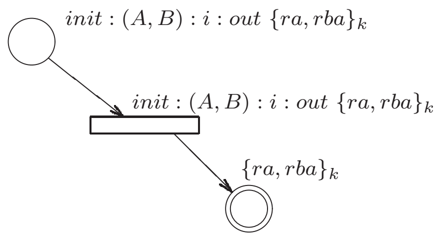
$init : (A, B) : i : Init(A, B)$
$init : (A, B) : i : out\,new(ra)\ ra$
ra
ra
$init : (A, B) : i : in\ \{ra, \chi, B\}_{Key(A,S)}, \{y, ra\}_{\chi} . \cdots$

$init : (A, B) : i : \mathbf{In}(in\ \{ra, \chi, B\}_{Key(A,S)}, \{y, ra\}_{\chi} . \cdots ; rba, k)$



$init : (A, B) : i : in\ \{ra, \chi, B\}_{Key(A,S)}, \{y, ra\}_{\chi} . \cdots$
$\{ra, k, B\}_{Key(A,S)}, \{rba, ra\}_k$
$init : (A, B) : i : in\ \{ra, k, B\}_{Key(A,S)}, \{rba, ra\}_k$
$init : (A, B) : i : out\ \{ra, rba\}_k$

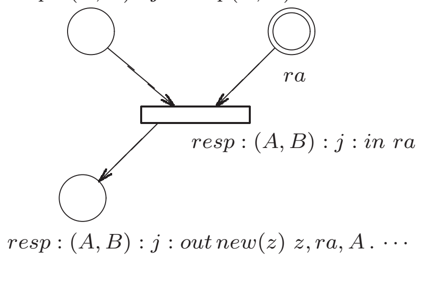
$init : (A, B) : i : \mathbf{Out}(out\ \{ra, rba\}_k)$



$init : (A, B) : i : out\ \{ra, rba\}_k$
$init : (A, B) : i : out\ \{ra, rba\}_k$
$\{ra, rba\}_k$

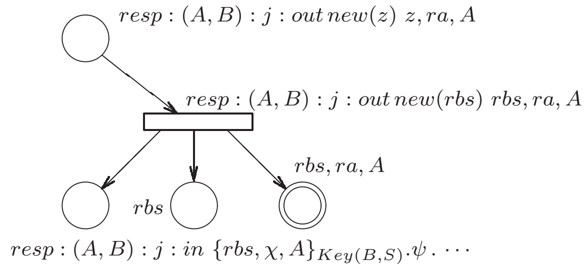
Responder events for every agent name $A, B \in \mathcal{A}$, names $ra, rba, rbs \in \mathbf{N}$, key k, and indices $j \in \omega$.

$resp : (A, B) : j : \mathbf{In}(Resp(A, B); ra)$

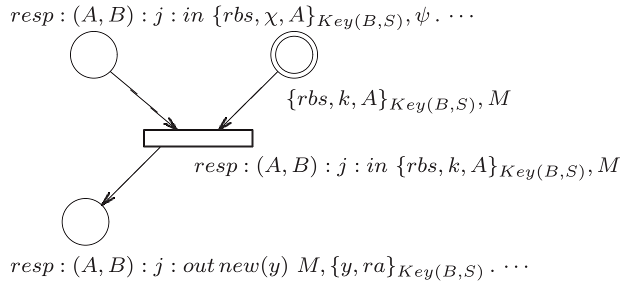


$resp : (A, B) : j : Resp(A, B)$
ra
$resp : (A, B) : j : in\ ra$
$resp : (A, B) : j : out\,new(z)\ z, ra, A . \cdots$

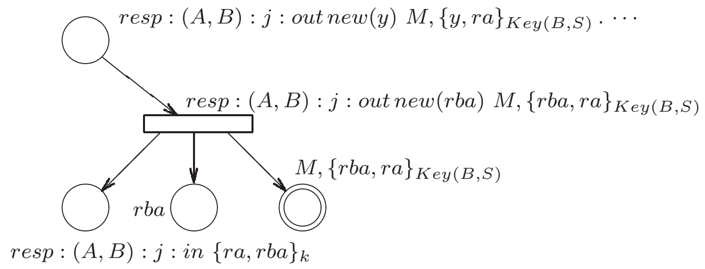
$resp : (A,B) : j : \mathbf{Out}(out\, new(z)\ z, ra, A \,.\, \cdots ; rbs)$


$resp : (A,B) : j : out\, new(z)\ z, ra, A$
$resp : (A,B) : j : out\, new(rbs)\ rbs, ra, A$
rbs, ra, A
rbs
$resp : (A,B) : j : in\ \{rbs, \chi, A\}_{Key(B,S)} . \psi \,.\, \cdots$

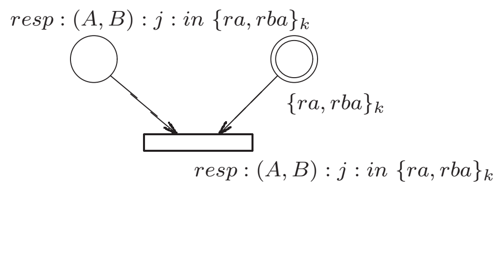
$resp : (A,B) : j : \mathbf{In}(in\ \{rbs, \chi, A\}_{Key(B,S)}, \psi \,.\, \cdots ; k, M)$


$resp : (A,B) : j : in\ \{rbs, \chi, A\}_{Key(B,S)}, \psi \,.\, \cdots$
$\{rbs, k, A\}_{Key(B,S)}, M$
$resp : (A,B) : j : in\ \{rbs, k, A\}_{Key(B,S)}, M$
$resp : (A,B) : j : out\, new(y)\ M, \{y, ra\}_{Key(B,S)} \,.\, \cdots$

$resp : (A,B) : j : \mathbf{Out}(out\, new(y)\ M, \{y, ra\}_{Key(B,S)} \,.\, \cdots ; rba)$

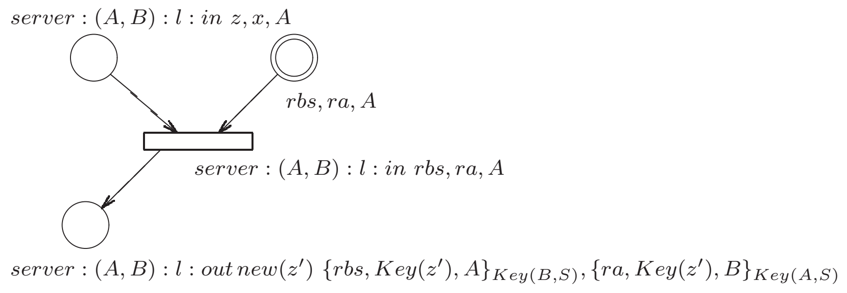

$resp : (A,B) : j : out\, new(y)\ M, \{y, ra\}_{Key(B,S)} \,.\, \cdots$
$resp : (A,B) : j : out\, new(rba)\ M, \{rba, ra\}_{Key(B,S)}$
$M, \{rba, ra\}_{Key(B,S)}$
rba
$resp : (A,B) : j : in\ \{ra, rba\}_k$

$resp : (A,B) : j : \mathbf{In}(in\ \{ra, rba\}_k)$

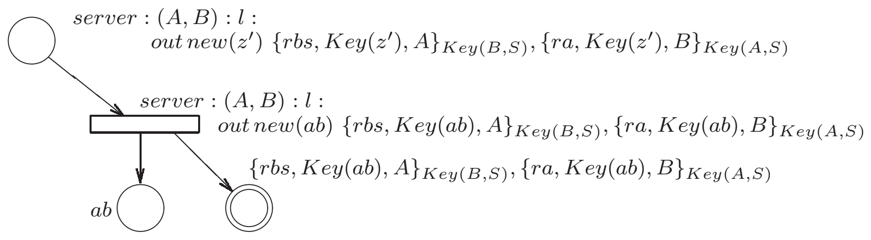

$resp : (A,B) : j : in\ \{ra, rba\}_k$
$\{ra, rba\}_k$
$resp : (A,B) : j : in\ \{ra, rba\}_k$

Server events for every agent name $A, B \in \mathcal{A}$, names $ra, rbs, ab \in \mathbf{N}$, and indices $l \in \omega$.

$server : (A, B) : l : \mathbf{In}(Server(A, B); rbs, ra)$


$server : (A, B) : l : in\ z, x, A$
rbs, ra, A
$server : (A, B) : l : in\ rbs, ra, A$
$server : (A, B) : l : out\,new(z')\ \{rbs, Key(z'), A\}_{Key(B,S)}, \{ra, Key(z'), B\}_{Key(A,S)}$

$server : (A, B) : l :$
$\quad \mathbf{Out}(out\,new(z')\ \{rbs, Key(z'), A\}_{Key(B,S)}, \{ra, Key(z'), B\}_{Key(A,S)}; ab)$


$server : (A, B) : l :$
$out\,new(z')\ \{rbs, Key(z'), A\}_{Key(B,S)}, \{ra, Key(z'), B\}_{Key(A,S)}$
$server : (A, B) : l :$
$out\,new(ab)\ \{rbs, Key(ab), A\}_{Key(B,S)}, \{ra, Key(ab), B\}_{Key(A,S)}$
$\{rbs, Key(ab), A\}_{Key(B,S)}, \{ra, Key(ab), B\}_{Key(A,S)}$
ab

6.1.4 Security properties

We formalise and prove a number of security properties for the ISO protocol. In this case we are interested in proper runs of the kind

$$\langle ISO, s_0, t_0\rangle \xrightarrow{e_1} \cdots \xrightarrow{e_w} \langle p_w, s_w, t_w\rangle \xrightarrow{e_{w+1}} \ldots$$

where t_0 is a network state, the messages on the network in which a protocol execution starts. Whenever we refer to a run of ISO in the following section we mean a run of this kind.

Secrecy of long-term keys and session keys. The ISO protocol makes use of long-term shared keys among agents and the server to distribute the session key and maintain it confidential. The protocol should not leak this shared keys otherwise all the communication is thwarted.

Theorem 6.1.1 *Let $A_0 \in \mathcal{A}$ be an agent name. Given a run of ISO, if the key $Key(A_0, S) \not\sqsubseteq t_0$ then at every stage w in the run $Key(A_0, S) \not\sqsubseteq t_w$.*

Proof. Suppose there is a run of ISO in which $Key(A_0, S)$ appears on the network. Since $Key(A_0, S) \not\sqsubseteq t_0$ there is a stage $v > 0$ in the run such that

$$Key(A_0, S) \not\sqsubseteq t_{v-1} \quad \text{and} \quad Key(A_0, S) \sqsubseteq t_v \ .$$

From the token game of nets with persistent conditions it follows that the event $e_v \in Ev(ISO)$ is such that

$$Key(A_0, S) \sqsubseteq e_v^o \ .$$

It can easily be checked that the shape of initiator events $e \in init : Ev(p_{init})$ and of server events $e \in server : Ev(p_{server})$ is such that $Key(A_0, S) \not\sqsubseteq e^o$. Even if messages sent by the server contain a key, the key is of the form $Key(ab)$ where ab is a name. Therefore $Key(ab)$ is syntactically different to $Key(A_0, S)$.

If $e_v \in resp : Ev(p_{resp})$ then it carries action

$$act(e) = resp : (A, B) : i : outnew(rba)\ M, \{rba, ra\}_k$$

where $A, B \in \mathcal{A}$, i an index, $rba, ra \in \mathbf{N}$, k is key and M is a message such that

$$Key(A_0, S) \sqsubseteq M \ .$$

By Control precedence there exists an event e_u such that

$$act(e_u) = resp : (A, B) : i : in\ \{rbs, k, A\}_{Key(B,S)}, M$$

where $rbs \in \mathbf{N}$ and

$$e_u \longrightarrow e_v \quad .$$

By the token game $M \sqsubseteq t_{u-1}$ and therefore $Key(A, S) \sqsubseteq t_{u-1}$ which is a contradiction since $u < v$.

The event e_v can't be a spy event. This can easily be checked using control precedence and the token game in a similar way as we did in the proof of secrecy for NSL private keys. □

It is desirable that the session keys distributed by the server remain secret. Only the agents that requested a session key, and of course the server that issued it, should have access to the key. In our setting secrecy of session keys is ensured by the following theorem:

Theorem 6.1.2 *If a run of ISO contains a server event s_2 such that*

$$\{rbs_0, Key(ab_0), A_0\}_{Key(B_0,S)} \sqsubseteq s_2^o$$

and if $Key(A_0, S), Key(B_0, S) \not\sqsubseteq t_0$ then at every stage w in the run $Key(ab_0) \notin t_w$.

Proof. We show a stronger property. We show that the key $Key(ab_0)$ never appears on the network in different surroundings than the ones prepared by the server. Consider the property on configurations $\langle p, s, t\rangle$

$$Q(p, s, t) \ \Leftrightarrow\ \sigma(Key(ab_0), t) \subseteq \sigma(Key(ab_0), s_2^o) \ .$$

The event s_2 is a server event such that

$$\{rbs_0, Key(ab_0), A_0\}_{Key(B_0,S)} \sqsubseteq s_2^o$$

and therefore

$$\sigma(Key(ab_0), s_2^o) = \{\{ra, Key(ab_0), B_0\}_{Key(A_0,S)}, \{rbs_0, Key(ab_0), A_0\}_{Key(B_0,S)}\}$$

for some name ra. If a run contains the event s_2 and if one can show that at every stage w in the run $Q(p_w, s_w, t_w)$ then $Key(ab_0) \notin t_w$ for every stage w in that run. Suppose the contrary. Suppose that at some stage in the run the property Q does not hold. Let v, by well-foundedness, be the first stage in the run such that $\neg Q(p_v, s_v, t_v)$. Since $Fresh(ab_0, s_2)$ it follows from the Freshness Principle 5.1.3 that

$$s_2 \longrightarrow e_v \quad .$$

Clearly $e_v \in Ev(ISO)$ and from the token game of nets with persistent conditions

$$\sigma(Key(ab_0), e_v^o) \nsubseteq \sigma(Key(ab_0), s_2^o) \quad .$$

The event e_v can only be an output event since $e^o = \emptyset$ for all input events e. We examine the possible output events of $Ev(ISO)$ and conclude that $e_v = s_2$ which is a contradiction.
Initiator output events. If $e_v \in init : Ev(p_{init})$ then $\sigma(Key(ab_0), e_v^o) = \emptyset$.
Responder output events. If $e_v \in resp : Ev(p_{resp})$ then it carries an action

$$act(e_v) = resp : (A, B) : i : out\, new(rbs)\ M, \{rbs, ra\}_k$$

where $A, B \in \mathcal{A}$, i an index, $rbs, ra \in \mathbf{N}$, k a key, and M a message such that

$$\sigma(Key(ab_0), M) \nsubseteq \sigma(Key(ab_0), s_2^o) \ .$$

By Control precedence there exists an event e_u such that

$$act(e_u) = resp : (A, B) : i : in\ \{rba, k, A\}_{Key(B,S)}, M$$

where $rba \in \mathbf{N}$ and

$$e_u \longrightarrow e_v \quad .$$

By the token game

$$\sigma(Key(ab_0), t_u) \nsubseteq \sigma(Key(ab_0), s_2^o)$$

which is a contradiction since $u < v$.
Server output events. If $e_v \in server : Ev(p_{server})$ and $\sigma(Key(ab_0), e_v^o) \neq \emptyset$, then $Fresh(ab, e_v)$. By Freshness it follows that $e_v = s_2$.
Spy output events. Since messages are persistent and v is a stage that follows the occurrence of s_2 in the run, $\sigma(Key(ab_0), t_{v-1}) \neq \emptyset$. Theorem 5.2.4 applies and so $e_v \notin spy : Ev(p_{spy})$. □

Authentication. The ISO 5-pass protocol should guarantee some degree of authentication between the principals. We first prove a useful lemma.

Lemma 6.1.3 *If a run of ISO is such that*

$$\{rbs_0, k_0, A_0\}_{Key(B_0,S)} \notin \sigma(k_0, t_0)$$

and $Key(B_0, S) \not\sqsubseteq t_0$ and if there exists a stage w in the run such that

$$\{rbs_0, k_0, A_0\}_{Key(B_0,S)} \in \sigma(k_0, t_w)$$

then the run contains a server event s_2 such that

$$\{rbs_0, k_0, A_0\}_{Key(B_0,S)} \sqsubseteq s_2^o$$

and $s_2 \longrightarrow e_{w+1}$.

Proof. Let w, by Well-foundedness, be the first stage in the run such that

$$\{rbs_0, k_0, A_0\}_{Key(B_0,S)} \in \sigma(k_0, t_w) \ .$$

By the token game

$$\{rbs_0, k_0, A_0\}_{Key(B_0,S)} \in \sigma(k_0, e_w^o) \ .$$

Clearly e_w can only be an output event. We examine the possible output events of $Ev(ISO)$ and conclude that $e_w = s_2$ and that

$$s_2 \longrightarrow e_{w+1} \quad .$$

Initiator output events. If $e_w \in init : Ev(p_{init})$ then $\sigma(k_0, e_w^o) = \emptyset$.
Responder output events. If $e_w \in resp : Ev(p_{resp})$ then it carries an action

$$act(e_w) = resp : (A, B) : i : out\, new(rbs)\ M, \{rbs, ra\}_k$$

where $A, B \in \mathcal{A}$, i an index, $rbs, ra \in \mathbf{N}$, and M a message such that

$$\{rbs_0, k_0, A_0\}_{Key(B_0,S)} \in \sigma(k_0, M) \ .$$

By Control precedence there exists an event e_v such that

$$act(e_v) = resp : (A, B) : i : in\ \{rbs, k, A\}_{Key(B,S)}, M$$

with rbs a name and

$$e_v \longrightarrow e_w \quad .$$

By the token game $\{rbs_0, k_0, A_0\}_{Key(B_0,S)} \in \sigma(k_0, t_v)$ which is a contradiction since $v < w$.
Spy output events. Since $Key(B_0, S) \not\sqsubseteq t_0$ Theorem 6.1.1 ensures that at every stage v in the run $Key(B_0, S) \notin t_v$. Theorem 5.2.5 applies and so $e_w \notin spy : Ev(p_{spy})$.
Server output events. Since it is the only remaining case there must exist a server event s_2 in the run such that $\{rbs_0, k_0, A_0\}_{Key(B_0,S)} \sqsubseteq s_2^o$. □

In an ISO 5-pass protocol exchange three participants are involved: the initiator, the responder and the server. The initiator relies on the responder to interact with the server and to get a session key. The interaction between responder and server is transparent to the initiator. From the responder point of view, however, a certain degree of authentication with the server is desired to ensure that the session keys are not obtained from some malicious intruder. Therefore one expects that to events of a responder which apparently show a communication with the server correspond matching events of the server.

Theorem 6.1.4 *If a run of ISO contains the responder event b_3 with action*

$$act(b_3) = resp : (A_0, B_0) : i_0 : in\ S, B_0, \{rbs_0, k_0, A_0\}_{Key(B_0,S)}, M_0$$

and if $Key(B_0, S) \not\sqsubseteq t_0$ then the run contains a server event s_2 such that

$$\{rbs_0, k_0, A_0\}_{Key(B_0,S)} \sqsubseteq s_2^o$$

where k_0 is a freshly created session key, i.e. $k_0 = Key(ab_0)$ with $Fresh(ab_0, s_2)$ and such that $s_2 \longrightarrow b_3$ *.*

Proof. By control precedence there exists an event b_2 with action

$$act(b_2) = resp : (A_0, B_0) : i_0 : out\, new(rbs_0)\ rbs_0, ra, A_0$$

for some name ra and such that

$$b_2 \longrightarrow b_3 \quad .$$

Therefore $Fresh(rbs_0, b_2)$ and by freshness

$$\{rbs_0, k_0, A_0\}_{Key(B_0,S)} \notin \sigma(k_0, t_0) \ .$$

From Lemma 6.1.3 it follows that there is a server event s_2 such that

$$\{rbs_0, k_0, A_0\}_{Key(B_0,S)} \sqsubseteq s_2^o$$

and such that

$$s_2 \longrightarrow b_3 \quad .$$

From the shape of server output events it follows that $k_0 = Key(ab_0)$ for some name ab_0 such that $Fresh(ab_0, s_2)$. □

One expects for example that when the responder B_0 finishes a session of the protocol apparently with the initiator A_0 then there was a session where A_0 executed the protocol as initiator apparently with B_0, and the events of the two participants agree on values of the exchanged messages such as for example the session key and the exchanged random challenges. This expectation is formalised as an authentication property of the responder with respect to the initiator.

Theorem 6.1.5 *If a run of ISO contains the responder event b_5 with action*

$$act(b_5) = resp : (A_0, B_0) : i_0 : in\ \{ra_0, rba_0\}_{k_0}$$

and if $Key(A_0, S), Key(B_0, S) \not\sqsubseteq t_0$ then the run contains initiator events a_1, a_2, a_3 carrying actions

$$\begin{aligned}
act(a_1) &= init : (A_0, B_0) : j : out\, new(ra_0)\ ra_0 \\
act(a_2) &= init : (A_0, B_0) : j : in\ \{ra_0, k_0, B_0\}_{Key(A_0,S)}, \{rba_0, ra_0\}_{k_0} \\
act(a_3) &= init : (A_0, B_0) : j : out\ \{ra_0, rba_0\}_{k_0} \ .
\end{aligned}$$

for some index j and $a_3 \longrightarrow b_5$ *.*

Proof. From control precedence it follows that

$$\begin{tikzcd} b_3 \arrow[d] \\ b_4 \arrow[d] \\ b_5 \end{tikzcd}$$

where

$$act(b_4) = resp : (A_0, B_0) : i_0 : out\, new(rba_0)\ M, \{rba_0, ra_0\}_{k_0}$$
$$act(b_3) = resp : (A_0, B_0) : i_0 : in\ \{rbs_0, k_0, A_0\}_{Key(B_0,S)}, M$$

for some name rbs_0 and message M. Since the run contains the event b_3 and since $Key(B_0, S) \notin t_0$ it follows from Theorem 6.1.4 that the run also contains a server event s_2 such that

$$\{rbs_0, k_0, A_0\}_{Key(B_0,S)} \sqsubseteq s_2^o\ .$$

Thus:

$$\begin{tikzcd} b_3 \arrow[d] & s_2 \arrow[l] \\ b_4 \arrow[d] & \\ b_5 & \end{tikzcd} .$$

Consider the property of configurations $\langle p, s, t\rangle$

$$Q(p, s, t) \ \Leftrightarrow\ \sigma(rba_0, t) \subseteq \{\{rba_0, ra_0\}_{k_0}\} \quad .$$

By freshness the property Q holds immediately after b_4, but clearly not immediately before b_5. By Well-foundedness there is an earliest stage v in the run such that $\neg Q(p_v, s_v, t_v)$. The event e_v is an output event such that

$$\sigma(rba_0, e_v^o) \not\subseteq \{\{rba_0, ra_0\}_{k_0}\}$$

and it follows b_4 but precedes b_5 in the run.

$$\begin{tikzcd} & b_3 \arrow[d] & s_2 \arrow[l] \\ & b_4 \arrow[d] \arrow[dl] & \\ e_v \arrow[r] & b_5 & \end{tikzcd} .$$

We inspect the output events of ISO to determine which event e_v is.
Spy output events. Since messages are persistent $\sigma(rba_0, t_{v-1}) \neq \emptyset$. Theorem 5.2.4 applies and so $e_v \notin spy : Ev(p_{spy})$.

Server output events. If $e_v \in server : Ev(p_{sever})$ then it carries an action of the form

$$\begin{aligned} act(e_v) = server : (A, B) : j :& \\ & out\, new(ab)\ \{rbs, Key(ab), A\}_{Key(B,S)}, \{ra, Key(ab), B\}_{Key(A,S)} \end{aligned}$$

such that $\sigma(rba_0, e_i^o) \neq \emptyset$, where $A, B \in \mathcal{A}$, j an index, $ra, rbs, ab \in \mathbf{N}$. The agent names A, B are such that $A, B \in s_0$. Since $Fresh(rba_0, b_4)$, by the freshness principle $A, B \neq rab_0$. It follows that $rbs = rba_0$ or $ra = rba_0$. By control precedence there exists an event e_u with action

$$act(e_u) = server : (A, B) : j : in\ rbs, ra, A$$

such that

$$\begin{tikzcd} e_u \arrow[r] & e_v \end{tikzcd} \quad .$$

Since $rbs = rba_0$ or $ra = rba_0$, by the token game,

$$\sigma(rba_0, t_{u-1}) \not\subseteq \{\{rba_0, ra_0\}_{k_0}\}$$

which contradicts our hypothesis.
Responder output events. If $e_v \in resp : Ev(p_{resp})$ then we distinguish two cases.
Case 1:

$$act(e_v) = resp : (A, B) : j : out\, new(rbs)\ rbs, ra, A$$

such that

$$\sigma(rba_0, rbs, ra, A) \not\subseteq \{\{rba_0, ra_0\}_{k_0}\} \ ,$$

where $A, B \in \mathcal{A}$, j an index, and $ra, rbs \in \mathbf{N}$. In this case, by freshness, $rbs \neq rba_0$. Therefore $ra = rba_0$. By control precedence there exists an event e_u with action

$$act(e_u) = resp : (A, B) : j : in\ rba_0$$

such that

$$\begin{tikzcd} e_u \arrow[r] & e_v \end{tikzcd} \quad .$$

Clearly, by the token game,

$$\sigma(rba_0, t_{u-1}) \not\subseteq \{\{rba_0, ra_0\}_{k_0}\}$$

which contradicts our hypothesis.
Case 2:

$$act(e_v) = resp : (A, B) : j : outnew(rba)\ M, \{rba, ra\}_k$$

where $A, B \in \mathcal{A}$, j an index, $rba, ra \in \mathbf{N}$, k a key and M a message. If

$$\sigma(rba_0, M) \not\subseteq \{\{rba_0, ra_0\}_{k_0}\}$$

then, by control precedence there exists an event e_u with action

$$act(e_u) = resp : (A, B) : j : in\ N, M$$

for some message N and such that

$$\begin{tikzcd} e_u \arrow[r] & e_v \end{tikzcd} \quad .$$

By the token game,
$$\sigma(rba_0, t_{u-1}) \not\sqsubseteq \{\{rba_0, ra_0\}_{k_0}\}$$
which contradicts our hypothesis.
If
$$\sigma(rba_0, \{rab, ra\}_k) \not\sqsubseteq \{\{rba_0, ra_0\}_{k_0}\}$$
then by freshness $rab \neq rba_0$ and therefore $ra = rba_0$. In this case, by control precedence one reaches a contradiction as in Case 1.
Initiator output events. If $e_v \in init : Ev(p_{init})$ then we distinguish two cases:
Case 1:
$$act(e_v) = init : (A, B) : j : out\, new(rba_0)\ rba_0$$
where $A, B \in \mathcal{A}$ and j an index. This case is excluded by freshness since $Fresh(rba_0, b_4)$.
Case 2:
$$act(e_v) = init : (A, B) : j : out\ \{ra, rba\}_{k_0}$$
where $A, B \in \mathcal{A}$, j an index, and $ra, rba \in \mathbf{N}$. This case is the only possible case remaining. Note that $rba_0 \sqsubseteq \{ra, rba\}_{k_0}$.

From the previous case analysis it follows that

$$\begin{tikzcd}
 & b_3 \arrow[d] & s_2 \arrow[l] \\
 & b_4 \arrow[d] \arrow[dl] & \\
a_3 \arrow[r] & b_5 &
\end{tikzcd}$$

where
$$act(a_3) = init : (A, B) : j : out\ \{ra, rba\}_{k_0}\ .$$
By control precedence

$$\begin{tikzcd}
a_1 \arrow[d] & b_3 \arrow[d] & s_2 \arrow[l] \\
a_2 \arrow[d] & b_4 \arrow[d] \arrow[dl] & \\
a_3 \arrow[r] & b_5 &
\end{tikzcd}$$

where

$$act(a_1) = init : (A, B) : j : outnew(ra)\ ra$$
$$act(a_2) = init : (A, B) : j : in\ \{ra, k_0, B\}_{Key(A,S)}, \{rba, ra\}_{k_0}\quad .$$

Clearly by freshness $ra \neq rba_0$, therefore $rba = rba_0$ since $rba_0 \sqsubseteq \{ra, rba\}_{k_0}$. Moreover $ra = ra_0$ since
$$\begin{tikzcd} a_2 \arrow[r] & a_3 \end{tikzcd}$$
and a_3 is the first event such that $\sigma(rba_0, a_3) \notin \{rba_0, ra_0\}_{k_0}$. In the run there is an event s_2 such that
$$\{rbs_0, k_0, A_0\}_{Key(B_0,S)} \sqsubseteq s_2^o\ .$$

From the proof of Theorem 6.1.2 (secrecy of session keys), it follows that at every stage w in the run

$$\sigma(k_0, t_w) \subseteq \{\{ra, k_0, B_0\}_{Key(A_0,S)}, \{rbs_0, k_0, A_0\}_{Key(B_0,S)}\}$$

for some name ra, therefore $B = B_0$ and $A = A_0$. We conclude that

$$\begin{aligned}
&act(a_1) = init : (A_0, B_0) : j : outnew(ra_0)\ ra_0\\
&act(a_2) = init : (A_0, B_0) : j : in\ \{ra_0, k_0, B_0\}_{Key(A_0,S)}, \{rba_0, ra_0\}_{k_0}\\
&act(a_3) = init : (A_0, B_0) : j : out\ \{ra_0, rba_0\}_{k_0}
\end{aligned}$$

and by freshness

$$\begin{tikzcd}
a_1 \arrow[d] & b_3 \arrow[d] & s_2 \arrow[l] \\
a_2 \arrow[d] & b_4 \arrow[l] \arrow[d] & \\
a_3 \arrow[r] & b_5 & .
\end{tikzcd}$$

□

The protocol also guarantees authentication of the server to the initiator. Despite initiator and server not interacting directly, one can show that when the initiator gets a session key, there was a run of the server that generated that key. This property turns out to be of central importance later in this section, in proving authentication of the initiator with respect to the responder.

Theorem 6.1.6 *If a run of ISO contains the initiator event a_2 with action*

$$act(a_2) = init : (A_0, B_0) : i_0 : in\ \{ra_0, k_0, B_0\}_{Key(A_0,S)}, \{rba_0, ra_0\}_{k_0}$$

and if $Key(A_0, S) \not\sqsubseteq t_0$ then the run contains the server event s_2 such that

$$\begin{tikzcd} s_2 \arrow[r] & a_2 \end{tikzcd}$$

and such that

$$\{ra_0, k_0, B_0\}_{Key(A_0,S)} \sqsubseteq s_2^o$$

where k_0 is a freshly created session key, i.e. $k_0 = Key(ab_0)$ with $Fresh(s_2, ab_0)$.

Proof. Since $Key(A_0, S) \not\sqsubseteq t_0$, from Lemma 6.1.1 it follows that at every stage w in the run $Key(A_0, S) \not\sqsubseteq t_w$. Moreover by control precedence there exists an event a_1 with action

$$act(a_1) = Resp : (A_0, B_0), i_0 : out\, new(ra_0)\ ra_0$$

and such that

$$\begin{tikzcd} a_1 \arrow[d] \\ a_2 \ . \end{tikzcd}$$

Therefore $Fresh(ra_0, a_1)$ and by freshness

$$\{ra_0, k_0, B_0\}_{Key(A_0,S)} \notin \sigma(k_0, t_0) \ .$$

From Lemma 6.1.3 it follows that there is a server event s_2 such that

$$\{ra_0, k_0, B_0\}_{Key(A_0,S)} \sqsubseteq s_2^o$$

and such that

$$\begin{tikzcd} s_2 \arrow[r] & a_2 \end{tikzcd} \ .$$

From the shape of the server output events it follows that $k_0 = Key(ab_0)$ for some name ab_0 such that $Fresh(s_2, ab_0)$. □

The protocol should guarantee authentication of the responder to the initiator.

Theorem 6.1.7 *If a run of ISO contains the initiator event a_2 with action*

$$act(a_2) = init : (A_0, B_0) : i_0 : in\ \{ra_0, k_0, B_0\}_{Key(A_0,S)}, \{rba_0, ra_0\}_{k_0}$$

and if $Key(A_0, S), Key(A_0, S) \not\sqsubseteq t_0$ then the run contains responder events b_1 and b_4 with actions

$$act(b_1) = resp : (A_0, B_0) : j : in\ ra_0$$
$$act(b_4) = resp : (A_0, B_0) : j : out\, new(rba_0)\ \{ra_0, k_0, B_0\}_{Key(A_0,S)}, \{rba_0, ra_0\}_{k_0} \ .$$

for some index j and such that $\begin{tikzcd} b_4 \arrow[r] & a_2 \end{tikzcd}$.

Proof. From control precedence it follows that

$$\begin{tikzcd} a_1 \arrow[d] \\ a_2 \end{tikzcd}$$

where

$$act(a_1) = init : (A_0, B_0) : i_0 : out\, new(ra_0)\ ra_0 \ .$$

Consider the property of configurations $\langle p, s, t\rangle$

$$Q(p, s, t) \ \Leftrightarrow\ \{rba_0, ra_0\}_{k_0} \notin \sigma(ra_0, t) \text{ and } \{ra_0, rba_0\}_{k_0} \notin \sigma(ra_0, t) \quad .$$

By freshness the property Q holds immediately after a_1, but clearly not immediately before a_2. By well-foundedness there is an earliest stage v in the run such that $\neg Q(p_v, s_v, t_v)$. The event e_v is an output event such that

$$\{rba_0, ra_0\}_{k_0} \in \sigma(ra_0, e_v^o) \quad \text{or} \quad \{ra_0, rba_0\}_{k_0} \in \sigma(ra_0, e_v^o)$$

and occurs after a_1 but precedes a_2 in the run.

$$\begin{tikzcd} a_1 \arrow[d] \arrow[dr] & \\ a_2 & e_v \arrow[l] \end{tikzcd} \ .$$

We inspect the output events of ISO to determine which event e_v is.
Spy output events. Theorem 5.2.5 applies and so $e_v \notin spy : Ev(p_{spy})$.
Server output events. If $e_v \in server : Ev(p_{sever})$ clearly $\{rba_0, ra_0\}_{k_0} \notin \sigma(ra_0, e_v^o)$ and $\{ra_0, rba_0\}_{k_0} \notin \sigma(ra_0, e_v^o)$.
Initiator output events. If $e_v \in init : Ev(p_{init})$ distinguish the following cases:
Case 1:

$$act(e_v) = init : (A, B) : j : out\, new(ra_0)\ ra_0$$

where $A, B \in \mathcal{A}$ and j is an index. Obviously

$$\{rba_0, ra_0\}_{k_0} \notin \sigma(ra_0, e_v^o) \quad \text{and} \quad \{ra_0, rba_0\}_{k_0} \notin \sigma(ra_0, e_v^o)\ .$$

Case 2:

$$act(e_v) = init : (A, B) : j : out\ \{rba_0, ra_0\}_{k_0}$$

where $A, B \in \mathcal{A}$ and j an index. By Control precedence there an event e_u with action

$$act(e_u) = init : (A, B) : j : in\ \{rba_0, k_0, B\}_{Key(A,S)}, \{ra_0, rba_0\}_{k_0}$$

an by the token game

$$\{ra_0, rba_0\}_{k_0} \in \sigma(ra_0, t_{u-1})$$

which is a contradiction.
Case 3:

$$act(e_v) = init : (A, B) : j : out\ \{ra_0, rba_0\}_{k_0}$$

where $A, B \in \mathcal{A}$ and j an index. This case is excluded in a similar way as is Case 2.
Responder output events. If $e_v \in resp : Ev(p_{resp})$ distinguish the following cases:
Case 1:

$$act(e_v) = resp : (A, B) : j : out\, new(rbs)\ rbs, ra, A$$

where $A, B \in \mathcal{A}$, j an index, and $rbs, ra \in \mathbf{N}$. Clearly this case is not possible.
Case 2:

$$act(e_v) = resp : (A, B) : j : out\, new(ra_0)\ \{rba_0, k_0, B\}_{Key(A,S)}, \{ra_0, rba_0\}_{k_0}$$

where $A, B \in \mathcal{A}$ and j an index. This case is not possible by freshness.
Case 3:

$$act(e_v) = resp : (A, B) : j : out\, new(rba_0)\ \{ra_0, k_0, B\}_{Key(A,S)}, \{rba_0, ra_0\}_{k_0}$$

where $A, B \in \mathcal{A}$ and j an index. Since the run contains the event a_2 it also contains the a server event s_2 such that

$$\{ra_0, k_0, B_0\}_{Key(A_0,S)} \sqsubseteq s_2^o\ .$$

(Theorem 6.1.6). From the proof of Theorem 6.1.2 (secrecy of session keys), it follows that at every stage w in the run

$$\sigma(k_0, t_w) \subseteq \{\{ra_0, k_0, B_0\}_{Key(A_0,S)}, \{rbs, k_0, A_0\}_{Key(B_0,S)}\}$$

for some name rbs, therefore $B = B_0$ and $A = A_0$.

From the previous case analysis and control precedence it follows that

$$\begin{tikzcd} a_1 \arrow[d] & b_1 \arrow[d] \\ a_2 & b_4 \arrow[l] \end{tikzcd}$$

where

$act(b_1) = resp : (A_0, B_0) : j : in\ ra_0$
$act(b_4) = resp : (A_0, B_0) : j : out\, new(rba_0)\ \{ra_0, k_0, B_0\}_{Key(A_0,S)}, \{rba_0, ra_0\}_{k_0}$.

as desired. □

Key compromise. It can happen that a session key, which is used by the initiator and the responder to perform "secure" communications after it has been distributed, gets corrupted. The session keys that are distributed with the ISO protocol satisfy a nice property. If they get corrupted, only the session in which they got corrupted is thwarted. The spy can't trick participants in the protocol in accepting the corrupted key as if it was a recently generated session key to be used again for another session – clearly this can only be true when none of the session participants is corrupted. Key compromise has been studied successfully for some protocols using the BAN Logic [12], the inductive method of Paulson [63] and strand spaces [35]. The approach that we use, is event based, and has similarities to the one suggested for the strand-space model even if we don't define, in this thesis, a notion of recentness as done for the strand-space model.

The inability of the spy to convince an initiator or a responder to use a corrupted session key for a different session to the session in which the key got corrupted, is ensured if the protocol runs adhere to a certain shape: runs don't contain more than one responder and one initiator event concerning the acceptance of the same session key. To model the corruption of a session key we add one more capability to the spy:

$$Spy_{10} \equiv in\ \{\psi\}_{Key(z')} \,.\, out\ Key(z') \quad .$$

This is clearly a very strong capability, that in fact allows all session keys to be corrupted. We show that any corrupted key will be accepted as a "good" key only once, namely in the session in which it got corrupted.

Observe that we can safely make use of previous theorems in proving the following results, even if the spy now has the Spy_{10} component. In all previous theorems and lemmas the limitation results of the spy are used only with respect to long-term keys of the form $Key(A, S)$ and $Key(B, S)$ – encryptions with these long-term keys are not made accessible by the term Spy_{10}.

The responder guarantee goes as follows:

Theorem 6.1.8 *Let k_0 be a key. Every run of ISO contains at most one occurrence of a responder event with action*

$$resp : (A, B) : i : in\ \{rbs, k_0, A\}_{Key(B,S)}, M$$

where $A, B \in \mathcal{A}$ such that $Key(B, S) \not\sqsubseteq t_0$, $rbs \in \mathbf{N}$, M a message, and index i.

Proof. Suppose that there exist two distinct event occurrences of the responder events b_3 and b'_3 in the run carrying actions

$$\begin{aligned} act(b_3) &= resp : (A_0, B_0) : i_0 : in\ \{rbs_0, k_0, A_0\}_{Key(B_0,S)} \\ act(b'_3) &= resp : (A_1, B_1) : i_1 : in\ \{rbs_1, k_0, A_1\}_{Key(B_1,S)} \end{aligned}$$

and suppose that $Key(B_0, S), Key(B_1, S) \not\sqsubseteq t_0$. From Control precedence it follows that

$$b_2 \longrightarrow b_3 \quad \text{and} \quad b'_2 \longrightarrow b'_3$$

where the events b_2 and b'_2 are such that $Fresh(rbs_0, b_2)$ and $Fresh(rbs_1, b'_2)$. Clearly b_2 and b'_2 are not the same event, else b_3 and b'_3 would be the same event which contradicts Proposition 4.4.4 (an event never occurs more than once in a run). It follows, by freshness, that $rbs_0 \neq rbs_1$.

Since $Fresh(rbs_0, b_2)$, by freshness

$$\{rbs_0, k_0, A_0\}_{Key(B_0,S)} \notin \sigma(k_0, t_0)\ .$$

From Lemma 6.1.3 it follows that there is a server event s_2 in the run such that

$$\{rbs_0, k_0, A_0\}_{Key(B_0,S)} \sqsubseteq s_2^o$$

and $k_0 = Key(ab_0)$ for some name ab_0 such that $Fresh(ab_0, s_2)$. Similarly one can show that the run contains another server event s'_2 such that

$$\{rbs_1, k_0, A_1\}_{Key(B_1,S)} \sqsubseteq {s'_2}^o$$

and $k_0 = Key(ab_0)$ such that $Fresh(ab_0, s_2)$. Therefore, by freshness it follows that s_2 and s'_2 are the same event. This, however is a contradictions since $rbs_0 \neq rbs_1$. □

A similar theorem guarantees that the initiator is never tricked in using a corrupted session key:

Theorem 6.1.9 *Let k_0 be a key. Every run of ISO contains at most one occurrence of an initiator event with action*

$$init : (A, B) : i : in\ \{ra, k_0, B\}_{Key(A,S)}, \{rba, ra\}_{k_0}$$

for every $A, B \in \mathcal{A}$ such that $Key(A, S), Key(B, S) \not\sqsubseteq t_0$, ra, rba names and index i. □

Observation. The proofs of the security theorems for the ISO 5-pass authentication mechanism reveal some weak points in the protocol: the server has no authentication guarantee; when the server gets a request for a new session key, it could have been made up by a stranger. Another weakness is that the strong agreement that exists between responder and initiator cannot be achieved between responder and server. Only a weaker authentication property holds (Theorem 6.1.4); if B in a protocol run acts as a responder and A as initiator then there is a corresponding run of the server. The server, however, can confuse initiator with responder and believe A is the responder in the protocol and not B. A spy as a middle man between responder and server can easily cause such confusion. Depending on the use one makes of this protocol, these weaknesses could constitute a security concern. To enforce strong agreement of the server and of the principals with respect to the server one could replace

$$B \longrightarrow S \;:\; rbs, ra, A, T2$$

$$\text{with} \quad B \longrightarrow S \;:\; \{rbs, ra, A, T2\}_{Key(B,S)}$$

Alternatively if one is concerned only about strengthening the authentication guarantee of the initiator and responder with respect to the server, the addition of two different shapes of messages T3 and T4 can achieve agreement – for example when the text T3 is the name of the initiator and T4 is omitted.

6.2 Woo-Lam key-translation protocol

Agent Alice wants to send a session key to Bob in a confidential way and so that some degree of authentication is ensured. To do so, if Alice does not already share a key with Bob but, instead, both Alice and Bob share keys with a trusted server, they could use a key translation protocol. In this section we study the Π_3 key translation protocol [100].

6.2.1 Informal decryption of the Π_3 protocol

The original Π_3 protocol is obtained from a more complex one via a number of simplifications and is slightly different from the protocol that we study, which we still call Π_3. To rule out an easy and well known attack [19] to the original version of the protocol (recalled at the end of this section) we add the name of the responder in the last massage exchanged in a protocol round.

Informally, the Π_3 protocol is described by the following sequence of actions:

$$\begin{array}{lll} (1) & A \rightarrow B: & A \\ (2) & B \rightarrow A: & n \\ (3) & A \rightarrow B: & \{n, Key(ab)\}_{K(A,S)} \\ (4) & B \rightarrow S: & \{A, \{n, Key(ab)\}_{K(A,S)}\}_{K(B,S)} \\ (5) & S \rightarrow B: & \{B, A, n, Key(ab)\}_{K(B,S)} \end{array}$$

The protocol has an initiator role, here A, a responder role, here B, and a trusted server that performs the translation, here S. The value n sent by the responder to the initiator in message (2) is a random challenge, a nonce, that has the purpose of ensuring "newness" of the distributed session key.

The session key $Key(ab)$ should remain confidential to the initiator, the responder and the trusted server. The protocol aims at providing the following form of authentication:

"Whenever a responder finishes a round of the protocol, the initiator of that protocol round is in fact the principal claimed in step (1) of the protocol" [100].

6.2.2 Programming the Π_3 protocol in SPL

Let $\mathcal{A}$ be the set of agents participating in the protocol and let $A, B, S \in \mathcal{A}$. The different roles of the protocol can be programmed as SPL processes in the following way:

$$\begin{array}{lcl} \mathit{Init}(A) & \equiv & \mathit{out}\ A. \\ & & \mathit{in}\ x. \\ & & \mathit{out}\, \mathit{new}(z)\ \{x, \mathit{Key}(z)\}_{\mathit{Key}(A,S)} \\ \\ \mathit{Resp}(A,B) & \equiv & \mathit{in}\ A. \\ & & \mathit{out}\, \mathit{new}(x)\ x. \\ & & \mathit{in}\ \psi. \\ & & \mathit{out}\ \{A, \psi\}_{\mathit{Key}(B,S)}. \\ & & \mathit{in}\ \{B, A, x, \chi\}_{\mathit{Key}(B,S)} \\ \\ \mathit{Server} & \equiv & \mathit{in}\ \{X, \{\psi'\}_{\mathit{Key}(X,S)}\}_{\mathit{Key}(Y,S)} \\ & & \mathit{out}\ \{Y, X, \psi'\}_{\mathit{Key}(Y,S)} \end{array}$$

The protocol system that we study is the following parallel composition of SPL-process terms:

$$\begin{array}{lcl} P_{init} & \equiv & \|_{A \in \mathcal{A}}\ !\ \mathit{Init}(A) \\ P_{resp} & \equiv & \|_{B \in \mathcal{A}}\ \|_{A \in \mathcal{A} \setminus \{B\}}\ !\ \mathit{Resp}(A,B) \\ P_{server} & \equiv & !\ \mathit{Server} \\ P_{spy} & \equiv & \mathit{Spy} \\ \\ \Pi_3 & \equiv & \|_{i \in \{init, resp, server, spy\}}\ P_i \end{array}$$

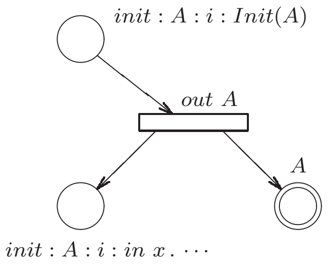
In this system every agent can be initiator of the protocol and every agent can respond to a protocol initiation done by any other agent. We exclude, however, the case where an agent B responds to a protocol initiation done by itself. If one would add $\mathit{Resp}(B,B)$ to the system, then an attack would go through. The attack is of the same kind of the one we exclude by adding the name of the responder in the last message exchanged in a protocol round – we recall this attack at the end of this section. The system contains a fixed server and, as done earlier for the ISO protocol, the system contains the general spy process Spy that we described in Section 4.1.10.

6.2.3 The events of the protocol

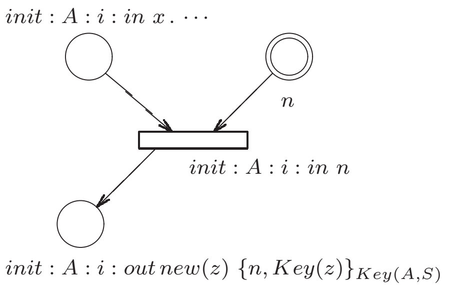
We classify the events $\mathit{Ev}(\Pi_3)$ of the Π_3 protocol. We don't list the spy events which belong to the Π_3 system – they have already been described earlier.

Initiator events for every agent name $A \in \mathcal{A}$, names n, ab, and index $i \in \omega$.

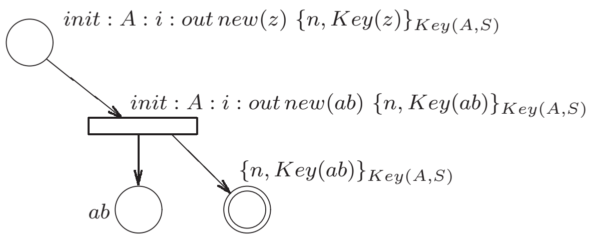
$init : A : i : \mathbf{Out}(Init(A))$



$init : A : i : \mathbf{In}(in\ x . out\, new(z)\ \{x, Key(z)\}_{Key(A,S)}; n)$

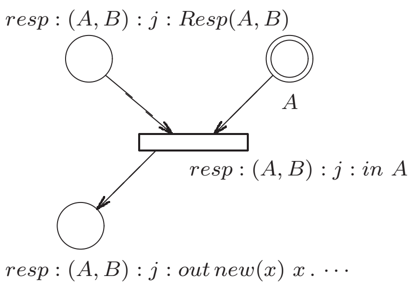

$init : A : i : in\ x . \cdots$
n
$init : A : i : in\ n$
$init : A : i : out\, new(z)\ \{n, Key(z)\}_{Key(A,S)}$

$init : A : i : \mathbf{Out}(out\, new(z)\ \{n, Key(z)\}_{Key(A,S)}; ab)$

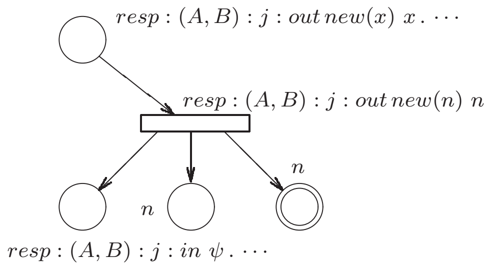

$init : A : i : out\, new(z)\ \{n, Key(z)\}_{Key(A,S)}$
$init : A : i : out\, new(ab)\ \{n, Key(ab)\}_{Key(A,S)}$
$\{n, Key(ab)\}_{Key(A,S)}$
ab

Responder events for every agent names $A, B \in \mathcal{A}$, names n, messages M, keys k and indices $j \in \omega$.

$resp : (A, B) : j : \mathbf{In}(Resp(A, B))$

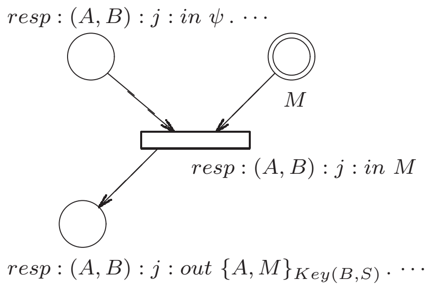

$resp : (A, B) : j : Resp(A, B)$
A
$resp : (A, B) : j : in\ A$
$resp : (A, B) : j : out\, new(x)\ x . \cdots$

$resp : (A,B) : j : \mathbf{Out}(out\, new(x)\ x\,.\,\cdots ; n)$



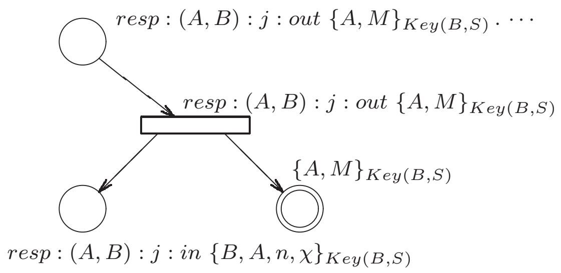
$resp : (A,B) : j : out\, new(x)\ x\,.\,\cdots$
$resp : (A,B) : j : out\, new(n)\ n$
n
n
$resp : (A,B) : j : in\ \psi\,.\,\cdots$

$resp : (A,B) : j : \mathbf{In}(in\ \psi\,.\,\cdots ; M)$



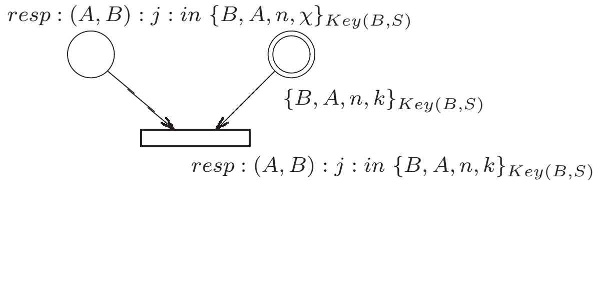
$resp : (A,B) : j : in\ \psi\,.\,\cdots$
M
$resp : (A,B) : j : in\ M$
$resp : (A,B) : j : out\ \{A,M\}_{Key(B,S)}\,.\,\cdots$

$resp : (A,B) : j : \mathbf{Out}(out\ \{A,M\}_{Key(B,S)}\,.\,\cdots)$



$resp : (A,B) : j : out\ \{A,M\}_{Key(B,S)}\,.\,\cdots$
$resp : (A,B) : j : out\ \{A,M\}_{Key(B,S)}$
$\{A,M\}_{Key(B,S)}$
$resp : (A,B) : j : in\ \{B,A,n,\chi\}_{Key(B,S)}$

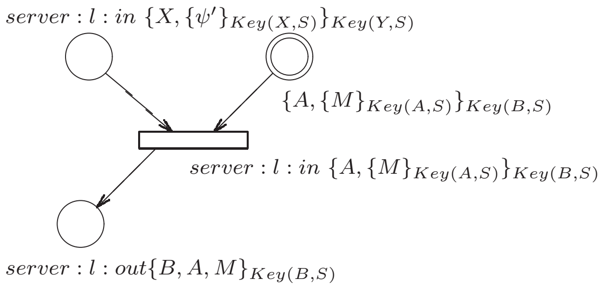
$resp : (A,B) : j : \mathbf{In}(in\ \{B,A,n,\chi\}_{Key(B,S)}; k)$



$resp : (A,B) : j : in\ \{B,A,n,\chi\}_{Key(B,S)}$
$\{B,A,n,k\}_{Key(B,S)}$
$resp : (A,B) : j : in\ \{B,A,n,k\}_{Key(B,S)}$

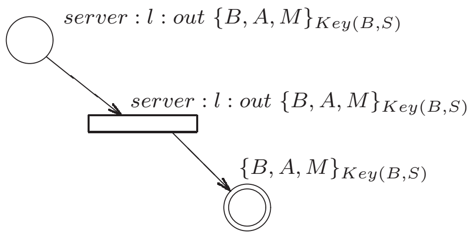
Server events for every agent names $A, B \in \mathcal{A}$, messages M and indices $l \in \omega$.

$server : l : \mathbf{In}(in\ \{X, \{\psi'\}_{Key(X,S)}\}_{Key(Y,S)} \cdot \cdots ; A, B, M)$



$server : l : in\ \{X, \{\psi'\}_{Key(X,S)}\}_{Key(Y,S)}$
$\{A, \{M\}_{Key(A,S)}\}_{Key(B,S)}$
$server : l : in\ \{A, \{M\}_{Key(A,S)}\}_{Key(B,S)}$
$server : l : out\{B, A, M\}_{Key(B,S)}$

$server : l : \mathbf{Out}(out\ \{B, A, M\}_{Key(B,S)})$



$server : l : out\ \{B, A, M\}_{Key(B,S)}$
$server : l : out\ \{B, A, M\}_{Key(B,S)}$
$\{B, A, M\}_{Key(B,S)}$

6.2.4 Security properties

We study secrecy and authentication properties for the Π_3 protocol. We are interested in proper runs of the kind

$$\langle \Pi_3, s_0, t_0 \rangle \xrightarrow{e_1} \cdots \xrightarrow{e_w} \langle p_w, s_w, t_w \rangle \xrightarrow{e_{w+1}} \ldots$$

where t_0 is a network state, the set of messages on the network in which a protocol execution starts. Whenever we refer to a run of Π_3 in the following section we mean a run of this kind.

Secrecy of long-term keys and session keys. The Π_3 protocol should not leak the long-term keys shared among the agents and the server.

Theorem 6.2.1 *Let $A_0 \in \mathcal{A}$. Given a run of Π_3, if $Key(A_0, S) \not\sqsubseteq t_0$ then at any stage w in the run $Key(A_0, S) \not\sqsubseteq t_w$.*

Proof. Suppose there is a run of Π_3 in which $Key(A_0, S)$ appears on the network. Since $Key(A_0, S) \not\sqsubseteq t_0$ there is a stage $v > 0$ in the run such that

$$Key(A_0, S) \not\sqsubseteq t_{v-1} \quad \text{and} \quad Key(A_0, S) \sqsubseteq t_v \ .$$

From the token game of nets with persistent conditions it follows that the event $e_v \in Ev(\Pi_3)$ is such that

$$Key(A_0, S) \sqsubseteq e_v^o \ .$$

The shape of initiator events $e \in init : Ev(p_{init})$ is such that $Key(A_0, S) \not\sqsubseteq e^o$. Even if messages sent by the initiator contain a key, the key is of the form $Key(ab)$ where ab is a name. Therefore $Key(ab)$ is syntactically different to $Key(A_0, S)$.

If $e_v \in resp : Ev(p_{resp})$ then it carries an action

$$act(e_v) = resp : (B, A) : i : out\ \{A, M\}_{Key(B,S)}$$

where $A, B, S \in \mathcal{A}$, i an index, and M is a message such that

$$Key(A_0, S) \sqsubseteq M\ .$$

By control precedence there exists an event e_u such that

$$act(e_u) = resp : (B, A) : i : in\ M$$

and

$$\begin{tikzcd} e_u \arrow[d] \\ e_v\ . \end{tikzcd}$$

By the token game $Key(A_0, S) \sqsubseteq t_j$, which is a contraddiction since $u < v$.

Similarly to the responder case, if $e_v \in server : Ev(p_{server})$ then one reaches a contraddiction by control precedence and the token game.

The event e_v can't be a spy event. This can easily be checked using control precedence and the token game in a similar way as done in the proof of secrecy for NSL private keys. □

It is desriable that the session keys distributed by the initiator remain secret. In our setting secrecy of session keys is ensured by the following theorem.

Theorem 6.2.2 *If a run of Π_3 contains an initiator event a_3 with action*

$$act(a_3) = init : A_0 : i_0 : out\,new(ab_0)\ \{B_0, Key(ab_0)\}_{Key(A_0,S)}$$

and if $Key(A_0, S), Key(B_0, S) \not\sqsubseteq t_0$ then at every stage w in the run $Key(ab_0) \notin t_w$.

Proof. We show a stronger property. We show that the key $Key(ab_0)$ never appears on the network in different surroundings than the ones prepared by the initiator and the server. Consider the property on configurations $\langle p, s, t\rangle$

$$Q(p, s, t) \Leftrightarrow$$
$$\sigma(Key(ab_0), t) \subseteq \{\{B_0, Key(ab_0)\}_{Key(A_0,S)}, \{B_0, A_0, Key(ab_0)\}_{Key(B_0,S)}\}\ .$$

If a run contains the event a_3 and one can show that at every stage w in the run $Q(p_w, s_w, t_w)$ then $Key(ab_0) \notin t_w$ for every stage w in that run. Suppose the contrary. Suppose that at some stage in the run the property Q does not hold. Let v, by well-foundedness, be the first stage in the run such that $\neg Q(p_v, s_v, t_v)$. Since $Fresh(ab_0, a_3)$ it follows from the Freshness Principle 5.1.3 that

$$\begin{tikzcd} a_3 \arrow[r] & e_v \end{tikzcd}$$

and $a_3 \neq e_v$. Clearly $e_v \in Ev(\Pi_3)$ and from the token game of nets with persistent conditions it follows that

$$\sigma(Key(ab_0), e_v^o) \not\sqsubseteq \{\{B_0, Key(ab_0)\}_{Key(A_0,S)}, \{B_0, A_0, Key(ab_0)\}_{Key(B_0,S)}\} \ .$$

The event e_v can only be an output event since $e^o = \emptyset$ for all input events e. We examine the possible output events of $Ev(\Pi_3)$.
Initiator output events. If $e_v \in server : Ev(p_{init})$ and $\sigma(Key(ab_0), e_v^o) \neq \emptyset$, then $Fresh(ab_0, e_v)$. By Freshness it follows that $e_v = a_3$ which is a contraddiction.
Responder output events. If $e_v \in resp : Ev(p_{resp})$ then it carries an action

$$act(e_v) = resp : (B, A) : i : out \ \{A, M\}_{Key(B,S)}$$

where $A, B \in \mathcal{A}$, i an index, and M a message such that

$$\sigma(Key(ab_0), M) \not\sqsubseteq \{\{B_0, Key(ab_0)\}_{Key(A_0,S)}, \{B_0, A_0, Key(ab_0)\}_{Key(B_0,S)}\} \ .$$

By Control precedence there exists an event e_u such that

$$act(e_u) = resp : (B, A) : i : in \ M$$

and

$$\begin{tikzcd} & e_u \arrow[d] \\ a_3 \arrow[r] & e_v \ . \end{tikzcd}$$

By the token game

$$\sigma(Key(ab_0), t_u) \not\sqsubseteq \{\{B_0, Key(ab_0)\}_{Key(A_0,S)}, \{B_0, A_0, Key(ab_0)\}_{Key(B_0,S)}\}$$

which is a contraddiction since $u < v$.
Server output events. Similarly to the responder case, if $e_v \in server : Ev(p_{server})$ then one reaches a contraddiction by control precedence and the token game.
Spy output events. Messages are persistent and the occurence of the event e_v follows that of the event a_3 in the run, therefore $\sigma(Key(ab_0), t_{v-1}) \neq \emptyset$. Theorem 5.2.4 applies and so $e_v \notin spy : Ev(p_{spy})$. □

Authentication The rather informal authentication requirement for Π_3 can easily be made precise in the Petri-net model. The property should hold for each run of the protocol. We are particularly concerned about those runs in which the responder completes a protocol round.

The informal correctness requirement we recalled above can be seen as an agreement property in the style suggested by Lowe (see [47]): To a completed responder round of B done apparently with A as initiator, should correspond in that protocol run a completed initiator round of A done apparently with B as responder. The messages that are exchanged should agree on their values. Formally, one can prove the following correctness property for the Π_3 protocol:

Theorem 6.2.3 *If a run of Π_3 contains the responder event b_5 with action*

$$act(b_5) = resp : (B_0, A_0) : i_0 : in\ \{B_0, A_0, n_0, k_0\}_{Key(B_0,S)}$$

and if $Key(A_0, S), Key(B_0, S) \not\sqsubseteq t_0$ then the run contains the initiator event a_3 with action

$$act(a_3) = init : A_0 : j : out\, new(ab_0)\ \{n_0, Key(ab_0)\}_{Key(A_0,S)})$$

with j an index and such that $a_3 \longrightarrow b_5$ and $k_0 = Key(ab_0)$. □

Proof. From control precedence it follows that

$$\begin{tikzcd} b_2 \arrow[d] \\ b_5 \end{tikzcd}$$

where b_2 is the responder event with action

$$act(b_2) = resp : (B_0, A_0) : i_0 : out\, new(n_0)\ n_0$$

and such that $Fresh(n_0, b_2)$. Consider the property on configurations

$$Q(p, s, t) \Leftrightarrow \{B_0, A_0, n_0, k_0\}_{Key(B_0,S)} \notin \sigma(n_0, t)\ .$$

By freshness the property Q holds immediately after b_2 but not immediately before b_5. By well-foundedness there is an earliest stage v in the run such that $\neg Q(p_v, s_v, t_v)$. The event e_v is an output event such that

$$\{B_0, A_0, n_0, k_0\}_{Key(B_0,S)} \in \sigma(n_0, e_v^o)$$

and it follows b_2 but it precedes b_5 in the run.

$$\begin{tikzcd} b_2 \arrow[d] \arrow[dr] & \\ b_5 & e_v \arrow[l] \end{tikzcd}\ .$$

We inspect the output events of Π_3 to determine which event e_v is.
Spy output events. Theorem 5.2.5 applies and so $e_v \notin spy : Ev(p_{spy})$.
Initiator output events. If $e_v \in init : Ev(p_{init})$ then

$$\{A_0, B_0, n_0, k_0\}_{Key(B_0,S)} \notin \sigma(n_0, e_v^o)\quad .$$

Responder output events. If $e_v \in resp : Ev(p_{resp})$ then

$$act(e_v) = resp : (B, A) : j : out\ \{A, M\}_{Key(B,S)}$$

and $\{A_0, B_0, n_0, k_0\}_{Key(B_0,S)} \in \sigma(n_0, M)$. By control precedence there exists an event e_u with action

$$act(e_u) = resp : (B, A) : j : in\ M$$

where $A, B \in \mathcal{A}$, j and index and M a message. By the token game

$$\{A_0, B_0, n_0, k_0\}_{Key(B_0,S)} \in \sigma(n_0, t_{u-1})$$

which is a contradiction. Observe that the case with $A = A_0$ and with $M = B_0, n_0, k_0$ is excluded since $Resp(B_0, B_0)$ is not included in the Π_3 system.
Server output events are the only possible events that remain to be considered. If $e_v \in server : Ev(p_{sever})$ then $e_v = s_2$ the server event with action

$$act(s_2) = server : l : out\ \{A_0, B_0, n_0, k_0\}_{Key(B_0,S)}$$

where l is an index.

Therefore

$$\begin{tikzcd} b_2 \arrow[d] \arrow[dr] & \\ b_5 & s_2 \arrow[l] \end{tikzcd}$$

and by control precedence and freshness

$$\begin{tikzcd} b_2 \arrow[d] \arrow[r] & s_1 \arrow[d] \\ b_5 & s_2 \arrow[l] \end{tikzcd}$$

where s_1 is the server event carrying action

$$act(s_1) = server : l : in\ \{A_0, \{n_0, k_0\}_{Key(A_0,S)}\}_{Key(B_0,S)}\ .$$

Consider the property on configurations

$$Q'(p, s, t) \Leftrightarrow \{A_0, \{n_0, k_0\}_{Key(A_0,S)}\}_{Key(B_0,S)} \notin \sigma(n_0, t)\ .$$

By freshness the property Q' holds immediately after b_2 but not immediately before s_1. By well-foundedness there is an earliest stage u in the run such that $\neg Q(p_u, s_u, t_u)$. The event e_u is an output event such that

$$\{A_0, \{n_0, k_0\}_{Key(A_0,S)}\}_{Key(B_0,S)} \in \sigma(n_0, e_u^o)$$

and it follows b_2 but it precedes s_1 in the run:

$$\begin{tikzcd} b_2 \arrow[dd] \arrow[dr] & & \\ & e_u \arrow[r] & s_1 \arrow[d] \\ b_5 & & s_2 \arrow[ll] \end{tikzcd}$$

In a similar way as we did before, we inspect the output events of Π_3 and determine that the event e_u is the responder event b'_4 with action

$$act(b'_4) = resp : (B_0, A_0) : h : out\ \{A_0, \{n_0, k_0\}_{Key(A_0,S)}\}_{Key(B_0,S)}$$

for some index h. By control precedence and freshness

$$\begin{tikzcd}
b_2 \arrow[dr] \arrow[ddd] & & \\
& b'_3 \arrow[d] & \\
& b'_4 \arrow[r] & s_1 \arrow[d] \\
b_5 & & s_2 \arrow[ll]
\end{tikzcd}$$

where b'_3 is the responder event with action

$$act(b'_3) = resp : (B_0, A_0) : h : in\ \{n_0, k_0\}_{Key(A_0,S)}\ .$$

Consider the property on configurations

$$Q''(p, s, t) \Leftrightarrow \{n_0, k_0\}_{Key(A_0,S)} \notin \sigma(n_0, t)\ .$$

By freshness the property Q'' holds immediately after b_2 but not immediately before b'_3. By well-foundedness there is an earliest stage w in the run such that $\neg Q(p_w, s_w, t_w)$. The event e_w is an output event such that

$$\{n_0, k_0\}_{Key(A_0,S)} \in \sigma(n_0, e^o_w)$$

and it follows b_2 but it precedes b'_3 in the run. Finally, the shape of all possible events of the system determines e_w to be the initiator event a_3 carrying action

$$act(a_3) = init : A_0 : j : out\, new(ab_0)\ \{n_0, k_0\}_{Key(A_0,S)}$$

where j is an index and where $k_0 = Key(ab_0)$.

$$\begin{tikzcd}
b_2 \arrow[r] \arrow[drr] \arrow[ddd] & a_3 \arrow[dr] & & \\
& & b'_3 \arrow[d] & \\
& & b'_4 \arrow[r] & s_1 \arrow[d] \\
b_5 & & & s_2 \arrow[lll]
\end{tikzcd}$$

□

Even if the statement of the authentication theorem above starts from the final event of the responder and asks for the final event of the initiator, it fits the shape of the agreement property that we discussed earlier. The complete responder and initiator rounds can be reconstructed from those two events:

- Whenever a sequence of configurations and events obtained from the net of Π_3 contains an event corresponding to the last action of the responder one can recognise that the responder indeed completed the round and who was claimed as initiator in the first step of the protocol. From the dependency among responder events in a protocol run it follows that if a run of the protocol contains the responder event b_5 with action

$$act(b_5) = resp : (B_0, A_0) : i_0 : in\ \{A_0, B_0, n_0, k_0\}_{Key(B_0,S)}$$

it also contains the responder event b_1 with action

$$act(b_1) = resp : (B_0, A_0) : i_0 : in\ A_0$$

and such that

$$\begin{tikzcd} b_1 \arrow[d] \\ b_5 \end{tikzcd}$$

Therefore B_0 believes that he responded to a request initiated by A_0.

- As before, whenever a run contains an initiator a_3 with action

$$act(a_3) = init : A_0 : j : out\, new(ab_0)\ \{n_0, Key(ab_0)\}_{Key(A_0,S)})$$

one can construct from the event dependencies the complete initiator round obtaining

$$\begin{tikzcd} a_1 \arrow[d] \\ a_2 \arrow[d] \\ a_3 \end{tikzcd}$$

where the events a_1 and a_2 have actions

$$act(a_1) = init : A_0 : j : out\ A_0$$
$$act(a_2) = init : A_0 : j : in\ n_0\ .$$

Therefore in that round A_0 believes that it was engaged in the protocol together with agent B_0 as responder.

Key compromise. As for the *ISO* protocol, key compromise does not affect more than one session. We do not report the details in this case but only remark that to model a session key getting corrupted one would extend the capabilities of the spy as done for the *ISO* protocol and add one more step in the protocol to represent a session, after key exchange, where initiator and responder use the exchanged key. For example

$$(6) \quad A \to B : \quad \{A\}_{Key(ab)} \quad .$$

Observation. As we recalled earlier, the Π_3 protocol that we studied is a slight modification of the original protocol and includes the name of the responder in the last message exchanged in a protocol round. This modification prevents the following attack which would go through if we didn't include the name of the responder in the last message but considered

$$(5) \quad S \to B: \quad \{A, n, Key(ab)\}_{K(B,S)}$$

instead:

$$\begin{tikzcd}
a_1 \arrow[r, "A"] & b_1 \arrow[d] & \\
& b_2 \arrow[r, "n"'] \arrow[d] & < spy > \arrow[dl, "{n,k}"] \\
& b_3 \arrow[d] & \\
& b_4 \arrow[d, "{\{A,n,k\}_{Key(B,S)}}"'] & \\
& b_5 & .
\end{tikzcd}$$

Chapter 7

Composing Strand Spaces

The last few years have seen the emergence of successful intensional, event-based, approaches to reasoning about security protocols. As is the case for the more explicit event-based model illustrated in the previous chapters, these more intentional methods are concerned with reasoning about the events that a security protocol can perform, and make use of a causal dependency that exists between events. The method of strand spaces [90, 88, 84] has been designed to support such an intensional, event-based, style of reasoning and has successfully been applied to a broad number of security protocols. However the strand spaces model of a protocol has some apparent limitations: Strand spaces don't compose readily, not using traditional process operations at least. Their form doesn't allow prefixing by a single event. Nondeterminism only arises through the choice as to where input comes from, and there is not a recognisable nondeterministic sum of strand spaces. Even an easy definition of parallel composition by juxtaposition is thwarted if "unique origination" is handled as a global condition on the entire strand space. That strand spaces are able to tackle a broad class of security protocols may therefore seem surprising. A reason for the adequacy of strand spaces lies in the fact that they can sidestep conflict if there are enough replicated strands available, which is the case for a broad range of security protocols.

This chapter has four main objectives. Firstly it extends the strand-space formalism to allow several operations on strand spaces to be defined. The operations form a strand-space language. Secondly the wide applicability of strand spaces to numerous security protocols and properties is backed up formally by showing that under reasonable conditions the extended model reduces to the original strand-space model. This result underpins the relation between nets and strand spaces discussed in the next chapter. Thirdly we address another issue of compositionality. Because we are only interested in safety properties we can make do with languages of strand-space bundles as models of process behaviour. We show how to compose such languages so that they may be used directly in giving the semantics of security protocols. Strand spaces that have substantially the same bundles can be regarded as equivalent and are congruent if they exhibit substantially the same open bundles. This congruence lays the ground for equational reasoning between strand spaces.

7.1 Some limitations of strand spaces

We discuss why strand spaces are hard to compose. Consider for example the ISO symmetric key two-pass unilateral authentication protocol (see [19]):

$$\begin{aligned} &A \to B : n \\ &B \to A : \{n, A\}_k \quad . \end{aligned}$$

Agents can engage in a protocol exchange under two different roles. The initiator, here A and the responder, here B. In a protocol round the initiator A chooses a fresh name n and sends it to the responder B. After getting the value, B encrypts it together with the initiator's identifier using a common shared key k. After getting the answer to her challenge, A can decrypt using the shared key and check whether the value sent matches the value received. In that case A can conclude that B is in fact operational.

$$\begin{tikzcd}
\mathit{out}\, n_0 \arrow[r] \arrow[d, Rightarrow] & \mathit{in}\, n_0 \arrow[d, Rightarrow] & \ldots & \mathit{out}\, n_i \arrow[r] \arrow[d, Rightarrow] & \mathit{in}\, n_i \arrow[d, Rightarrow] & \ldots \\
\mathit{in}\, \{n_0, A_0\}_k & \mathit{out}\, \{n_0, A_0\}_k \arrow[l] & & \mathit{in}\, \{n_i, A_0\}_k & \mathit{out}\, \{n_i, A_0\}_k \arrow[l] &
\end{tikzcd}$$

Figure 7.1: ISO protocol

The strand-space graph in Figure 7.1 describes the simple case of only two agents, A_0 and B_0, acting as initiator and responder respectively. For simplicity the graph has been drawn using the actions labelling the events in place of the events themselves. In this simple case the strand-space graph of the ISO protocol itself forms a bundle.

7.1.1 Unique origination

In the strand-space example above, all names n_i are uniquely originating – for each n_i there is only one strand in which the first action containing n_i is an output action. Unique origination intends to describe a name as fresh, perhaps chosen at random, and under the assumptions of Dolev and Yao [28], unguessable. For a construction of parallel composition of strand spaces it is therefore reasonable to require that names uniquely originating on components remain so on the composed strand space. Simple juxtaposition of strand spaces does not ensure this. For example consider a strand space for the ISO protocol which allows both agents A_0 and B_0 to engage in the protocol in any of the two possible roles. In Figure 7.2 the strand space formed out of two copies of the one in Figure 7.1. Figure 7.3 shows a possible bundle on such strand space. It describes a protocol run with two complete rounds. One in which A_0 is initiator and B_0 responder and another where the roles are inverted. Though the name n_0 is no longer uniquely originating on that strand space. A name's freshness is with respect to a run of a protocol more than to the whole set of possible executions. A notion of unique origination "on the bundle" seems more appropriate.

$$\begin{tikzcd}
\mathit{out}\, n_0 \arrow[r] \arrow[d, Rightarrow] \arrow[ddr] & \mathit{in}\, n_0 \arrow[d, Rightarrow] & \ldots & \mathit{out}\, n_i \arrow[r] \arrow[d, Rightarrow] \arrow[ddr] & \mathit{in}\, n_i \arrow[d, Rightarrow] \\
\mathit{in}\, \{n_0, A_0\}_k & \mathit{out}\, \{n_0, A_0\}_k \arrow[l] & & \mathit{in}\, \{n_i, A_0\}_k & \mathit{out}\, \{n_i, A_0\}_k \arrow[l] \\
\mathit{out}\, n_0 \arrow[r] \arrow[d, Rightarrow] \arrow[uur] & \mathit{in}\, n_0 \arrow[d, Rightarrow] & & \mathit{out}\, n_i \arrow[r] \arrow[d, Rightarrow] \arrow[uur] & \mathit{in}\, n_i \arrow[d, Rightarrow] \\
\mathit{in}\, \{n_0, B_0\}_k & \mathit{out}\, \{n_0, B_0\}_k \arrow[l] & & \mathit{in}\, \{n_i, B_0\}_k & \mathit{out}\, \{n_i, B_0\}_k \arrow[l]
\end{tikzcd}$$

Figure 7.2: ISO protocol - symmetric roles

$$\begin{tikzcd}
\mathit{out}\, n_0 \arrow[r] \arrow[d, Rightarrow] \arrow[ddr] & \mathit{in}\, n_0 \arrow[d, Rightarrow] \\
\mathit{in}\, \{n_0, A_0\}_k & \mathit{out}\, \{n_0, A_0\}_k \arrow[l] \\
\mathit{out}\, n_0 \arrow[d, Rightarrow] & \mathit{in}\, n_0 \arrow[d, Rightarrow] \\
\mathit{in}\, \{n_0, B_0\}_k & \mathit{out}\, \{n_0, B_0\}_k \arrow[l]
\end{tikzcd}$$

Figure 7.3: A possible bundle

7.1.2 Nondeterminism and strand-sequentiality

Nondeterminism in strand spaces arises only through the choice in a bundle of where input comes from. There is no recognisable way of modelling situations in which bundles may be taken either only over one strand space or over another. Juxtaposing strands as we did for example in Figure 7.2 allows bundles to include events of both components as is the case for the bundle in Figure 7.3.

One seems to encounter even more difficulties in the attempt to define a construction of prefixing a strand space with an action. Strands can't branch to parallel sub-strands and prefixing each strand of the space with an action would cause as many repetitions of that action as there are strands participating in a bundle. So for example if one prefixed each strand of the strand space of Figure 7.1 with the action $\mathit{out}\, m$ then bundles like the following are permitted:

$$\begin{tikzcd}
\mathit{out}\, m \arrow[d, Rightarrow] & \mathit{out}\, m \arrow[d, Rightarrow] \\
\mathit{out}\, n_0 \arrow[r] & \mathit{in}\, n_0 \arrow[d, Rightarrow] \\
& \mathit{out}\, \{n_0, A_0\}_k \quad .
\end{tikzcd}$$

Another attempt to define the prefixing operation would be to consider a separate strand containing only the prefixing action. However this does not force actions of the strands that are meant to be prefixed by that action to causally depend on that action.

7.2 An extension of strand spaces

In this section we extend the definition of strand space introducing a notion of conflict which we adapt from event structures (see e.g.[96]). We differ from the original definition of strand spaces in the treatment of unique origination which is taken care of in the definition of bundle rather than being a condition on the entire strand space – the "parametric strand spaces" of [17] achieve a similar effect as to unique origination and are related.

7.2.1 Strand spaces with conflict

The strands of a strand space consist of sequences of output and input actions. Actions are

$$Act = \{out\,new\,\vec{n}\,M \mid M \text{ msg}, \ \vec{n} \text{ distinct names}\} \cup \{in\,M \mid M \text{ msg}\}.$$

In $out\,new\,\vec{n}\,M$, the list $\vec{n}$ contains distinct names that are intended to be fresh ("uniquely originating") at the event.

Definition 7.2.1 A strand space with conflict $(\langle s_i\rangle_{i\in I}, \#)$ consists of:

(i) $\langle s_i\rangle_{i\in I}$ an indexed set of strands. An individual stand s_i, where $i \in I$, is a finite sequence of output or input actions in Act.

(ii) $\# \subseteq I \times I$ a symmetric, irreflexive binary conflict relation on strand indices.

Strand spaces with an empty conflict relation correspond to those of the standard definition of [90]. We denote by ϵ the empty strand space with no strands and with an empty conflict relation. [1] The empty strand space is different to a strand space $(\langle\lambda\rangle_{i\in I}, \#)$ where each strand is the empty sequence of actions λ. We write $|s|$ for the length of the sequence s.

7.2.2 The graph of a strand space

For a strand space $(\langle s_i\rangle_{i\in I}, \#)$ define the *strand-space graph* $(E, \Rightarrow, \rightarrow, act)$ associated with it as usual (see [90]). It is the graph with nodes (events)

$$E = \{(i,l) \mid i \in I \ , 1 \leq l \leq |s_i|\} \quad ,$$

actions labelling events $act(i,h) = s_i[h]$ and two edge relations. The first expresses precedence among events on the same strand,

$$(i,k) \Rightarrow (i,k+1) \text{ iff } (i,k),(i,k+1) \in E \quad ,$$

[1] We won't make much use of this particular strand space; it is however the identity for the operations of parallel composition and nondeterministic sum of strand spaces.

and the second expresses all possible communication,

$$(i,l) \rightarrow (j,h) \text{ iff } act(i,l) = out\, new\, \vec{n}\, M \text{ and } act(j,h) = in\, M \quad .$$

An event is an *input event* if its action is an input action and an event is an *output event* if its action is an output. The names $names(e)$ of an event e are all the names appearing on the action associated with e – the ones that are marked as “new”, denoted by $new(e)$, together with those in the message of the action.

7.2.3 Bundles

Bundles of a strand space describe runs in a computation.

Definition 7.2.2 A bundle b of a strand space with conflict $\#$ is a finite subgraph $(E_b, \Rightarrow_b, \rightarrow_b, act_b)$ of the strand-space graph $(E, \Rightarrow, \rightarrow, act)$ such that:

(i) if $e \Rightarrow e'$ and $e' \in E_b$ then $e \Rightarrow_b e'$, *(control precedence)*

(ii) if $e \in E_b$ and $act_b(e) = in\, M$ then there exists a unique $e' \in E_b$ such that $e' \rightarrow_b e$, *(output-input precedence)*

(iii) if $e, e' \in E_b$ such that $act_b(e) = out\, new\, \vec{n}\, M$ and $n \in \vec{n} \cap names(e')$ then either $e \Rightarrow_b^* e'$ or there exists an input event e'' such that $n \in names(e'')$ and $e'' \Rightarrow_b^* e'$, *(freshness)*

(iv) if $(i,h), (j,k) \in E_b$ then $\neg(i \# j)$, *(conflict freeness)*

(v) the relation $\Rightarrow_b \cup \rightarrow_b$ is acyclic. *(acyclicity)*

The empty graph, denoted by λ, is a bundle. It will be clear from the context whether λ stands for the empty bundle or whether it denotes the empty sequence of actions. The empty strand space has only one bundle, the empty bundle.

Points (i), (ii), (v) of the definition of bundle for a strand space with conflict match the standard definition of [90]. Point (iii) ensures freshness of “new” values in a bundle. Point (iv) doesn't allow events from conflicting strands to appear in a bundle. Write $\leq_b$ for the reflexive and transitive closure of $\Rightarrow_b \cup \rightarrow_b$.

Proposition 7.2.3 *If b is a bundle then $\leq_b$ is a partial order on E_b.*

Proof. A bundle is an acyclic subgraph of the strand-space graph. □

The relation $\leq_b$ determines the partial order on events occurring in a computation described by b. Names introduced as “new” don't appear on events preceding their introduction and are never introduced as “new” more than once.

Proposition 7.2.4 *Let b be a bundle of a strand space and let $e, e' \in E_b$ bundle events such that $act_b(e) = out\, new\, \vec{n}\, M$. If $n \in \vec{n} \cap names(e')$ then $e \leq_b e'$ and if $act_b(e') = out\, new\, \vec{m}\, M'$ then $n \notin \vec{m}$.*

Proof. Suppose that $e \not\leq_b e'$ and therefore $e \not\Rightarrow^* e'$. There exists an input event $e'' \in b$ such that $n \in names(e'')$ and $e'' \Rightarrow_b^* e'$ (freshness). The bundle b is acyclic and each input event in b is preceded by a matching output event (output-input precedence). Therefore there exists an output event $e_1 \in E_b$ such that $n \in names(e_1)$ and such that for every input event e_2 if $e_2 \Rightarrow_b^* e_1$ then $n \notin names(e_2)$. The event e can't precede e_1 on the same strand ($e \not\Rightarrow_b^* e_1$), otherwise $e \leq_b e'$. The events e and e_1 are both in b thus contradicting the freshness property of b.

If $act_b(e') = out\, new\, \vec{m}\, M'$ and $n \in \vec{m}$ then $e \leq_b e'$ and $e' \leq_b e$, and therefore $e = e'$. □

There are other possible choices for the freshness condition (iii). A weaker condition could be the following:

> if $e, e' \in E_b$ are bundle events such that $act_b(e) = out\, new\, \vec{n}\, M$ and such that $n \in \vec{n} \cap names(e')$ then $e \leq_b e'$.

This condition however would allow bundles of the kind

$$\begin{tikzcd}
out\, new\, n\, M \arrow[d, Rightarrow] & in\, m \arrow[d, Rightarrow] \\
out\, m \arrow[ur] & out\, n
\end{tikzcd}$$

If the two strands are distinct processes, the second strand is not supposed to send the name n without receiving it first from somewhere. Graphs like that are not considered bundles in the original treatment of strand spaces [90] and are also excluded by the slightly more involved freshness condition of Definition 7.2.2.

7.2.4 Re-indexing of strand spaces

We regard two strand spaces as substantially the same if they differ only on the indices of their strands and therefore one strand space can be obtained from the other by a simple "re-indexing" operation. [2]

Definition 7.2.5 Given $(\langle s_i \rangle_{i \in I}, \#)$ and $(\langle t_j \rangle_{j \in J}, \#')$ two strand spaces write

$$(\langle s_i \rangle_{i \in I}, \#) \cong (\langle t_j \rangle_{j \in J}, \#')$$

if there exists a bijection $\pi : I \to J$ such that:

(i) $\forall i \in I \,.\, s_i = t_{\pi(i)}$ and

(ii) $\forall i, j \in I \,.\, i \# j \Leftrightarrow \pi(i) \#' \pi(j)$.

The relation $\cong$ is an *equivalence relation* on strand spaces. A bijection π which establishes such equivalence is called a *re-indexing of strand spaces*. Moreover take $(\langle s_i \rangle_{i \in I}, \#)$ a strand space, J a set, and $\pi : I \to J$ a bijection. We define the strand space $(\langle t_j \rangle_{j \in J}, \pi(\#))$ where

[2] If the indices carry structure (some might involve agent names for example) we might refine the permissible re-indexings.

- $\forall j \in J \,.\, t_j = s_{\pi^{-1}(j)}$ and
- $\forall j, j' \in J \,.\, j\, \pi(\#)\, j'$ iff $\pi^{-1}(j) \# \pi^{-1}(j')$.

The relation $\pi(\#)$ is irreflexive and symmetric and establishes the equivalence

$$(\langle s_{\pi(i)} \rangle_{i \in I}, \pi(\#)) \cong (\langle s_i \rangle_{i \in I}, \#) \quad .$$

Proposition 7.2.6 *Let $(\langle s_i \rangle_{i \in I}, \#)$ and $(\langle t_j \rangle_{j \in J}, \#')$ be two strand spaces such that $(\langle s_i \rangle_{i \in I}, \#) \cong (\langle t_j \rangle_{j \in J}, \#')$ for a bijection $\pi : I \to J$. Given b a bundle of $(\langle s_i \rangle_{i \in I}, \#)$ then $\pi(b)$ obtained from b by changing all strand indices according to π is a bundle of $(\langle t_j \rangle_{j \in J}, \#')$.*

Proof. The proposition follows from the assumption that b is a bundle and from the definition of re-indexing on bundles. □

7.3 Constructions on strand spaces

The extension of the strand-space formalism with a conflict relation and the different treatment of unique origination as illustrated in the previous section allow operations such as prefixing, parallel composition, and sum of strand spaces to be defined in terms of traditional process operations.

7.3.1 Prefixing

The operation of prefixing a strand space with an action is complicated by the strand-space formalism not permitting strands to branch. Only if the strand space to be prefixed is such that every two different strands are in conflict can each strand be prefixed with the action. Then the conflict relation disallows repetitions of the prefixing action in bundles. Given $\alpha \in \mathit{Act}$ and a strand space $(\langle s_i \rangle_{i \in I}, \#)$ such that for all $i, j \in I$ if $i \neq j$ then $i \# j$, define

$$\alpha.(\langle s_i \rangle_{i \in I}, \#) \overset{def}{=} (\langle \alpha s_i \rangle_{i \in I}, \#) \quad .$$

We understand the special case of prefixing the empty strand space with an action, to yield the empty strand space

$$\alpha.\epsilon = \epsilon \quad .$$

When prefixing a strand space consisting of only empty strands one obtains

$$\alpha.(\langle \lambda \rangle_{i \in I}, \#) = (\langle \alpha \rangle_{i \in I}, \#) \quad .$$

7.3.2 Parallel composition

The operation of parallel composition of two strand spaces is the disjoint union of their sets of strands and conflict relations. Disjoint union is achieved by tagging the first space with index 0 and the second with index 1. Given strand spaces $(\langle s_i^0 \rangle_{i \in I}, \#^0)$ and $(\langle s_j^1 \rangle_{j \in J}, \#^1)$ define

$$(\langle s_i^0 \rangle_{i \in I}, \#^0) \mid\mid (\langle s_j^1 \rangle_{j \in J}, \#^1) \overset{def}{=} (\langle s_h \rangle_{h \in H}, \#)$$

where

$$H = (\{0\} \times I) \cup (\{1\} \times J), \quad s_{(0,i)} = s^0_i \quad \text{and} \quad s_{(1,i)} = s^1_i \ .$$

Two strands are in conflict only if they belong to the same component and are in conflict within that component. More precisely for k either 0 or 1 define

$$h \,\#\, h' \quad \text{iff} \quad h = (k,i),\ h' = (k,j) \text{ and } i \,\#^k\, j \ .$$

The relation $\#$ is irreflexive and symmetric.

We extend the definition of parallel composition to an operation indexed over a set. Given a collection of strand spaces $(\langle s^k_i\rangle_{i\in I_k}, \#^k)$ indexed by k in a set K, define

$$||_{k\in K}(\langle s^k_i\rangle_{i\in I_k}, \#^k) \overset{def}{=} (\langle s_h\rangle_{h\in H}, \#) \ ,$$

where

$$H = \sum_{k\in K} I_k, \quad s_{(k,i)} = s^k_i, \quad \text{and where} \quad (k,i) \,\#\, (k',i') \text{ iff } k = k' \text{ and } i \,\#^k\, i' \ .$$

In particular if $K = \emptyset$ then the parallel composition yields the empty strand space.

As a special case of parallel composition of strand spaces consider the strand space obtained by composing infinitely many but equal strand spaces. Abbreviate

$$||_{k\in\omega}(\langle s_i\rangle_{i\in I}, \#) \overset{def}{=} !\,(\langle s_i\rangle_{i\in I}, \#) \ .$$

One easily observes that

$$!\,(\langle s_i\rangle_{i\in I}, \#) = (\langle s_{(n,i)}\rangle_{(n,i)\in\omega\times I}, !\#)$$

where $!\#$ is the binary relation over $\omega \times I$ such that

$$(n,i) \,!\#\, (m,i') \text{ iff } n = m \text{ and } i \,\#\, i' \ .$$

7.3.3 Nondeterministic sum

The nondeterministic sum and the parallel composition of strand spaces construct the same indexed set of strands. The conflict relation of a summed space, in addition to the existing conflicts, imposes conflict between every two strands that belong to different components. Given strand spaces $(\langle s^0_i\rangle_{i\in I}, \#^0)$ and $(\langle s^1_j\rangle_{j\in J}, \#^1)$ define

$$(\langle s^0_i\rangle_{i\in I}, \#^0) \ + \ (\langle s^1_j\rangle_{j\in J}, \#^1) \overset{def}{=} (\langle s_h\rangle_{h\in H}, \#)$$

where

$$H = (\{0\} \times I) \cup (\{1\} \times J), \quad s_{(0,i)} = s^0_i \quad \text{and} \quad s_{(1,i)} = s^1_i \ .$$

Two strands are in conflict only if they belong to different components or are already in conflict within a component. More precisely for k and k' either 0 or 1 define

$$(k,i) \,\#\, (k',j) \quad \text{iff} \quad k \neq k' \text{ or if } k = k' \text{ and } i \,\#^k\, j \ .$$

The relation $\#$ is irreflexive and symmetric.

The sum of strand spaces extends to an operation indexed over a set similarly to the parallel composition operation. Define

$$\sum_{k \in K} S_k \overset{def}{=} (\langle s_h \rangle_{h \in H}, \#)$$

where $H = \sum_{k \in K} I_k$, $s_{(k,i)} = s_i^k$, and where

$$(k,i) \# (k',i') \text{ iff either } k \neq k' \text{ or } (k = k' \text{ and } i \#^k i') \ .$$

7.3.4 Re-indexing and strand-space constructions

The operations that we definied on strand spaces, satisfy the following properties:

Proposition 7.3.1 *Let S_0 , S_1, and S_2 be strand spaces.*

1. $S_0 || \epsilon \cong S_0$
2. $S_0 || (S_1 || S_2) \cong (S_0 || S_1) || S_2$
3. $S_0 || S_1 \cong S_1 || S_0$

and similarly for $+$.

Proof. Let $S_0 = (\langle s_i^0 \rangle_{i \in I}, \#^0)$, $S_1 = (\langle s_j^1 \rangle_{j \in J}, \#^1)$, and $S_2 = (\langle s_k^2 \rangle_{k \in K}, \#^2)$.

1. The strand space $S_0 || \epsilon$ has the same strands as S_0, but indexing set $\{0\} \times I$. Let $\sigma : \{0\} \times I \to I$ be the projection to the second component, which is a bijection. One obtains an equivalence between the two strand spaces. Every $s_{(0,i)}$ in $S_0 || \epsilon$ is equal to s_i, therefore $s_{(0,i)} = s_{\sigma(0,i)}$. Let $\#$ be the conflict relation of $S || \epsilon$. If $(0,i) \# (0,i')$ then there is conflict $i \#^0 i'$, therefore $\sigma(0,i) \#^0 \sigma(0,i')$.

2. Both spaces have the same strands but the first has indexing set

$$H = (\{0\} \times I) \cup (\{1\} \times ((\{0\} \times J) \cup (\{1\} \times K)))$$

while the second has indexing set

$$H' = (\{0\} \times ((\{0\} \times I) \cup (\{1\} \times J))) \cup (\{1\} \times K) \ .$$

The following function $\sigma : H \to H'$

$$\sigma(h) = \begin{cases} (0,(0,i)) & \text{if } h = (0,i) \\ (0,(1,j)) & \text{if } h = (1,(0,j)) \\ (1,k) & \text{if } h = (1,(1,k)) \end{cases}$$

is a bijection and establishes the equivalence.

3. Follows easily.

□

The equivalence $S + S \cong S$ does not exist for every strand space S. For example consider the strand space S composed out of one single strand with index i. The indexing set of $S + S$ is $\{(0,i),(1,i)\}$. There is no bijection between a set of one element and a set of two elements.

Strands are not permitted to branch and the prefixing operation prefixes each single strand of the space. The following proposition holds:

Proposition 7.3.2 *Let S_0 and S_1 be strand spaces and α an action. Then*

$$\alpha.(S_0 + S_1) = \alpha.S_1 + \alpha.S_0 \quad .$$

Proof. Follows from the definition of the prefixing operation and sum of strand spaces. □

7.4 A process language for strand spaces

The constructions we have shown in the previous section form a language $\mathcal{S}$ of strand spaces with the following grammar:

$$S \ ::= \ L \mid \sum_{j \in J} S_j \mid ||_{j \in J} S_j$$

where $L \in \mathcal{L}$, the language of "sequential strand spaces" given by

$$L \ ::= \ \langle \lambda \rangle \mid \alpha.L \mid \sum_{j \in J} L_j \ .$$

The strand space $\langle \lambda \rangle$ has only one strand which is the empty sequence of actions and with the empty conflict relation.[3] The bundles of strand spaces in $\mathcal{L}$ form linearly ordered sets of events, and therefore can be thought of as runs of a sequential process.

A strand-space term of language $\mathcal{S}$ is a "par" process in the sense that parallel composition is only at the top level and therefore consists of a parallel composition and sum of sequential processes. Of particular interest are "!-par" processes which are those terms of $\mathcal{S}$ of the form $!S$. As shown in Section 7.7 conflict can be eliminated from such strand spaces.

7.5 Open bundles and open-bundle languages

The usual semantics of a strand space is in terms of its set of bundles. In this section we show how, by broadening to open bundles, the bundle space can be constructed in a compositional way from bundle spaces. As shown in Section 7.6 an interesting congruence relation between strand spaces is based on open bundles rather than bundles and the compositional account of the bundle space presented here is useful in showing that such relation is indeed a congruence.

7.5.1 Open bundles

It will be useful to weaken the definition of bundle to the one of *open bundle*, so that bundles can be composed. An open bundle is a graph with the same structure of a bundle, but where input events need not necessarily be related to output events. In

[3]Let the index of the empty strand in $\langle \lambda \rangle$ be a distinguished index $*$.

this sense the open bundle is "open" to the environment for communication on input events that are not already linked to output events. An open bundle needs to be such that it can always be transformed into a bundle of perhaps a larger strand space.

Definition 7.5.1 An open bundle b of a strand space with conflict $\#$ is a finite subgraph $(E_b, \Rightarrow_b, \rightarrow_b, act_b)$ of the strand-space graph $(E, \Rightarrow, \rightarrow, act)$ such that:

(i) if $e \Rightarrow_G e'$ and $e' \in E_b$ then $e \Rightarrow_b e'$, *(control precedence)*

(ii) if $e' \rightarrow_b e$ and $e'' \rightarrow_b e$ then $e' = e''$, *(output-input correspondence)*

(iii) if $e, e' \in E_b$ s.t. $act(e) = out\,new \vec{n} M$ and $n \in \vec{n} \cap names(e')$ then either $e \Rightarrow_b^* e'$ or there exists an input event $e'' \in E_b$ such that $n \in names(e'')$, and $e'' \Rightarrow_b^* e'$, *(freshness)*

(iv) if $(i, h), (j, k) \in E_b$ then $\neg(i \,\#\, j)$, *(conflict freeness)*

(v) the relation $\Rightarrow_b \cup \rightarrow_b \cup \hookrightarrow$ is acyclic, where $e \hookrightarrow e'$ if $new(e) \cap names(e') \neq \emptyset$ and if $e \neq e'$. *(acyclicity)*

In an open bundle "freshness" dependencies need not be caught through $\Rightarrow_b$ and $\rightarrow_b$. Point (v) takes account of additional freshness dependencies and excludes graphs like

$$\begin{tikzcd} in\,n \arrow[d, Rightarrow] & in\,m \arrow[d, Rightarrow] \\ out\,new\,m\;m \arrow[ur] & out\,new\,n\;n \end{tikzcd} .$$

7.5.2 Control graphs

Our constructions for open bundles take us through the intermediary of control graphs (which are similar to pomsets [65] and message sequence charts [41]).

Definition 7.5.2 A *control graph*, with indices I, is a graph $(E, \rightarrow, act)$ where

- $E \subseteq I \times \mathbb{N}$ such that if $(i, h) \in E$ and $h > 1$ then $(i, h-1) \in E$ (when we write $(i, h-1) \Rightarrow (i, h)$), and
- $\rightarrow \subseteq E \times E$, and $act : E \rightarrow Act$.

Denote by $ind(g)$ the set of indices of a control graph g. Control graphs can be ordered by inclusion, more precisely if g, g' are control graphs write $g \subseteq g'$ whenever $E_g \subseteq E_{g'}$, $\rightarrow_g \subseteq \rightarrow_{g'}$ and $act_g \subseteq act_{g'}$.

Strand-space graphs, bundles and open bundles are examples of control graphs. We say a control graph is an open bundle when it is finite and satisfies axioms (ii), (iii) and (v) of Definition 7.5.1 above. We say the control graph is an open bundle of a certain strand space if in addition it is subgraph of that strand-space graph and satisfies axiom (iv).

Prefixing extends a control graph with new initial control nodes $(i, 1)$ where i is an index. Every event in the original graph that has index i is shifted one position later in the control structure while keeping its causal dependencies. For i, j indices and $h \in I\!\!N$, define $(j, h)/i = (j, h+1)$ if $j = i$ and $(j, h)/i = (j, h)$ otherwise. For a set of indices I, $\alpha \in Act$ and g a control graph, define the control graph

$$(\alpha, I) \,.\, g = (E, \rightarrow, act)$$

where

- $E = \{(i, 1) \mid i \in I\} \cup \{e/i \mid e \in E_g \ \&\ i \in I\}$,
- $e/i \rightarrow e'/i$ whenever $e \rightarrow_g e'$, and
- take $act(i, 1) = \alpha$ and $act(e/i) = act_g(e)$ for every $e \in E_g$ and $i \in I$.

Deletion removes from a control graph the initial control nodes with certain component indices. When a node $(i, 1)$ is removed, every event in the original graph that has index i is shifted one position earlier in the control structure while keeping its causal dependencies. For i, j indices and $h \in I\!\!N$, define $(j, h)\backslash i = (j, h-1)$ if $j = i$ and $h > 1$ and define $(j, h)\backslash i = (j, h)$ otherwise. For a set of indices I, and g a control graph, define the control graph

$$g\backslash I = (E, \rightarrow, act)$$

where

- $E = \{e\backslash i \mid e \in E_g \setminus \{(i, 1) \mid i \in I\} \ \&\ i \in I\}$,
- $e\backslash i \rightarrow e'\backslash i$ whenever $e \rightarrow_g e'$, and
- $act(e\backslash i) = act_g(e)$ for every $e \in E_g \setminus \{(i, 1) \mid i \in I\}$ and $i \in I$.

Control graphs can be *composed* by tagging components to keep them disjoint and by juxtaposing them. For i, j indices and $h \in I\!\!N$, define $j : (i, h) = ((j, i), h)$. For a control graph g and index j define

$$j : g = (E, \rightarrow, act)$$

where $E = \{j : e \mid e \in E_g\}$, $j : e \rightarrow j : e'$ whenever $e \rightarrow_g e'$, and $act(j : e) = act_g(e)$. Let J be a set of indices and g_j control graphs for every $j \in J$. Define

$$\|_{j \in J} g_j = \bigcup_{j \in J} j : g_j \ .$$

Control graphs can be *restricted* to only those nodes that concern a particular component. For a control graph g and index j define

$$g_{|j} = (E, \rightarrow, act)$$

where $E = \{e \mid j : e \in E_g\}$, $e \rightarrow e'$ whenever $j : e \rightarrow_g j : e'$, and $act(e) = act_g(j : e)$.

7.5.3 From open bundles to bundles

The definition of open bundle ensures that b extends to a bundle over a (bigger) strand space. Let g, g' be control graphs. Define

$$g \preceq g' \quad \text{iff} \quad E_g = E_{g'}, \ \rightarrow_g \subseteq \rightarrow_{g'} \text{ , and } act_g = act_{g'} \ .$$

Write

$$g\!\uparrow = \ \{b \mid g \preceq b \text{ and } b \text{ an open bundle}\} \quad .$$

Proposition 7.5.3 *Let b, d be two control graphs such that $b \preceq d$. If d is an open bundle then b is an open bundle.*

Proof. Observe that if $b \preceq d$ then $\hookrightarrow_b = \hookrightarrow_d$ and therefore if $\Rightarrow_d \cup \rightarrow_d \cup \hookrightarrow_d$ is acyclic so is $\Rightarrow_b \cup \rightarrow_b \cup \hookrightarrow_b$. If d is an open bundle then it also satisfies (ii) and (iii) of the definition of open bundle and therefore so does b. □

Open bundles can always be extended to form a bundle as is shown by the following theorem. Its proof shows a way to construct a bundle from an open bundle. The crux of the construction is that of adding an output action corresponding to every open input action and a communication edge from the output event to the input event. Care has to be taken when names of open-input actions are introduced as "new" somewhere in the open bundle – the output action that is added in correspondence to the open input requires preceding input actions that bind the names in question so that open-bundle freshness is not violated. Such input actions are introduced when necessary. Additionally, our construction introduces a new name generation on a dummy output event which precedes all other events that are added. This more technical detail is not important here but is central later, in characterising the greatest congruence relation inside bundle equivalence (see proof of Theorem 7.6.5).

Theorem 7.5.4 *Let b be an open bundle of a strand space S. There exists a strand space T and an open bundle t of T such that there is $b' \in (t||b)\!\uparrow$ and b' is a bundle of $T||S$.*

Proof. Given an open bundle b we construct an open bundle t so that when composed with b, communication edges can be added to form a bundle of a bigger strand space. Let I be the set of open input actions in b. For every $i \in I$ let n_i be a name such that $n_i \notin names(b)$ and $n_i \neq n_j$ if $i \neq j$. Consider the following strand space

$$T = \langle s_i \rangle_{in\, M \in I}$$

such that

$$s_i = out\, new\, n_i \ D \ . \ in \ N_1 \ . \ \cdots \ . \ inN_l \ . \ outM$$

where D is a dummy message and $\{N_1, \ldots, N_l\}$ is the smallest, possibly empty, set of messages such that for every $e \in E_b$ if $act(e) = out\, new\, \vec{n} \ N$ and $\vec{n} \cap names(M) \neq \emptyset$ then $N \in \{N_1, \ldots, N_l\}$.

Let G_T be the strand-space graph of T and let t be an open bundle such that $t \preceq G_T$ (for example the one with no communication edges). Let b' be the graph obtained from $t \parallel b$ by connecting the open inputs of b to the corresponding output events in t and the input events of t to those output events that introduce new names which appear on the open inputs of b. By construction, $b' \in (t \mid\mid b)\uparrow$ and b' does not have any open inputs. Output-input correspondence (*ii*) is clearly respected by b'. Freshness (*iii*) holds for b and names marked as new in t do not appear in b. Suppose there is an event in b on which the new name n is chosen and suppose n appears on an output event in t. As required from (*iii*), by construction of b' there is an input event on which n appears and which precedes the output event. The relation $\rightarrow_{b'} \cup \Rightarrow_{b'}$ is acyclic (*v*). If it was not acyclic then the construction of b' would induce a cycle on $\Rightarrow_b \subseteq \rightarrow_b \cup \hookrightarrow_b$, which clearly is not possible. The graph b' is therefore a bundle of $T \mid\mid S$. □

7.5.4 Open-bundle languages

The language of open bundles of a strand-space term is defined as follows:

Definition 7.5.5 Let $L \in \mathcal{L}$ and $S \in \mathcal{S}$. Define

$$\begin{aligned}
\mathcal{O}(\langle\lambda\rangle) &= \{\lambda\}\\
\mathcal{O}(\alpha.L) &= \{\lambda\} \;\cup\; \{(\alpha,\{i\})\,.\,\lambda \mid i \in ind(G_L)\} \;\cup\\
&\qquad \bigcup\{(\alpha, ind(b))\,.\,b\uparrow \;\mid b \in \mathcal{O}(L)\setminus\{\lambda\}\}\\
\mathcal{O}(\sum_{j\in J} S_j) &= \{j : b \mid b \in \mathcal{O}(S_j)\}\\
\mathcal{O}(||_{j\in J} S_j) &= \bigcup\{(\bigcup_{i\in I} i : b_i)\uparrow \;\mid I \subseteq J \;\&\; I \text{ finite } \&\; \forall i \in I\,.\,b_i \in \mathcal{O}(S_i)\} \quad .
\end{aligned}$$

Theorem 7.5.6 *If S is a strand-space term in $\mathcal{S}$ then the elements of $\mathcal{O}(S)$ are exactly the open bundles of the strand space denoted by S.*

Proof. Let S be the strand-space term and G_S the strand-space graph of the strand space denoted by S. The proof has two parts.

The first part shows, by induction on the structure of strand space terms, that if b is an open bundle of the strand space denoted by S then $b \in \mathcal{O}(S)$.

Case $S = \langle\lambda\rangle$. The only open bundle of S is λ and clearly $\lambda \in \mathcal{O}(S)$.

Case $S = \alpha.L$. If $b = \lambda$ then $b \in \mathcal{O}(S)$ by definition. Let $b \neq \lambda$ be an open bundle of S and therefore $b \subseteq G_{\alpha.L}$. Clearly

$$b\backslash ind(G_L) \subseteq G_{\alpha.L}\backslash ind(G_L) = G_L \ .$$

The control graph $b\backslash ind(G_L)$ is an open bundle of L – it is a subgraph of the strand-space graph of L and conditions (ii), (iv), (v) are satisfied because they hold for b. Because of the "sequential" shape of b open bundle of $L \in \mathcal{L}$, deletion does not affect Freshness (iii). Since $b\backslash ind(G_L)$ is an open bundle of L, the induction hypothesis applies and therefore

$$b\backslash ind(G_L) \in \mathcal{O}(L) \ .$$

Observe that

$$b\backslash ind(G_L) = b\backslash ind(b) \quad \text{and} \quad act_b(ind(b), 1) = \alpha$$

and therefore
$$(\alpha, ind(b)).(b\backslash ind(b)) \preceq b \quad .$$
If $b\backslash ind(b) = \lambda$ then $b = (\alpha, ind(b)).\lambda \in \mathcal{O}(L)$. Suppose instead that $b\backslash ind(b) \neq \lambda$ then $ind(b\backslash ind(b)) = ind(b)$, since b is an open bundle of the sequential strand space L. It follows that
$$(\alpha, ind(b\backslash ind(b)).(b\backslash ind(b)) \preceq b \quad .$$
The control graph b is an open bundle, therefore
$$b \in (\alpha, ind(b\backslash ind(b)).(b\backslash ind(b)) \uparrow$$
and $b \in \mathcal{O}(\alpha.L)$.

Case $S = \sum_{j\in J} S_j$. Let b be an open bundle of S and therefore $b \subseteq G_{\sum_{j\in J} S_j}$. Clearly
$$b_{|j} \subseteq (G_{\sum_{j\in J} S_j})_{|j} = G_{S_j}$$
for every index $j \in J$. By conflict freeness there exists an index $i \in J$ such that $b = i : b_{|i}$. It follows that $b_{|i}$ is an open bundle of S_i. By the induction hypothesis $b_{|i} \in \mathcal{O}(S_i)$ and therefore $b \in \mathcal{O}(\sum_{j\in J} S_j)$.

Case $S = ||_{j\in J} S_j$. Let b be an open bundle of S and therefore $b \subseteq G_{\|_{j\in J} S_j}$. Clearly
$$b_{|j} \subseteq (G_{\|_{j\in J} S_j})_{|j} = G_{S_j}$$
for every $j \in J$. It is easy to check that the control graph $b_{|j}$ is an open bundle of S_j since b is an open bundle of S. By the induction hypothesis $b_{|j} \in \mathcal{O}(S_j)$ for every $j \in J$. Observe that
$$\bigcup_{j\in J} b_{|j} \preceq b \; .$$
Since b is an open bundle it follows that
$$b \in (\bigcup_{j\in J} b_{|j}) \uparrow$$
and therefore $b \in \mathcal{O}(||_{j\in J} S_j)$.

The second part shows inductively on the structure of strand-space terms that every $b \in \mathcal{O}(S)$ is an open bundle of the strand space denoted by S.

Case $S = \langle\lambda\rangle$. If $b \in \mathcal{O}(\langle\lambda\rangle)$ then $b = \lambda$. The empty graph λ is an open bundle of every strand space.

Case $S = \alpha.L$. Let $b \in \mathcal{O}(\alpha.L)$. If $b = \lambda$ then it is an open bundle of S. Let
$$b = (\alpha, \{i\}) \, . \, \lambda$$
with $i \in ind(G_L)$. Observe that
$$(\alpha, \{i\}) \, . \, \lambda \subseteq (\alpha, ind(L)) \, . \, G_L = G_{\alpha.L} \; .$$
and therefore b is an open bundle of $\alpha.L$. Suppose instead that
$$b \in ((\alpha, ind(b')) \, . \, b') \uparrow$$

for some $b' \in \mathcal{O}(L) \setminus \{\lambda\}$. Clearly b is an open bundle. We need to check that it is an open bundle of $\alpha.L$. By the induction hypothesis b' is a open bundle of L and therefore $b' \subseteq G_L$. It follows that

$$(\alpha, ind(b')) \,.\, b' \subseteq (\alpha, ind(G_L)) \,.\, G_L = G_{\alpha.L} \ ,$$

thus $b \subseteq G_{\alpha.L}$. Conflict freeness (iv) of b follows from that of b'.

Case $S = \sum_{j\in J} S_j$. If $b \in \mathcal{O}(\sum_{j\in J} S_j)$ then there exists $j \in J$ such that $b = j : b'$ and $b' \in \mathcal{O}(S_j)$. By the induction hypothesis b' is an open bundle of S_j, thus $j : b'$ is clearly an open bundle of $\sum_{j\in J} S_j$.

Case $S = ||_{j\in J} S_j$. If $b \in \mathcal{O}(||_{j\in J} S_j)$ then there exists a finite set $I \subseteq J$ such that $b \in (\bigcup_{i\in I} b_i)\!\uparrow$ and $b_i \in \mathcal{O}(S_i)$ for every $i \in I$. The control graph b is an open bundle of the same strand space for which $||_{i\in I} b_i$ is an open bundle. From the induction hypothesis it follows that for every $i \in I$ the graph b_i is an open bundle of S_i and therefore $b_i \subseteq G_{S_i}$. Since $I \subseteq J$ it follows that

$$\bigcup_{i\in I} b_i \subseteq \bigcup_{j\in J} G_{S_j} \preceq G_S$$

and therefore b is an open bundle of S as desired. □

7.6 Strand-space equations

We have seen an equivalence relation that relates two strand spaces if, via re-indexing, they become the same space. It is easy to check that this relation is a congruence with respect to the operations of the strand-space language we introduced earlier in this chapter. It is however a very concrete relation and too discriminating for a model in which security properties are expressed as safety properties on the language of bundles of a strand space. One doesn't want to distinguish between strand spaces that have isomorphic bundle languages. Unfortunately the equivalence relation $\approx$ on strand-space terms, obtained by taking term equivalence iff they denote strand spaces with essentially the same bundles, is not a congruence. In this section we study a finer equivalence that takes account of the open bundles of a strand space rather than bundles. This relation turns out to be an interesting congruence, in fact the largest congruence within $\approx$.

7.6.1 A strand-space equivalence

Let b, b' be two bundles. Write $b \cong b'$ iff there exists a bijection $\phi : E_b \to E_{b'}$ such that

(i) if $e \to_b e'$ then $\phi(e) \to_{b'} \phi(e')$,

(ii) if $e \Rightarrow_b e'$ then $\phi(e) \Rightarrow_{b'} \phi(e')$,

(iii) $act_b(e) = act_{b'}(\phi(e))$.

Definition 7.6.1 Let S and S' be two strand-space terms in $\mathcal{S}$. Define $\approx$ the symmetric relation such that $S \approx S'$ iff for every bundle b of S there exists a bundle b' of S' such that $b \cong b'$.

Proposition 7.6.2 *The relation $\approx$ is an equivalence relation.* □

The equivalence relation $\approx$ is not a congruence relation. Consider, for example, the strand-space terms $\mathit{in}\, M\,.\,\epsilon$ and $\mathit{in}\, N\,.\,\epsilon$ where N and M are two different messages. These two strand-space terms are in the relation $\approx$ – they both have only one bundle, the empty bundle and they can be easily distinguished in a simple context when, for example, composed in parallel with $\mathit{out}\, M\,.\,\epsilon$. Then

$$\mathit{in}\, M\,.\,\epsilon \mid\mid \mathit{out}\, M\,.\,\epsilon \not\approx \mathit{in}\, N\,.\,\epsilon \mid\mid \mathit{out}\, M\,.\,\epsilon\ .$$

The parallel composition on the left hand side has a bundle of the form

$$\begin{tikzcd} \mathit{in}\, M & \mathit{out}\, M \arrow[l] \end{tikzcd} \quad ,$$

that on the right only λ.

7.6.2 A strand-space congruence

A context for a strand-space term in the language $\mathcal{S}$ is defined as follows:

$$C ::= [\ ] \mid \alpha.C \mid ||_{i\in I}T_i \mid \Sigma_{i\in I}T_i$$

where for each context of the form $||_{i\in I}T_i$ or $\Sigma_{i\in I}T_i$ there is exactly one $i \in I$ such that T_i is a context C and $T_j \in \mathcal{S}$ for all $j \in I \setminus \{i\}$. The context $[\ ]$ is a placeholder for a strand-space term. We write $C[S]$ for the term obtained by replacing the strand-space term S for $[\ ]$ in the context C in the obvious way. An equivalence relation on strand-space terms is a congruence if it respects all contexts.

Definition 7.6.3 Let S and S' be two strand-space terms in $\mathcal{S}$. Define $\approx_{\mathcal{O}}$ to be the symmetric relation such that $S \approx_{\mathcal{O}} S'$ if for every open bundle b of S there exists an open bundle b' of S' such that $b \cong b'$.

Proposition 7.6.4 *The relation $\approx_{\mathcal{O}}$ is a congruence.*

Proof. The relation $\approx_{\mathcal{O}}$ is obviously an equivalence relation. We show by induction on the structure of contexts that if $S \approx_{\mathcal{B}} S'$ then $C[S] \approx_{\mathcal{B}} C[S']$ for every context C. We show that for every open bundle b of $C[S]$ there exists an open bundle b' of $C[S']$ such that $b \cong b'$.

Case $C = [\ ]$. Obvious.

Case $C = \alpha.C'$ for some context C'. Let

$$b \in \mathcal{O}(\alpha.C[S])\ .$$

From Definition 7.5.5 of bundle space it follows that b is one of the following graphs:

- $b = \lambda$ and therefore clearly belongs to $\mathcal{O}(\alpha.C[S'])$.
- $b = (\alpha, \{i\})\,.\,\lambda$ where $i \in \mathit{ind}(C[S])$ thus for every $j \in \mathit{ind}(C[S'])$

$$b \cong (\alpha, \{j\})\,.\,\lambda \in \mathcal{O}(\alpha.C[S'])\ .$$

- $b \in ((\alpha, ind(b')) \,.\, b') \uparrow$ where $b' \in \mathcal{O}(C[S]) \setminus \{\lambda\}$. By the induction hypothesis there exists $b'' \in \mathcal{O}(C[S'])$ such that $b' \cong b''$ and therefore

  $$(\alpha, ind(b')) \,.\, b' \cong (\alpha, ind(b'')) \,.\, b''$$

  and

  $$((\alpha, ind(b'')) \,.\, b'')\uparrow \subseteq \ \mathcal{O}(\alpha.C[S']) \ .$$

  Let ϕ be a bijection such that $\phi((\alpha, ind(b')) \,.\, b') = (\alpha, ind(b'')) \,.\, b''$. Clearly

  $$b \cong \phi(b) \in ((\alpha, ind(b'')) \,.\, b'')\uparrow \ .$$

Case $C = \Sigma_{i \in I} T_i$ where the term T_j is a context C for the index j in I. Let

$$b \in \mathcal{O}(\Sigma_{i \in I} T_i[S]) \ .$$

From Definition 7.5.5 of bundle space it follows that the open bundle $b = i : b_i$ for some $i \in I$ and for some $b_i \in \mathcal{O}(T_i[S])$. If $i \neq j$ then $i : b_i \in \mathcal{O}(\Sigma_{i \in I} T_i[S'])$. If instead $i = j$ then by the induction hypothesis there exists $b'_i \in \mathcal{O}(C[S'])$ such that $b_i \cong b'_i$ and therefore

$$i : b_i \cong i : b'_i \in \mathcal{O}(\Sigma_{i \in I} T_i[S']) \ .$$

Case $C = ||_{i \in I} T_i$ such that only the term T_j is a context where $j \in I$. If

$$b \in \mathcal{O}(||_{i \in I} T_i[S])$$

then

$$b \in (\bigcup_{i \in J} b_i)\uparrow$$

where $b_i \in \mathcal{O}(T_i)$ when $i \neq j$ and $b_j \in \mathcal{O}(C[S])$. By the induction hypothesis there exists

$$b'_j \in \mathcal{O}(C[S'])$$

such that $b_j \cong b'_j$. Let $b'_i = b_i$ for all $i \neq j$. It follows that

$$(\bigcup_{i \in J} b'_i)\uparrow \subseteq \ \mathcal{O}(||_{i \in I} T_i[S']) \ .$$

Moreover $\bigcup_{i \in J} b_i \cong \bigcup_{i \in J} b'_i$. Let ϕ is the bijection such that $\phi(\bigcup_{i \in J} b_i) = \bigcup_{i \in J} b'_i$. Clearly

$$b \cong \phi(b) \in (\bigcup_{i \in J} b'_i) \uparrow \ .$$

□

Theorem 7.6.5 *The relation $\approx_{\mathcal{O}}$ is the largest congruence relation inside $\approx$.*

Proof. Consider the set

$$D = \{R \mid R \text{ congruence relation and } R \subseteq \approx\}$$

Clearly $\bigcup D$ is a congruence relation and the largest congruence inside $\approx$. It remains to show that

$$\approx_{\mathcal{O}} = \bigcup D .$$

By Proposition 7.6.4 the relation $\approx_{\mathcal{B}}$ is a congruence and since $\approx_{\mathcal{O}} \subset \approx$ it follows that $\approx_{\mathcal{O}} \subseteq \bigcup D$. Let S and S' be two strand-space terms and let b be an open bundle of S such that $b \ncong b'$ for all open bundles b' of S'. If no congruence relation in D contains the pair (S, S') then $\bigcup D \subseteq \approx_{\mathcal{O}}$. Let $R \in D$ and suppose $(S, S') \in R$. We find a context C such that $(C[S], C[S']) \notin R$ and therefore R is not a congruence relation. Consider the strand space T as defined in the construction we saw in the proof of Theorem 7.5.4. Consider the context $T \mid\mid [\,]$. Let t be an open bundle of T and $b'' \in (t \mid\mid b) \uparrow$ be the bundle of $T \mid\mid S$ as defined in the proof of Theorem 7.5.4. Since

$$(T \mid\mid S, T \mid\mid S') \in R$$

there exists a bundle d of $T \mid\mid S'$ such that

$$b'' \cong d .$$

Let ϕ be an isomorphism between the two bundles. Clearly $b''_{|2} = b$ and

$$\phi(2 : b)_{|2}$$

is an open bundle since b is. Observe that $\phi(2 : b)_{|2} \subseteq G_{S'}$ and therefore is an open bundle of S' – otherwise, if $\phi(2 : b)_{|2} \nsubseteq G_{S'}$ there would be an event of b that introduces as new the same name that is introduced in t (strands of T all start with the generation of a new name not belonging to b), which by construction is not possible. Clearly $b \cong \phi(2 : b)_{|2}$ and we reach a contradiction.

□

7.7 Eliminating conflict

An agent that participates in a security protocol is often thought of as executing a sequential program during which it sends messages and chooses from a number of available messages which one to input. If one doesn't restrict the number of rounds of the protocol an agent can do one can hope to model the protocol with a strand space with conflict of the form $!(\langle s_i \rangle_{i \in I}, \#)$. In this section we show that under these conditions a simpler model, obtained by dropping the conflict relation, exhibits substantially the same behaviour as the more complex strand space with conflict.

Consider a conflict relation $\#$ on a set I and consider the relation $!\#$ on $\omega \times I$ such that $(n, i) \,!\#\, (m, j)$ iff $n = m$ and $i \# j$. The form of the $!\#$ conflict relation suggests the following lemma:

Lemma 7.7.1 *Given I a set of indices, a finite set $A \subseteq \omega \times I$, and $\#$ a conflict relation over I then there exists a bijection $\pi : \omega \times I \to \omega \times I$ where for all $(n, i) \in \omega \times I$ there exists $m \in \omega$ such that $\pi(n, i) = (m, i)$ and $\neg(\pi(u) \,!\#\, \pi(v))$ for all $u, v \in A$.*

Proof. By induction on the size of the set A.

The basic case $A = \emptyset$: take π for example to be the identity function.

The induction step: Given a set A suppose there is a bijection π satisfying the desired property and such that A is conflict free with respect to $\pi^{-1}(!\#)$. Take $A' = A \cup \{(n', i')\}$ with $(n', i') \notin A$. Let $\pi(n', i') = (m', i')$ and distinguish two cases:

If $m \neq m'$ for all $(m, j) \in \pi(A)$ then A' is conflict free with respect to $\pi^{-1}(!\#)$. Suppose the contrary. Suppose there is $(n, i) \in A$ such that $\pi(n, i) \,!\#\, \pi(n', i')$. If $\pi(n, i) = (m, i)$ then $m = m'$ by definition of $!\#$.

If there exists $(m, j) \in \pi(A)$ such that $m = m'$, then consider the following function:

$$\pi'(n, i) = \begin{cases} (m'', i') & \text{if } (n, i) = (n', i') \\ (m', i') & \text{if } (n, i) = \pi^{-1}(m'', i') \\ \pi(n, i) & \text{otherwise} \end{cases}$$

where $m'' \in \omega$ such that $m \neq m''$ for all $(m, j) \in \pi(A)$. Since A is a finite set, such m'' exists. This function is as π but swaps the new element (n', i'), that could introduce conflict, with one that doesn't. It is easy to check that π' is a bijection and that it satisfies the required property. It remains to check that A' is conflict free with respect to ${\pi'}^{-1}(!\#)$. First observe that $\pi'(A) = \pi(A)$ because $(n', i') \notin A$ and $\pi^{-1}(m'', i') \notin A$ (we chose m'' so that $(m'', i') \notin \pi(A)$). It follows that A is conflict free with respect to ${\pi'}^{-1}(!\#)$. Since ${\pi'}^{-1}(!\#)$ is irreflexive and symmetric we require:

$$\forall (n, i) \in A \,.\, \neg(\pi'(n, i) \,!\#\, \pi'(n', i')).$$

Suppose instead that there exists $(n, i) \in A$ such that $\pi(n, i) \,!\#\, \pi(n', i')$. Let $\pi(n, i) = (m, i)$, then $(m, i) \,!\#\, (m'', i')$. From the conflict relation $!\#$ it follows that $m = m''$. On the other hand $(m, i) \in \pi(A)$. Therefore $m \neq m''$.

□

The previous lemma and the observation that two different copies of the same strand space have the same strands at corresponding positions, yield:

Theorem 7.7.2 *Consider strand spaces* $!\,(\langle s_i \rangle_{i \in I}, \emptyset)$ *and* $!\,(\langle s_i \rangle_{i \in I}, \#)$. *Let* b *be a bundle of* $!\,(\langle s_i \rangle_{i \in I}, \emptyset)$. *There exists a strand space* S *such that* b *is a bundle of* S *and* $S \cong !\,(\langle s_i \rangle_{i \in I}, \#)$.

Proof. Given b a bundle of $!\,(\langle s_i \rangle_{i \in I}, \emptyset)$ take A the set of indices of strands participating with nodes in b. The set A is finite because so is b and $A \subseteq \omega \times I$. By Lemma 7.7.1 there is a bijection $\pi : \omega \times I \to \omega \times I$ such that

$$\forall (n, i) \in \omega \times I \,.\, \pi(n, i) = (n', i)$$

for some $n' \in \omega$ and A is conflict free with respect to $\pi^{-1}(!\#)$. Consider the strand space with conflict

$$S = (\langle t_{(m,i)} \rangle_{(m,i) \in \omega \times I}, \pi^{-1}(!\#))$$

where $t_{(m,i)} = s_i$ for all $m \in \omega$. The equivalence $S \cong !(\langle s_i \rangle_{i \in I}, \#)$ stands. In fact π is a bijection such that

(i) $t_{(m,i)} = s_{\pi(m,i)}$ since $s_{\pi(m,i)} = s_{(n',i)} = s_i = t_{(m,i)}$

(ii) $(m,i)\,\pi^{-1}(!\#)\,(u,j)$ iff $\pi(m,i)\,!\#\,\pi(u,j)$.

The two strand spaces $!\,(\langle s_i\rangle_{i\in I},\emptyset)$ and S have the same strand-space graph, therefore since b is a bundle over $!(\langle s_i\rangle_{i\in I},\emptyset)$, and since A is conflict free with respect to $\pi^{-1}(!\#)$ it follows that b is a bundle of S. □

To summarise, the behaviour in terms of bundles of a replicated strand space with conflict corresponds, modulo re-indexing, to that of the strand space we obtain dropping the conflict relation.

Corollary 7.7.3 *Consider strand spaces* $!(\langle s_i\rangle_{i\in I},\emptyset)$ *and* $!(\langle s_i\rangle_{i\in I},\#)$.

1. *If b is bundle of $!\,(\langle s_i\rangle_{i\in I},\#)$ then b is bundle of $!\,(\langle s_i\rangle_{i\in I},\emptyset)$.*
2. *If b is bundle of $!\,(\langle s_i\rangle_{i\in I},\emptyset)$ then there exists a re-indexing π such that $\pi(b)$ is a bundle of $!\,(\langle s_i\rangle_{i\in I},\#)$.*

Proof. The first point is obvious, the second point follows directly from Theorem 7.7.2 and Proposition 7.2.6. □

Consequently, the strand space with conflict denoted by a "!-par" process has the same bundles up to re-indexing as a strand space without conflict.

Independently from the work described in this thesis, another treatment of strand spaces with conflict is due to Halpern and Pucella [37]. The previous section, which shows that conflict can be eliminated for a broad range of protocols, is an advance over their results. Conflict elimination appears to follow from a basic design principle for cryptographic protocols. Principals usually are not required to maintain state across different sessions but execute protocol sessions in a purely local way. Denial-of-Service attacks that are based on the state that would have to be maintained can so be avoided.

Chapter 8

Relating models

A number of security-protocol models exist. Some of them have been compared and formally related. For example the strand-space method has been related to the multiset-rewriting method [17] and to linear logic [15]. For many other methods, however, formal comparisons have not been studied yet. One can get a better understanding of various security-protocol models when one compares them to other models. In this chapter we relate a number of event-based approaches to security protocol analysis.

This Chapter starts by showing how strand spaces [90] relate to event structures [94]. This result is an example of how models that have been developed for a special purpose, the verification of security protocols, are often more traditional general-purpose models for concurrency in disguise. Later in this Chapter we study the relation of SPL with other security-protocol models. We are particularly interested in comparing SPL with the strand-spaces and Paulson's inductive approach [61] since these two more intentional event-based models inspired the design of SPL. As a side result of our comparison, a link between the traditional transition-system semantics of process languages and strand spaces. The Chapter ends with a comparison between SPL and other traditional models for concurrency: basic and coloured Petri-nets [66, 95, 42], event-structures and Mazurkiewicz trace languages [50].

Security properties such as secrecy and authentication or even anonymity can be expressed as safety properties, properties which stand or fall according to whether they hold for all finite behaviours. The results in this chapter relate models with respect to the languages, i.e., sets of finite behaviours, they generate. This is not unduly restrictive, however, as in security protocols we are mainly interested in safety properties.

8.1 Event structures from strand spaces

In this section we show how strand spaces relate to event structures. Recall that a bundle of a strand space is a graph and therefore a set of edges and nodes. It turns out that the bundles of a strand space ordered by inclusion correspond to the finite configurations of a prime event structure. Prime event structures are a particularly simple kind of event structure where the enabling of events can be expressed as a global partial order of causal dependency. Event structures as a model of concurrent processes were introduced in [58, 92] – see also [93, 94, 96, 97].

8.1.1 Prime event structures

Definition 8.1.1 A prime event structure $(E, \#, \leq)$ consists of a set E of events partially ordered by the causal dependency relation $\leq$ and a binary, symmetric, irreflexive conflict relation $\# \subseteq E \times E$, which satisfy

(i) $\{e' \mid e' \leq e\}$ is finite for all $e \in E$, and

(ii) if $e \# e' \leq e''$ then $e \# e''$ for all $e, e', e'' \in E$.

Definition 8.1.2 The configurations of an event structure $E = (E, \#, \leq)$ consist of those subsets $x \subseteq E$ which are

(i) conflict free: $\forall e, e' \in x \,.\, \neg(e \# e')$ and

(ii) left closed: $\forall e, e'.e' \leq e \in x \Rightarrow e' \in x$.

Write $\mathcal{F}(E)$ for the set of configurations of an event structure and $\mathcal{F}^{fin}(E)$ for its finite configurations.

8.1.2 A prime event structure from a strand space

For a strand space S write $\mathcal{B}$ for the set of bundles of S. Consider the partial order $(\mathcal{B}, \subseteq)$. Bundles are here viewed as sets of nodes and edges. Say a subset X of $\mathcal{B}$ is *compatible* iff

$$\exists b \in \mathcal{B} \,.\, \forall b' \in X \,.\, b' \subseteq b \ .$$

We will say X is pairwise compatible if for every two bundles $b_1, b_2 \in X$ there exists a bundle $b \in \mathcal{B}$ such that $b_1 \subseteq b$ and $b_2 \subseteq b$.

Proposition 8.1.3 *Let S be a strand space and $\mathcal{B}$ the set of bundles of S. The partial order $(\mathcal{B}, \subseteq)$ satisfies the following properties:*

1. *if $X \subseteq \mathcal{B}$, X is finite and pairwise compatible, then $\bigcup X \in \mathcal{B}$.* *(coherence)*

2. *if $X \subseteq \mathcal{B}$ and X is compatible, then $\bigcap X \in \mathcal{B}$.* *(stability)*

Proof. Let G be the strand space graph of S.

1. The graph $\bigcup X$ is obviously a finite subgraph of G since X is finite and each element in X is finite. We check that the graph $\bigcup X$ satisfies all the other requirements of Definition 7.2.2:

   (i) Consider an event $e \in \bigcup X$. There is a bundle $b \in X$ with $e \in b$. By control precedence on b, if $e' \Rightarrow_G e$ then $e' \Rightarrow_b e$. From $b \subseteq \bigcup X$ it follows that $e' \Rightarrow_{\bigcup X} e$.

(ii) Output-input precedence holds. Let e be an input event in $\bigcup X$. There is a bundle $b \in \bigcup X$ with $e \in b$. An output event e' exists such that $e' \to_b e$ (output-input precedence on b) and therefore $e' \to_{\bigcup X} e$. There is a unique such output event. Suppose the contrary. Suppose there are two output events e', e'' such that $e' \to_{\bigcup X} e$ and $e'' \to_{\bigcup X} e$. Then two distinct bundles $b, b' \in X$ exist such that $e' \to_b e$ and $e'' \to_{b'} e$. On the other hand X is pairwise compatible and so there is a bundle containing both b and b' which would however contradict output-input precedence on that bigger bundle.

(iii) Let e, e' be two events in $\bigcup X$ such that

$$act(e) = out\, new\, \vec{n}\, M \quad \text{and} \quad n \in \vec{n} \cap names(e') .$$

Since X is pairwise compatible, there exists a bundle b' in X that contains both e and e'. The bundle b' respects freshness and so either $e \Rightarrow^*_{b'} e'$ or there exists an input event e'' such that $n \in names(e'')$ and $e'' \Rightarrow^*_{b'} e'$. Consequently freshness holds on $\bigcup X$.

(vi) The graph $\bigcup X$ does not contain conflicting events since X is pairwise compatible.

(v) Let e be an event in $\bigcup X$. The set X is pairwise compatible therefore there exists $b \in X$ with $e \in b$ and such that if $e' \to_{\bigcup X} e$ then $e' \to_b e$. Since b is a bundle if $e' \Rightarrow_{\bigcup X} e$ then $e' \Rightarrow_b e$ (control precedence). Therefore

$$\{e' \mid e' \preceq_{\bigcup X} e\} = \{e' \mid e' \preceq_b e\}$$

and it follows that the relation $\Rightarrow_{\bigcup X} \cup \to_{\bigcup X}$ is acyclic.

Therefore $\bigcup X \in \mathcal{B}$.

2. Every element of X is a bundle in $\mathcal{B}$. Therefore $\bigcap X$ forms a finite subgraph of G. We check that the graph $\bigcap X$ satisfies all the other requirements of Definition 7.2.2.

(i) Let $e \in \bigcap X$. Every $b \in X$ is such that $e \in b$. For each $b \in X$ if $e' \Rightarrow_G e$ then $e' \Rightarrow_b e$ (control precedence on b). It follows that $e' \Rightarrow_{\bigcap X} e$.

(ii) If $e \in \bigcap X$ and e is an input event then every bundle $b \in X$ contains the event e and contains an output event e_b such that $e_b \to_b e$. The set X is compatible and therefore there exists $b' \in \mathcal{B}$ such that $e_b \to_{b'} e$ for every $b \in X$. Since b' is a bundle, there exists an event e' such that $e' = e_b$ for every $b \in X$. The event e' is the unique event such that $e' \to_{\bigcap X} e$.

(iii) Let e, e' be two events in $\bigcap X$ such that

$$act(e) = out\, new\, \vec{n}\, M \quad \text{and} \quad n \in \vec{n} \cap names(e') .$$

The events e, e' belong to every bundle in X. Freshness holds on each bundle b in X. Thus either $e \Rightarrow^*_b e'$ or there exists an input event e'' such that $n \in names(e'')$ and $e'' \Rightarrow^*_b e'$. If $e \Rightarrow^*_{b'} e'$ for some bundle b' in X then $e \Rightarrow^*_G e$ and therefore $e \Rightarrow^*_b e'$ for every bundle b in X. The other case of a bundle b' where $e'' \Rightarrow^*_b e'$ is similar. Freshness holds on $\bigcap X$.

(vi) The elements of X are bundles and therefore conflict free. Consequently $\bigcap X$ is conflict free.

(v) Every element of X is a bundle in $\mathcal{B}$ therefore the relation $\Rightarrow_{\bigcap X} \cap \rightarrow_{\bigcap X}$ is acyclic.

Therefore $\bigcap X \in \mathcal{B}$.

$\square$

Given a bundle $b \in \mathcal{B}$ and an event $e \in E_b$ define

$$\lceil e \rceil_b \overset{def}{=} \bigcap \{b' \in \mathcal{B} \mid e \in b' \ \wedge \ b' \subseteq b\} \ .$$

Proposition 8.1.4 *Let $\mathcal{B}$ be the bundles of a strand space. For every bundle $b \in \mathcal{B}$ and every event $e \in E_b$ the set $\lceil e \rceil_b$ is a bundle in $\mathcal{B}$. For every finite and compatible subset $X \subseteq \mathcal{B}$*

$$\textit{if } \lceil e \rceil_b \subseteq \bigcup X \textit{ then } \exists b' \in X \,.\, \lceil e \rceil_b \subseteq b'$$

We call a bundle $\lceil e \rceil_b$ a prime of $(\mathcal{B}, \subseteq)$.

Proof. For ever $b \in \mathcal{B}$ and every $e \in b$ the set

$$\{b' \in \mathcal{B} \mid e \in b' \ \wedge \ b' \subseteq b\}$$

is compatible. It follows from the stability property of $(\mathcal{B}, \subseteq)$ (Proposition 8.1.3) that the elements $\lceil e \rceil_b$ are bundles in $\mathcal{B}$.

Let $X \subseteq \mathcal{B}$ be finite and compatible. If $\lceil e \rceil_b \subseteq \bigcup X$ then $e \in b'$ for some $b' \in X$. The two bundles b' and $\lceil e \rceil_b$ in $\mathcal{B}$ are compatible (they are both included in $\bigcup X$), therefore from the stability property of $(\mathcal{B}, \subseteq)$ it follows that $b' \cap \lceil e \rceil_b \in \mathcal{B}$. We know that $e \in b'$ and that $b' \cap \lceil e \rceil_b \subseteq b$. Therefore $\lceil e \rceil_b \subseteq b' \cap \lceil e \rceil_b \subseteq b'$. $\square$

The primes form a basis for the partial order $(\mathcal{B}, \subseteq)$ as the following proposition shows.

Proposition 8.1.5 *Let S be a strand space and $\mathcal{B}$ its bundles. Every bundle $b \in \mathcal{B}$ can be obtained as the union of the primes included in b*

$$b = \bigcup \{p \mid p \subseteq b, \ p \textit{ prime}\}.$$

Proof. Obviously

$$\bigcup \{p \mid p \subseteq b, \ p \text{ prime}\} \subseteq b \ .$$

On the other hand, if $e \in b$ then $e \in \lceil e \rceil_b$. The prime $\lceil e \rceil_b$ is a bundle included in b and therefore if $e' \Rightarrow_b e$ then $e' \Rightarrow_{\lceil e \rceil_b} e$, and if $e' \rightarrow_b e$ then $e' \rightarrow_{\lceil e \rceil_b} e$. Thus

$$b \subseteq \bigcup \{\lceil e \rceil_b \mid e \in b\} \ .$$

$\square$

Let $\mathcal{B}$ the set of bundles of a strand space S. Consider the following structure

$$Pr(\mathcal{B}) \overset{def}{=} (P, \#, \subseteq)$$

where P is the set of primes of $\mathcal{B}$ and where $p \,\#\, p'$ iff the two primes p and p' are not compatible.

Theorem 8.1.6 *The structure $Pr(\mathcal{B})$ is a prime event structure.*

Proof. It is easy to check that $Pr(\mathcal{B})$ is a prime event structure. The relation $\subseteq$ is a partial order of primes. Bundles and in particular primes are finite sets. Thus for each prime p the set $\{p' \mid p' \subseteq p\}$ is finite. Moreover if p, p', p'' are primes such that $p \,\#\, p' \subseteq p''$ then $p \,\#\, p''$ – otherwise p and p'' would be compatible and therefore p and p' would be compatible too. □

8.1.3 Relating strand spaces and prime event structures

Theorem 8.1.7 *The map*

$$\phi : (\mathcal{B}, \subseteq) \cong (\mathcal{F}^{fin} Pr(\mathcal{B}), \subseteq)$$

such that

$$\phi(b) = \{p \mid p \subseteq b,\ p \ prime\}$$

is an isomorphism of partial orders with inverse map $\theta : \mathcal{F}^{fin} Pr(\mathcal{B}) \to \mathcal{B}$ given by $\theta(x) = \bigcup x$.

Proof. The map ϕ is obviously well-defined and so is θ. If x is a finite configuration of $Pr(\mathcal{B})$, it is then a finite set of bundles in $\mathcal{B}$ which is conflict free and left closed. Since it is conflict free it is pairwise compatible. Therefore from the coherence property of Proposition 8.1.3 it follows that $\bigcup x \in \mathcal{B}$.

It is clear that both maps ϕ and θ are order preserving. We show that they are mutual inverses and therefore give the required isomorphism. From Proposition 8.1.5 it follows that $\theta(\phi(b)) = b$ for all $b \in \mathcal{B}$. Let x be a finite configuration of $Pr(\mathcal{B})$. We require that $\phi(\theta(x)) = x$, i.e.,

$$\{p \mid p \subseteq \bigcup x,\ p \text{ prime}\} = x\ .$$

If p is a prime such that $p \subseteq \bigcup x$, then there exists a bundle p' in x (another prime) such that $p \subseteq p'$ (Proposition 8.1.4). The configuration x is left-closed, thus $p \in x$. On the other hand, x is a set of primes and therefore

$$x \subseteq \{p \mid p \subseteq \bigcup x,\ p \text{ prime}\}\ .$$

□

Observation 8.1.8 Since $Pr(\mathcal{B})$ is a prime event structure the partial order

$$(\mathcal{F} Pr(\mathcal{B}), \subseteq)$$

is a Scott domain of information, in particular a prime algebraic domain (see [58, 94, 96]). This domain however can include configurations which are infinite and therefore are not bundles in the usual sense.

The prime event structure $Pr(\mathcal{B})$ underlies the strand space semantics Syverson gave to the BAN logic [83].

8.2 Strand spaces from SPL

The behaviour of a strand space expressed in terms of bundles is closely related to the runs of an SPL net and to that of a more traditional model, the SPL-transition semantics. For "par" processes and strand spaces with conflict, and for "!par" processes and traditional strand spaces one can show that the events, or actions in a run of the process are those appearing in a "linearisation" of a bundle of a certain strand space.

8.2.1 "Par" processes

Agents that participate in a protocol, execute a program that implements a protocol role. In many cases (see for example the protocols listed in the Clark-Jacob library [19]) the program consists of a sequence of instructions that send messages onto the network or receive messages from the network. Then a security protocol is a distributed program that can be conveniently described as a parallel composition of sequential processes, a "par" process. Usually the components of a "par" process describing a protocol are replicated – no restrictions are imposed on the number of rounds an agent can engage in the same protocol role. We call the replicated "par" processes "!par".

The set of SPL-"par" processes is described by the following grammar:

$$\begin{array}{lcl} sp & ::= & nil \mid out\, new\, \vec{x}\ M\,.\,sp \mid in\, pat\, \vec{x}\vec{\chi}\vec{\psi}\ M\,.\,sp \\ p & ::= & sp \mid \|_{i\in I} p_i \end{array}$$

Recall that the process term nil in SPL is an abbreviation for the empty parallel composition. The "par" processes clearly form a subset of SPL processes.

The examples of security protocols that we studied in this thesis are "!par" processes.

8.2.2 A strand space from a "par" process

The components of a "par" process denote a set of strands, a strand for each maximal sequence of events of that component coinciding at control points. For example to an initiator component of the NSL system

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

correspond maximal event sequences

$$\mathbf{Out}(out\, new\, x\ \{x, A\}_{Pub(B)}; n)\ \mathbf{In}(in\ \{n, y, B\}_{Pub(A)}; m)\ \mathbf{Out}(out\ \{m\}_{Pub(B)})$$

for each $n, m \in \mathbf{N} \setminus \{A, B\}$ which denote strands

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

The strand space associated to an SPL-"par" process that we construct in this section consists of all the strands obtained from all components of the process.

More precisely, let p be a closed process term. Define $I(p)$ the set of maximal sequences $e_1 \cdots e_l$ of events in $Ev(p)$ such that

1. ${}^{c}e_1 \subseteq Ic(p)$,
2. $e_i^n \cap names(p) = \emptyset$ for all $1 \leq i \leq l$, and
3. $e_i^c = {}^{c}e_{i+1}$ for all $1 \leq i < l$.

Note that if p is a "par" process, all the events in a sequence $u \in I(p)$ have the same index that we denote with $index(u)$. If the events in u do not carry an index then let $index(u) = *$ where $*$ is a distinguished index.

Proposition 8.2.1 *Let $e_1 \cdots e_l \in I(p)$. If $i \neq j$ then ${}^{c}e_i \neq^{c} e_j$.*

Proof. The size of control conditions decreases strictly along a sequence of events in $I(p)$ – each control postcondition of an event is smaller in size to the control precondition of the event. □

The events that appear in a sequence of $I(p)$ can be seen as events of a strand-space graph, with precedence edges defined by the precedence among events of a sequence and by all possible communication edges between matching output and input messages. The actions associated to the nodes are those of the events. From an SPL-"par" process one can construct a strand space as follows:

Definition 8.2.2 *(A strand space from an SPL-"par" process).* Given a closed "par" process p define

$$S(p) = (\langle s_u \rangle_{u \in I(p)}, \#)$$

to be the strand space where

1. $s_u[j] = act(u[j])$ for every $u \in I(p)$ and j such that $1 \leq j \leq |u|$,
2. $u \# v$ iff $index(u) = index(v)$ and $u \neq v$.

□

The graph of a strand space $S(p)$ has nodes of the from (u, i) with $u \in I(p)$ and $1 \leq i \leq |u|$. Since u is a sequence of events in $Ev(p)$ each node in the graph denotes the event $u[i] \in Ev(p)$. For this reason we will sometimes make no distinction between nodes of a strand-space graph and the events that they may denote.

8.2.3 Relating SPL-net semantics and strand spaces

The behaviour of a strand space obtained from a "par" process is closely related to its net semantics. Bundles can be linearised into sequences of events, so that the casual dependencies among events in a bundle determines how events precede each other in a linearisation of the bundle. Clearly there are many ways to linearise a bundle. It turns out that there is a one to one correspondence between runs in the net of a "par" process and linearisations of all possible bundles of the strand space obtained from that process.

Definition 8.2.3 *(Linearisation).* Given a bundle b of a strand space, a *linearisation* of b is a sequence of nodes $e_1 \cdots e_l$ such that $\{e_1, \ldots, e_l\}$ are all the nodes of b and for all $e_i, e_j \in b$ if $e_i \Rightarrow_b e_j$ or $e_i \rightarrow_b e_j$ then e_i precedes e_j in the sequence.

In a strand space $S(p)$ a linearisation of a bundle denotes a sequence of events in $Ev(p)$ in the obvious way, that we call *event-linearisation* of the bundle. □

The set of linearisations of bundles of a strand space is prefix closed – given a strand space and a linearisation of one of its bundles we find that every prefix of the linearisation is a linearisation of a bundle of the strand space.

Proposition 8.2.4 *Let L be a sequence of nodes and e a node such that Le is a linearisation of a bundle of a strand space S. The sequence L is a linearisation of a bundle of S.*

Proof. Let b be the bundle of S that has as one of its linearisations the sequence Le. Removing from b the node e and all edges incident to e results in another bundle of S. In fact the graph that one obtains is clearly finite and acyclic and conflict free. The sequence Le is a linearisation of b thus b does not contain any outgoing edges of e so that removing e from b does not violate neither control, nor output-input precedence, nor freshness and L is clearly a linearisation of the reduced bundle. □

Let p_0 be a closed "par" process. We construct a set of bundles of $S(p_0)$ from a proper run in $Net(p_0)$ such such that the events in the run form an event-linearisation of every bundle in the set. A bundle of $S(p_0)$ is a subgraph of its strand-space graph and has nodes (u, i) where $u \in I(p_0)$ and $1 \leq i \leq |u|$. Consider an event in the run and suppose it has index r. The subsequence of the run composed by all the events that have index r is a prefix of a sequence u in $I(p_0)$. If the length of the prefix is j then the nodes (u, i) with $i \leq j$ should belong to one of the bundles that we constructed from the run. There can be more than one sequence in $I(p_0)$ which has a prefix formed out of all the events of a run with a common index. A run determines a set of bundles rather than a single bundle.

Let $v = e_1 \cdots e_n$ be the sequence of events in a finite and proper run

$$\langle p_0, s_0, \emptyset\rangle \xrightarrow{e_1} \cdots \xrightarrow{e_n} \langle p_n, s_n, t_n\rangle$$

in the SPL-net, where p_0 is a closed "par" process. Denote with $v_{|r}$ the subsequence of v consisting of all those output and input events that have index r.

Proposition 8.2.5 *Let r be an index of an event in the sequence v. There exists $u \in I(p_0)$ such that $v_{|r}$ is prefix of u.*

Proof. We show that the sequence $v_{|r} = a_1 \cdots$ is such that

1. ${}^{c}a_1 \subseteq Ic(p_0)$,
2. $a_i^n \cap names(p) = \emptyset$ for all $1 \leq i \leq |v_{|r}|$, and
3. $a_i^c = {}^{c}a_{i+1}$ for all $1 \leq i < |v_{|r}|$,

and therefore, since it is composed out of events in $Ev(p_0)$ it is clearly a prefix of a sequence in $I(p_0)$.

Among the events that carry index r, the event a_1 is the first to appear in v, an SPL run, and therefore a_1 has no preceding event that would mark its control condition – a_1 necessarily has index r. It follows from the token game that

$$^{c}a_1 \subseteq Ic(p_0) \ .$$

The events in v form a proper run of $Net(p)$ for which the Freshness Principle 5.1.3 holds. Therefore $a_i^n \cap names(p) = \emptyset$ for all $1 \leq i \leq |v_{|r}|$.

Suppose that $a_i^c \neq {}^{c}a_{i+1}$ at some stage i in $v_{|r}$. Let i be the first stage at which this happens. From the token game it follows that either ${}^{c}a_{i+1} \in Ic(p_0)$ or there exists a previous event in the run such that $a_j^c = {}^{c}a_{i+1}$. If ${}^{c}a_{i+1} \in Ic(p_0)$ then Proposition 4.3.1 is contradicted since $Ic(p_0)$ would contain two different control conditions that share the same index. In fact ${}^{c}a_1 \neq {}^{c}a_{i+1}$ (Proposition 4.4.4) and a_1, a_{i_1} have both the same index. If, instead, $a_j^c = {}^{c}a_{i+1}$ where a_j is an event that precedes a_{i+1} in the run then ${}^{c}a_{j+1} = {}^{c}a_{i+1}$ which contradicts Proposition 4.4.4 since $j < i$. □

Theorem 8.2.6 *The sequence v is an event linearisation of a bundle of $S(p_0)$.*

Proof. Given v the sequence of events in a finite and proper run of $Net(p_0)$ the following proof explicitly constructs a bundle that has v as one of its linearisations.

Let $I \subseteq I(p_0)$ such that

- for every $u \in I$ the subsequence $v_{|index(u)}$ is a non-empty prefix of u,
- for all $i \in \{index(e) \mid e \in v\}$ there exists a unique $u \in I$ with prefix $v_{|i}$ where $index(e)$ is the index of e.

As Proposition 8.2.5 says, we can find a set with those properties for every sequence of events in a run in the SPL-net. Consider the control graph

$$G(I, v) = (E, \rightarrow, act)$$

with

- nodes $E = \{(u, j) \mid u \in I \text{ and } 1 \leq j \leq |v_{|i}|\}$,
- communication edges $(w, h) \rightarrow (u, j)$ for every $(u, j) \in E$ such that $u[j]$ is an input event and where $w[h]$ is the first event in v such that $w[h]^o =^o u[j]$, and
- actions $act(u, j) = act(u[j])$.

The $\rightarrow$ relation among nodes of the graph is well defined. Since v is a sequence of events in a run in the SPL-net starting from the configuration $\langle p_0, s_0, \emptyset \rangle$, for every input event of a message in the run there exists a previous output event that marks that message (see Output-input Principle 5.1.5).

The sequence v is an event-linearisation of $G(I, v)$. The set

$$\{u[j] \mid (u, j) \in E\}$$

contains exactly all the events that appear in v. Suppose that $(w, h) \rightarrow (u, j)$. The event $w[h]$ is the first event in v that marks ${}^{o}u[j]$ and therefore it has to precede $u[j]$ in v. Suppose that $(w, h) \Rightarrow (u, j)$. There is a subsequence $v_{|i}$ of v such that

$$v_{|i} = \cdots w[h] u[j] \cdots$$

and so $w[h]$ precedes $u[j]$ in v.

It remains to show that the control graph $G(I,v)$ is a bundle of $S(p_0)$. The set I is a finite set of sequences therefore $G(I,v)$ is finite. Moreover $I \subseteq I(p_0)$, making $G(I,v)$ a subgraph of the strand-space graph of $S(p_0)$. We need now to check the other requirements of the definition of bundle (see Definition 7.2.2):

(i) Control precedence holds since $G(I,v)$ is a control graph.

(ii) Output-input precedence follows from the definition of $G(I,v)$.

(iii) Freshness holds. Let (u,j) and (w,h) be two nodes in the bundle such that

$$act(u[j]) = out\, new\, \vec{n}\ M \quad \text{and} \quad n \in \vec{n} \cap names(w[h]) \quad .$$

Suppose $(u,j) \not\Rightarrow^* (w,h)$. As for any event of $Ev(p_0)$ so for $w[h]$

$$\text{if } n \in names(w[h]) \text{ then } n \in w[h]^n \cup names({}^{c}w[h]) \cup names({}^{c}w[h]) \ .$$

From the Freshness Principle 5.1.3 of the SPL-net it follows that $n \notin w[h]^n$. If $n \in names({}^{c}w[h])$ then $w[h]$ is an input event and clearly

$$(w,h) \Rightarrow^* (w,h) \ .$$

If $n \in names({}^{c}w[h])$ then there must be an event $e \in v$ with the same index as $w[h]$ such that $n \in names(e^c)$. Let e be the first such event in v. The event $u[j]$ is in v, sequence of events of a run from configuration $\langle p_0, s_0, \emptyset \rangle$, therefore $n \notin s_0$ and $n \notin names({}^{c}e)$. Clearly

$$e \Rightarrow^* (w,h)$$

and $n \notin e^n$ (Freshness Principle 5.1.3), therefore e is an input event as desired.

(iv) Conflict freeness holds since each sequence of I has a different index.

(v) The relation $\Rightarrow \cup \rightarrow$ is acyclic. Suppose it is not. Then there is a cycle in $G(I,v)$ and therefore a finite path of $\rightarrow$ and $\Rightarrow$ edges that from a node (u,i) leads back to it. Given that v is an event-linearisation of $G(I,v)$, the event $u[i]$ appears more than once in v, contradicting Proposition 4.4.4.

□

Observe that v is the sequence of events in a run from an initial marking that does not include any output conditions. This is a necessary restriction when relating to strand spaces whose behaviour is in terms of bundles. It is likely that one can lift the condition on the initial marking if one related to open bundles instead (see Chapter 7).

On the other hand one can construct a run in the net of a "par" process from any event-linearisation of any bundle of the strand space of that process.

Theorem 8.2.7 *Every event-linearisation of every bundle of $S(p_0)$ is the sequence of events of a run in $Net(p_0)$.*

Proof. Inductively on the number of nodes in a bundle of $S(p_0)$.

A bundle is a control graph, therefore if it has only one node it has to be a node of the form $(u, 1)$ where $u \in I(p_0)$. Its even-linearisation is $u[1]$. From the output-input precedence it follows that for a bundle with a single node

$$u[1] = i : \mathbf{Out}(out\, new\, \vec{x}\ M \,.\, p\,;\, \vec{n})$$

where i is an index, M is a message, p a process, and $\vec{n}$ are distinct names. Since $u \in I(p_0)$,

$$out\, new\, x\ M \,.\, p \in Ic(p_0) \quad \text{and} \quad \vec{n} \cap names(p_0) = \emptyset \quad .$$

Let $s_0 = names(p_0)$ then

$$\langle p_0, s_0, \emptyset\rangle \xrightarrow{u[1]} \langle p_0[\vec{x}/\vec{n}], s_0 \cup \vec{n}, \{M[\vec{x}/\vec{n}]\}\rangle$$

is a proper run in $Net(p_0)$. Note that the occurrence of the event $u[1]$ does not cause contact (see Proposition 4.4.3).

Consider a bundle of $S(p_0)$ with $n \geq 2$ nodes and $e_1 \cdots e_{n-1}e_n$ one of its event-linearisations. The sequence $e_1 \cdots e_{n-1}$ is an event-linearisation of a bundle of $S(p_0)$ with $n - 1$ nodes (see Proposition 8.2.4). By the induction hypothesis, there exists a run

$$\langle p_0, s_0, t_0\rangle \xrightarrow{e_1} \cdots \xrightarrow{e_{n-1}} \langle p_{n-2}, s_{n-2}, t_{n-2}\rangle$$

in $Net(p_0)$. From the definition of $I(p_0)$ it follows that either ${}^{c}e_n \subseteq Ic(p_0)$ or $e_j^c =^c e_n$ for some preceding event e_j in the event linearisation.

We show that

$${}^{c}e_n \subseteq Ic(p_{n-2}) \quad .$$

Suppose the contrary. The sequence $e_1 \cdots e_{n-1}$ are the events of a run in $Net(p_0)$ therefore if ${}^{c}e_n \not\subseteq Ic(p_{n-2})$ there exists an event e_h with $h < n$ such that ${}^{c}e_h =^c e_n$. Clearly $e_h \neq e_n$ since they both belong to a linearisation of a bundle. The events e_h, e_n have the same indices thus belong to the same sequence in $I(p_0)$. This, however, contradicts Proposition 8.2.1.

The event e_n is either an input or an output event. Let

$$e_n = i : \mathbf{Out}(out\, new\, \vec{x}\ M \,.\, p\,;\, \vec{n}) \ .$$

We need to show that $\vec{n} \cap s_{n-2} = \emptyset$. Suppose that $\vec{n} \cap s_{n-2} \neq \emptyset$. The events in the sequence $e_1 \cdots e_{n-1}$ are the events of a proper run in $Net(p_0)$. Either $\vec{n} \cap s_0 \neq \emptyset$ which can't be the case since e_n belongs to a sequence in $I(p_0)$ or $\vec{n} \cap e_j^n \neq \emptyset$ where e_j is in $e_1 \cdots e_{n-1}$. In this second case the bundle would contain $e_j \neq e_n$ such that $e_j^n \cap e_n^n \neq \emptyset$, contradicting Proposition 7.2.4, and so contradicts the freshness condition of the bundle. Therefore the event e_n is enabled at $\langle p_{n-2}, s_{n-2}, t_{n-2}\rangle$ and does not cause contact (see Proposition 4.4.3). The sequence of transitions

$$\langle p_0, s_0, t_0\rangle \xrightarrow{e_1} \cdots \xrightarrow{e_{n-1}} \langle p_{n-2}, s_{n-2}, t_{n-2}\rangle \xrightarrow{e_n} \langle p_{n-1}, s_{n-1}, t_{n-1}\rangle$$

is a proper run in $Net(p_0)$. If instead

$$e_n = i : \mathbf{In}(input\, \vec{x}\vec{\psi}\ M \,.\, p\,;\, \vec{n}, \vec{L})$$

it follows form the definitions of bundle and linearisation of a bundle that there exists a preceding event e_j in the sequence $e_1 \cdots e_n$ such that $e_j^o = ^o e_n$. Therefore $^o e_n \subseteq t_{n-2}$ and the event e_n is enabled at configuration $\langle p_{n-2}, s_{n-2}, t_{n-2}\rangle$ (and does not cause contact – see Proposition 4.4.3), so

$$\langle p_0, s_0, t_0\rangle \xrightarrow{e_1} \cdots \xrightarrow{e_{n-1}} \langle p_{n-2}, s_{n-2}, t_{n-2}\rangle \xrightarrow{e_n} \langle p_{n-1}, s_{n-1}, t_{n-1}\rangle$$

is a run in $Net(p_0)$. □

We related the net-behaviour of a "par" process with that of a strand space with conflict. Conflict can be eliminated obtaining a traditional strand space whenever enough replication is at hand (see Section 7.7). There is enough replication in "!par" processes which therefore can be related to strand spaces of the traditional kind.

Corollary 8.2.8 *Let p be a "par" process and let s be a set of names containing all names in p. Let $S(!p) = (\langle s_u\rangle_{u\in I(!p)}, \#)$ be the strand space obtained from the process term $!p$.*

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

Proof.

1. If v the sequence of events of a finite run in $Net(!p)$ then it is an event-linearisation of a bundle b of $S(!p)$ (Theorem 8.2.6). Clearly b is also a bundle of $(\langle s_u\rangle_{u\in I(!p)}, \emptyset)$.
2. Let $S(p) = (\langle s_v\rangle_{v\in I(p)}, \#')$ There exists a re-indexing π such that
$$S(!p) \;\cong\; !S(p) \quad .$$
If b is a bundle of $(\langle s_u\rangle_{u\in I(!p)}, \emptyset)$ then the graph $\pi(b)$ is a bundle of $!(\langle s_v\rangle_{v\in I(p)}, \emptyset)$. Form Corollary 7.7.3 it follows that there is a further re-indexing π' such that $\pi'\pi(b)$ is a bundle of the strand space with conflict $!S(p)$, thus yielding $\pi\pi'\pi(b)$ a bundle of $S(!p)$. By Theorem 8.2.7 any of the event-linearisations of $\pi\pi'\pi(b)$ is a sequence of events of a run in $Net(!p)$.

□

8.2.4 Relating SPL-transition semantics and strand spaces

More traditional process-language models such as transition systems are strongly related to the strand space model for security protocols. From the discussion of the previous section (Corollary 8.2.8) and the relation that exists between the transition semantics of SPL and its event-based semantics (Theorem 4.4.1) it follows that:

Corollary 8.2.9 *Let p be a "par" process and let s be a set of names containing all names in p. Let $S(!p) = (\langle s_u\rangle_{u\in I(!p)}, \#)$ be the strand space obtained from the process term $!p$.*

- *The sequence of actions in a finite transition sequence from $\langle !p, s, \emptyset\rangle$, a proper configuration, is the sequence of actions in an event-linearisation of a bundle in $(\langle s_u\rangle_{u\in I(!p)}, \emptyset)$.*
- *Every bundle of $(\langle s_u\rangle_{u\in I(!p)}, \emptyset)$ can be re-indexed so that the sequence of actions in any of its event-linearisations is that of a proper sequence of transitions of p.*

$\square$

8.3 Inductive rules from SPL

Paulson's inductive rules for a security protocol capture the actions it and a spy can perform [61, 62]. Through allowing persistent conditions, we can represent a collection of inductive rules as a net in which the events stand for rule instances and runs for sequences of rule instances which form a derivation from the rules. In particular, instances of inductive rules for security protocols can be represented as events in a net for which all but the name conditions are persistent. According to such a semantics, once a protocol can input it can do so repeatedly. Once it can output generating new names it can do so repeatedly, provided this doesn't lead to clashes with names already in use. Paulson's traces and the associated runs of the net will necessarily include such "stuttering". In this section we first describe how to obtain a net of rule instances from a net of an SPL process and show that they relate provided enough "replication" is introduced.

8.3.1 Rule instances from SPL

The net of SPL-rule instances has:

- *rule conditions* $\mathbf{C}' \cup \mathbf{O} \cup \mathbf{N}$ where $\mathbf{N}$ are name conditions and $\mathbf{O}$ are persistent output conditions, as before, but now with additional persistent conditions $\mathbf{C}' \subset \mathit{Contc}$ consisting of closed input and output process terms (all indexed process terms are omitted),
- *rule instances* which are all the output and input events in **Events** that do not carry indices.

Let r be a function from SPL-conditions to rule-conditions which removes the indices tagging control conditions and leaves output and name conditions unchanged. For example:

$$r(i : j : \mathit{out\,new}\ M . p) = \mathit{out\,new}\ M . p \quad .$$

Extend r to SPL events: let r replace all the control conditions of an SPL-event by their images under r – intuitively, an event e is replaced by a rule instance $r(e)$:

$$r({}^{c}e, {}^{o}e, {}^{n}e, e^{c}, e^{o}, e^{n}) = (r({}^{c}e), {}^{o}e, {}^{n}e, r(e^{c}), e^{o}, e^{n}) \quad .$$

The following properties clearly hold for the function r:

Proposition 8.3.1

1. *Given two sets of conditions $A, B \subseteq \mathbf{C}$, $r(A \cup B) = r(A) \cup r(B)$.*

*2. Given an event $e \in$ **Events**, $r({}^c e) = {}^c r(e)$ and $r(e^c) = r(e)^c$.*

□

The *net of rule instances $R(p)$* of a closed process term p is the net with rule conditions and events $rEv(p)$. For a closed process term p let p^* be the process term obtained by inserting a replication before every input and output process sub-term in p. For example

$$(in\ M\,.\,out\,new\,x\ N) \ = !\,in\ M\,.!\,out\,new\,x\ N \quad .$$

Proposition 8.3.2 *If p is a closed process term then $R(p^*) = R(p)$.*

Proof. Inductively on the structure of p. □

8.3.2 Relating SPL-net semantics and rule instances

The replication in front of each input and output process sub-term ensures that at each stage of a run of a process p^* each control condition is marked infinitely many times though carrying each time a different index – in this sense control conditions become persistent. The following lemma explains this more formally:

Lemma 8.3.3 *Let $\mathcal{M}_i$ be a marking in a run in $Net(p^*)$. For every finite subset C of the control conditions marked in $\mathcal{M}_i$*

$$r(\mathcal{M}_i \setminus C) = r(\mathcal{M}_i) \quad .$$

Proof. The control conditions marked in $\mathcal{M}_0$ are

$$Ic(p^*) = \bigcup_{i \in \omega} i : B$$

where B is a set of control conditions. Therefore $r(Ic(p^*)) = r(B)$. Since C is a finite subset of $Ic(p^*)$ there exists $j \in \omega$ such that $j : B \cap C = \emptyset$ and clearly $r(Ic(p^*) \setminus C) = r(B)$.

Consider

$$\cdots \mathcal{M}_i \xrightarrow{e_{i+1}} \mathcal{M}_{i+1} \quad .$$

It follows from the token game that if B are the control conditions of $\mathcal{M}_i$ then the ones of $\mathcal{M}_{i+1}$ are

$$(B \setminus {}^c e_{i+1}) \cup \bigcup_{i \in \omega} i : B'$$

where B' is a set of control conditions. From the induction hypothesis it follows that

$$r(B \setminus {}^c e_{i+1}) = r(B \setminus ({}^c e_{i+1} \cup C)) = r(B)$$

and clearly

$$r(\bigcup_{i \in \omega} i : B') = r(\bigcup_{i \in \omega} i : B' \setminus C) = r(B') \quad .$$

Therefore $r(\mathcal{M}_{i+1} \setminus C) = r(\mathcal{M}_{i+1})$. □

Now having restricted to a process with sufficient replication we can establish a close relation between the behaviours of $Net(p^*)$ and $R(p^*)$.

Theorem 8.3.4 *Let p be a close process term.*

- *A proper run*
$$\mathcal{M}_0 \xrightarrow{e_1} \cdots \xrightarrow{e_n} \mathcal{M}_n$$
in $Net(p^*)$ yields a run
$$r\mathcal{M}_0 \xrightarrow{r(e_1)} \cdots \xrightarrow{r(e_n)} r\mathcal{M}_n$$
in $R(p^*)$.

- *To a run in $R(p^*)$*
$$\mathcal{M}'_0 \xrightarrow{e'_1} \cdots \xrightarrow{e'_n} \mathcal{M}'_n$$
with $\mathcal{M}'_0 = r\mathcal{M}_0$ proper, there is a run
$$\mathcal{M}_0 \xrightarrow{e_1} \cdots \xrightarrow{e_n} \mathcal{M}_n$$
in $Net(p^*)$, with $r(e_i) = e'_i$ and $r\mathcal{M}_i = \mathcal{M}'_i$ for all $0 < i \leq n$.

Proof.

- By induction on the length of a run in $Net(p^*)$. Suppose a finite run
$$\mathcal{M}_0 \xrightarrow{e_1} \cdots \xrightarrow{e_n} \mathcal{M}_n \xrightarrow{e_{i+1}} (\mathcal{M}_n \setminus {}^c e_{n+1}) \cup e^{\cdot}_{n+1}$$
in $Net(p^*)$ and suppose a corresponding run
$$r\mathcal{M}_0 \xrightarrow{r(e_1)} \cdots \xrightarrow{r(e_n)} r\mathcal{M}_n$$
in $R(p^*)$. Clearly $r({}^c e_{n+1}) \subseteq r\mathcal{M}_n$ and given that $r({}^c e_{n+1}) ={}^c (r(e_{n+1}))$ the event $r(e_{n+1})$ is enabled yielding
$$r\mathcal{M}_0 \xrightarrow{r(e_1)} \cdots \xrightarrow{r(e_n)} r\mathcal{M}_n \xrightarrow{r(e_{n+1})} r\mathcal{M}_n \cup r(e_{n+1})^{\cdot}$$
in $R(p^*)$ (there is no contact at control conditions since in $R(p^*)$ control conditions are persistent). From Lemma 8.3.3 and Proposition 8.3.1 it follows that
$$r((\mathcal{M}_n \setminus {}^c e_{n+1}) \cup e^{\cdot}_{n+1}) = r\mathcal{M}_n \cup r(e_{n+1})^{\cdot} \quad .$$

- By induction on the length of a run in $R(p^*)$. Assume a finite run
$$\mathcal{M}'_0 \xrightarrow{e'_1} \cdots \xrightarrow{e'_n} \mathcal{M}'_n \xrightarrow{e'_{n+1}} \mathcal{M}'_n \cup e'^{\cdot}_{n+1}$$
in $R(p^*)$ with $\mathcal{M}'_0 = r\mathcal{M}_0$ and assume a corresponding run
$$\mathcal{M}_0 \xrightarrow{e_1} \cdots \xrightarrow{e_n} \mathcal{M}_n$$
in $Net(p^*)$ with $\mathcal{M}'_i = r\mathcal{M}_i$ and $e'_i = r(e_i)$ for all $0 < i \leq n$. Let $\{c'\} ={}^c e'_{n+1}$. The control condition c' is in $r\mathcal{M}'_n$ therefore there exists a control condition c such that $r(c) = c'$ and $c \in \mathcal{M}_n$. Therefore there exists an event $e \in Ev(p^*)$

such that $c =^c e$ (From the token game and Proposition 4.3.5, $c \in^c p^*$. Apply Proposition 4.3.6). From Proposition 4.3.3 it follows that all the events in **Events** with c as control precondition belong to $Ev(p^*)$. In particular there is an event $e_{n+1} \in Ev(p^*)$ such that $r(e_{n+1}) = e'_{n+1}$. The event e_{n+1} is clearly enabled at the marking $\mathcal{M}_n$ yielding

$$\mathcal{M}_0 \xrightarrow{e_1} \cdots \xrightarrow{e_n} \mathcal{M}_n \xrightarrow{e_{n+1}} (\mathcal{M}_n \setminus {}^c e_{n+1}) \cup e^{\cdot}_{n+1}$$

in $R(p^*)$ (by Proposition 4.4.3 there is no contact at control conditions). As before it is easy to check that $r\mathcal{M}_{n+1} = \mathcal{M}'_{n+1}$.

□

8.4 Other models from SPL

Because strand spaces can easily be turned into event structures, Sections 8.2 and 8.1 yield an event structure for each "!-par" process. But, without any restrictions, we can relate the net semantics to traditional independence models such as event structures and Mazurkiewicz trace languages. The crux of the construction is that of eliminating the persistent conditions from the $Net(p)$, of a closed process term p, in an initial marking of control conditions $Ic(p)$ to produce a basic net. In Chapter 3 we have described how to unfold a net with persistent conditions to one of the basic kind. Here we specialise that construction and the correctness result to $Net(p)$ – we first unfold to a coloured net and then to a basic net. It is well-known how to unfold a basic net to a Mazurkiewicz trace language and event structure [97].

Recall that

$${}^{\cdot}b = \{e \mid b \in e^{\cdot}\} \cup \{*\}$$

and that

$$b^{\cdot} = \{e \mid b \in {}^{\cdot}e\} \ .$$

Write In for the set of input events in $Net(p)$ and Out for the set of output events in $Net(p)$.

8.4.1 Coloured nets from SPL

Let p be a closed process term. Recall

$$Net(p) = (\mathbf{C} \cup \mathbf{O} \cup \mathbf{N}, \mathbf{O}, Ev(p), pre, post)$$

where pre and $post$ are pre- and postcondition maps for the SPL net as defined in Chapter 3. Its persistent conditions are $\mathbf{O}$ and its start in an initial marking with control conditions $Ic(p)$. The net $Net(p)$ unfolds to the coloured net:

$$(\mathbf{C} \cup \mathbf{O} \cup \mathbf{N}, Ev(p), \Delta, pre, post) \quad .$$

The colouring function Δ associates colours to events and conditions. One can attach simpler colours to events than the colours attached to events in the more general construction. The only events with persistent preconditions are input events, which carry exactly one persistent condition, therefore the colouring function Δ on events is simplified to the following:

$$\Delta(e) = \begin{cases} \{e' \mid e'^{o} = {}^{o}e\} \cup \{*\} & \text{if } e \in In \\ \{\delta\} & \text{if } e \in Out \end{cases} .$$

Recall that δ is the default colour. Conditions are coloured as following:

$$\Delta(b) = \begin{cases} {}^{\cdot}b \times b^{\cdot} & \text{if } b \in \mathbf{O} \\ \{\delta\} & \text{otherwise} \end{cases} .$$

The pre and postcondition maps in the coloured net are as follows:

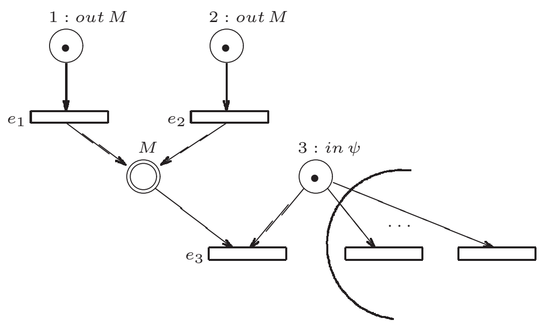
$$\begin{aligned} pre(e,c) &= (({}^{c}e \cup {}^{n}e) \times \{\delta\}) \cup ({}^{o}e \times \{(c,e)\}) \\ post(e,c) &= ((e^{c} \cup e^{n}) \times \{\delta\}) \cup (e^{o} \times \{(e,e') \mid e^{o} = {}^{o}e'\}) \end{aligned} .$$

where (e,c) are pairs of events e and colours $c \in \Delta(e)$. The pre and postcondition maps specialises those of the general construction in Chapter 3 where the postcondition map is slightly simpler: postconditions do not include the coloured preconditions of an event. Events never occur more than once in a run of $Net(p)$ and therefore coloured conditions obtained from the unfolding of persistent conditions need to hold only once for each event that requires them as preconditions.

As an example of the construction of a coloured net from a closed SPL process, consider the process:

$$out\ M \parallel out\ M \parallel in\ \psi$$

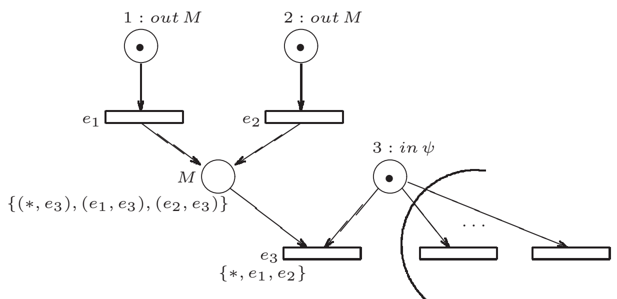
and its net with persistent conditions (where we draw only the relevant conditions) in an initial marking:



In the net above

$$\begin{aligned} e_1 &= 1 : \mathbf{Out}(out\, M) \\ e_2 &= 2 : \mathbf{Out}(out\, M) \\ e_3 &= 3 : \mathbf{In}(in\, \psi; M) \end{aligned} .$$

The net unfolds to the following coloured net:



where the set of colours of events and conditions is indicated when it is not the default colour δ.

8.4.2 Basic nets from SPL

The coloured net that we just obtained from $Net(p)$ unfolds in the usual way yielding the basic net

$$\mathcal{N}(p) = (B, E, pre, post)$$

with conditions

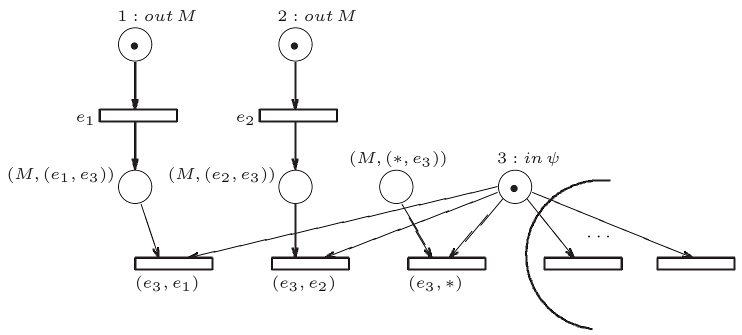
$$B = \bigcup_{b \in B} \{b\} \times \Delta(b)$$

and events

$$E = Out \times \{\delta\} \cup \{(e, *) \mid e \in In\} \cup \{(e, e') \mid e \in In \;\&\; e' \in Out \;\&\; e'^{o} =^{o} e\} \quad .$$

Pre- and postcondition maps are as before. Assume that this net starts in an initial marking whose control conditions are $Ic(p)$, the same to those of the net with persistent conditions $Net(p)$.

When we apply this unfolding to the example above we obtain the basic net:



Define the map $\sigma : E \to Out \cup In$ that leaves events in Out unchanged and project pairs $(e, *)$, (e, e') to the component e.

Corollary 8.4.1

1. *If $e_1 \cdots e_k$ is the event sequence of a run in $Net(p)$ from the initial marking $Ic(p) \cup s \cup t$, where $s \subseteq \mathbf{N}$ and $t \subseteq \mathbf{O}$, then the net $\mathcal{N}(p)$ has a run with event sequence $e'_1 \cdots e'_k$ where $e_1 \cdots e_k = \sigma(e'_1) \cdots \sigma(e'_k)$.*

2. *If $\mathcal{N}(p)$ has a run with event sequence $e'_1 \cdots e'_k$ then there is a run of $Net(p)$ with the event sequence $\sigma(e'_1) \cdots \sigma(e'_k)$.*

Proof. From Proposition 4.4.4 events never occur more than once in a run of $Net(p)$ therefore Theorem 3.5.1 applies. □

8.4.3 Mazurkiewicz trace languages from SPL

A Mazurkiewicz trace language (see [50, 97]) is a language in which the alphabet has a relation of independence.

Definition 8.4.2 A Mazurkiewicz trace language consists of

$$(M, L, I)$$

where L is a set, $I \subseteq L \times L$ is a symmetric, irreflexive relation called the independence relation, and M is a nonempty subset of strings L^* such that

(i) prefix closed: $\forall u \in L^* . \forall a \in L$ if $ua \in M$ then $u \in M$,

(ii) I closed: $\forall u, v \in L^* . \forall a, b \in L$ if $uabv \in M$ and aIb then $ubav \in M$,

(iii) coherent: $\forall u \in L^* . \forall a, b \in L$ if $ua \in M$ and $ub \in M$ and aIb then $uab \in M$.

□

It is well known that the sequences of events of runs of a basic net form a Mazurkiewicz trace language (see [97]). In particular this is the case for the basic net $\mathcal{N}(p)$ obtained from the net with persistent conditions $Net(p)$ of a closed process term p. More precisely define

$$T(p) = (R, E, I)$$

where

- R is the set of event sequences of proper and finite runs in $\mathcal{N}(p)$,
- E is the set of events of $\mathcal{N}(p)$, and
- $I \subseteq E \times E$ is such that eIe' iff ${}^{\cdot}e^{\cdot} \cap {}^{\cdot}e'^{\cdot} = \emptyset$.

Corollary 8.4.3 *Let p be a closed process term then $T(p)$ is a Mazurkiewicz trace language.*

Proof. Instance of the result in [97]. □

We have seen previously how to obtain a basic net from a net with persistent conditions of an SPL process. Corollary 8.4.1 and Corollary 8.4.3 yield:

Corollary 8.4.4 *Given a closed process term p there exists a Mazurkiewicz trace language (M, L, I) such that*

1. *If $e_1 \cdots e_k$ is the sequence of the events of a run in $Net(p)$ then there is a string of events $e'_1 \cdots e'_k \in M$ such that $e_1 \cdots e_k = \sigma(e'_1) \cdots \sigma(e'_k)$.*

2. *If $e'_1 \cdots e'_k \in M$ then there is a run of $Net(p)$ with event sequence $\sigma(e'_1) \cdots \sigma(e'_k)$.*

$\square$

Let (M, L, I) be a Mazurkiewicz trace language. Define the relation $\backsimeq$ among strings of the language to be the smallest equivalence relation such that

$$uabv \backsimeq ubav \text{ if } aIb$$

for $uabv, ubav \in M$. An equivalence class $\{u\}_{\backsimeq}$ for $u \in M$ is called a *trace*. For $u, v \in M$ define

$$u \lesssim v \text{ iff } \exists w \,.\, uw \backsimeq v$$

It turns out that (see [97]) the quotient $\lesssim / \backsimeq$ is a partial order on traces.

8.4.4 Event structures from SPL

The partial order of a trace language that we defined previously is associated to a partial order of casual dependency among events in an event structure. In the this section we use this fact to obtain an event structure from any closed SPL process term. The event structure that one obtains is a prime event structure, the kind of event structure that we introduced earlier in Section 8.1.1.

Firstly we define a notion of *events* of a Mazurkiewicz trace language $T(p)$ for a closed process term p. Let R be the strings of $T(p)$. Following [97], events are taken to be the equivalence classes with respect to $\sim$, the smallest equivalence relation on nonempty strings such that for $ue, ue'e, ve \in R$

1. $ue \sim ue'e$ if ${}^{\cdot}e'^{\cdot} \cap {}^{\cdot}e^{\cdot} = \emptyset$ and
2. $ue \sim ve$ if $u \backsimeq v$.

Write

$$ev(u) = \{\{v\}_{\sim} \mid v \text{ nonempty prefix of } u\}$$

and define the following event structure:

$$tle(T(p)) = (\mathcal{E}, \leq, \#)$$

where

- $\mathcal{E} = \{\{u\}_{\sim} \mid u \in R\}$
- $e \leq e'$ iff $\forall u \in R \,.\, e' \in ev(u) \Rightarrow e \in ev(u)$
- $e\#e'$ iff $\forall u \in R \,.\, e \in ev(u) \Rightarrow e' \notin ev(u)$

The representation theorem for trace languages into event structures shown in [97] can be specialised to one for trace languages of a closed process term of SPL.

Corollary 8.4.5 *Let p be a closed process term and $T(p)$ trace language associated with it with strings R. There is an order isomorphism*

$$\phi : (R_{|\backsimeq}, \lesssim /\backsimeq) \cong (\mathcal{F}^{fin} tle(T(p)), \subseteq)$$

where $\phi(\{u\}_{\backsimeq}) = ev(u)$.

Proof. Special case of the result presented in [97]. □

Corollary 8.4.1 and Corollary 8.4.5 clearly yield:

Corollary 8.4.6 *For a closed process term p the prime event structure $tle(T(p))$ is such that*

1. *If $e_1 \cdots e_k$ is the sequence of events of a run in $Net(p)$ then there is a run in the basic net $\mathcal{N}(p)$ with event sequence $e'_1 \cdots e'_k$ such that $e_1 \cdots e_k = \sigma(e'_1) \cdots \sigma(e'_k)$. The set $ev(e'_1 \cdots e'_k)$ is a configuration of $tle(T(p))$.*

2. *For every finite configuration x of $tle(T(p))$ there exists a run in the basic net $\mathcal{N}(p)$ with event sequence $e'_1 \cdots, e'_k$ such that $ev(e'_1 \cdots e'_k) = x$. The sequence $\sigma(e'_1) \cdots \sigma(e'_k)$ is the event sequence of a run in $Net(p)$.*

□

Chapter 9

Conclusion

Communication security is becoming central today, at a time when computation becomes more and more a collaboration between distributed entities. Not all systems have the same security goals. A number of methodologies have been developed to achieve different degrees of security and many new and better techniques are to come. In deciding which security measures to adopt one needs assurance that the "bought" technology indeed does achieve the desired security goals. Equally important is a precise description of scenarios where things can go wrong and attacks can succeed. Informal arguments too often give wrong assurances and therefore are not enough. Formal methods attempt to make security guarantees precise. Formal results, however have sometimes been misunderstood; it can happen that a protocol studied in one formal model appears to be secure while an attack can be easily found in a different model. Little research has been done so far to compare different formal methods for security and more needs to be done in making formal reasoning for security easier and more accessible. One reason for a certain reluctance toward the formal-methods approach to security is the gap that often exists between a formal model and the actual implementations of security mechanisms. It is not always clear whether or not a formal property transfers to a particular implementation.

This thesis contributes to the formal methods approach for security protocols. The thesis can be divided into two parts. The first chapters introduce a new language and semantics for security protocols. The Petri-net semantics of the language SPL captures properties that aid formal reasoning about the security of a protocol. A number of examples underpin this argument. The Petri-net model of security protocols is shown to be closely related to a more traditional transition semantics which can easily be implemented.

We have implemented SPL as a prototype language for security protocols. Ideally, if our implementation is correct, the properties that hold for the Petri-net model of a protocol are properties of the "running" protocol. We did not report about our prototype implementation in this thesis but point to some published papers instead [13, 21].

The second part of the thesis shows how our approach relates to a number of other methods for security-protocol verification. Connections are established also to more traditional, general purpose models for concurrency. These relations give new insights that suggest how to improve special purpose methods for security protocols, for example by turning them into compositional models. We hope that this part of our research can contribute to adapting more established verification techniques such as those used for verifying properties of Petri-nets to the analysis of security protocols – the hope is to make reasoning simpler and protocol checkers more automatic.

The following are some directions for future work:

Language extensions. The SPL language is very simple and contains only few constructs. Despite its minimality, it suffices to describe numerous security protocols that are important building blocks of industrial-strength protocols. To be able to program an entire, more complex, protocol some extensions to the language are helpful. Message expressions could be extended so that one can describe other cryptographic functions such as hash functions. A couple of basic message operations like message addition should also be possible. Recursive process definitions would be useful to model recursively defined protocols (see for example [60] or group key-exchange protocols [9]). We used input patterns to perform positive tests on the network contents and negative tests on names are handled by not allowing contact on name conditions. Special non-persistent message conditions could be introduced to have negative tests on messages. We expect the language extensions that we mentioned to be easy and not to substantially affect the net semantics of SPL.

More on security properties. Other properties of protocols can be studied using SPL and its event-based semantics. We started to investigate anonymity properties which can conveniently be described on the sequences of events describing protocol runs. We are currently investigating methods and principles that help in proving anonymity properties.

Proof methods based on equivalence relations. We have introduced a congruence relation on strand spaces – two strand spaces are equivalent if they have substantially the same open bundles. It would be interesting to try to make use of the relation to show security properties along the lines of what has initially been done in proving properties of protocols modelled as Spi-calculus processes [5]. The congruence relation on strand spaces appears however to be too fine grained. To turn it into a more useful relation one might have to introduce a "hiding" to hide those actions of protocols which should be internalised when analysing their security.

Logics. We hope our work helps towards a more high-level analysis of security protocols. To this end, we see the net semantics of SPL as giving a potentially useful, concrete model theory for logics for security protocols. The net runs are histories on which to interpret security properties, perhaps expressed in the style of BAN logic. On another tack, Petri nets form models of linear logic [29], close it seems to the linear logic for security protocols based on multiset rewriting [15].

More on unifying research. A role of the language SPL, is that it can support, and so help relate, different semantics useful in the analysis of security protocols - we have seen several examples. Many more security protocol methods exists and therefore close relations between SPL and other methods might exist. For instance, the "authentication by typing" approach of Gordon and Jeffrey [34] annotates the syntax of the Spi calculus process to mark events of a protocol in correspondence assertion. This seems close to what is done in our and other event-based approaches. Future goals are to relate to a probabilistic semantics, moving away from the perfect cryptography assumption, and to study what equivalences and compositional reasoning fit with the rather intensional event-based methods dealt with here.

Bibliography

[1] M. Abadi. Secrecy by typing in security protocols. In *Proceedings of Theoretical Aspects of Computer Software, Third International Symposioum*, volume 1281 of *LNCS*, pages 611–638. Springer-Verlag, 1997.

[2] M. Abadi. Two Facets of Authentication. In *Proceedings of the 11th IEEE Computer Security Foundations Workshop*, pages 27–32. IEEE Computer Society Press, 1998.

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

[4] M. Abadi, C. Fournet, and G. Gonthier. Secure Communications Processing for Distributed Languages. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*. IEEE Press, May 1999.

[5] M. Abadi and A. D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. *Information and Computation*, 148:1–70, 1999.

[6] M. Abadi and R. Needham. Prudent Engineering Practice for Cryptographic Protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, 1996.

[7] M. Abadi and M. R. Tuttle. A Semantics for a Logic of Authentication. In *Proceedings of the Symposium on Principles of Distributed Computing*. ACM, 1991.

[8] *Advanced course on Petri nets*, volume 254,255 of *LNCS*. Springer-Verlag, 1987.

[9] J. Alves-Foss. An Efficient Secure Authenticated Group Key Exchange Algorithm for Large and Dynamic Groups. In *Proceedings of the 23rd National Information Systems Security Conference*, 2000.

[10] M. Barjaktanovic, C. Shiu-Kai, C. Hosmer, D. Rosenthal, M. Stillman, G. Hird, and D. Zhou. Analysis and Implementation of Secure Electronic Mail Protocols. In *Proceedings of the Workshop on design and formal verification of security protocols*, 1997.

[11] G. Bella. *Inductive Verification of Criptographic Protocols*. PhD thesis, Clare College, University of Cambridge, 2000.

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

[13] M. Cáccamo, F. Crazzolara, and G. Milicia. The ISO 5-pass authentication in χ-Spaces. In *Proceedings of the Security and Management Conference (SAM)*, pages 490–495, Las Vegas, June 2002.

[14] L. Cardelli. Mobility and security. Lecture notes, Marktoberdorf Summer School, 1999.

[15] I. Cervesato, N. Durgin, M. Kanovich, and A. Scedrov. Interpreting Strands in Linear Logic. In *In Proceedings of FMCS'00*, 2000.

[16] I. Cervesato, N. A. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov. A Meta-notation for Protocol Analysis. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, pages 55–69. IEEE Computer Society Press, June 1999.

[17] I. Cervesato, N. A. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov. Relating Strands and Multiset Rewriting for Security Protocol Analysis. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, July 2000.

[18] S. Christensen and N. D. Hansen. Coloured Petri Nets Extended with Place Capacities, Test Arcs and Inhibitor Arcs. In *Application and Theory of Petri nets, 14th International Conference*, volume 691 of *LNCS*, pages 187–206. Springer-Verlag, 1993.

[19] J. A. Clark and J. L. Jacob. A survey of authentication protocol literature. Technical Report 1.0, 1997.

[20] E. M. Clarke, S. Jha, and W. Marrero. Using State Space Exploration and a Natural Deduction Style Message Derivation Engine to Verify Security Protocols. In *Proceedings of the IFIP Working Conference on Programming Concepts and Methods*, 1998.

[21] F. Crazzolara and G. Milicia. Developing Security Protocols in χ-Spaces. In *Proceedings of the 7th Nordic Workshop on Secure IT Systems (NordSec)*, Karlstad, Sweden, November 2002.

[22] F. Crazzolara and G. Winskel. Events in security protocols. In *Proceedings of the Eight ACM Conference on Computer and Communications Security,* Philadelphia, November 2001. ACM Press.

[23] F. Crazzolara and G. Winskel. Petri nets in cryptographic protocols. In *Proceedings of the 16th International Parallel & Distributed Processing Symposium – 6th International Workshop on Formal methods for Parallel Programming: Theory and Practice*, San Francisco, April 2001. IEEE Press.

[24] F. Crazzolara and G. Winskel. Composing Strand Spaces. In *Proceedings of Foundations of Software Technology and Theoretical Computer Science*, volume 2556 of *LNCS*, pages 97–108, Kanpur, India, December 2002.

[25] G. Denker and J. Millen. CAPSL integrated protocol environment. In *DARPA Information Survivability Conference (DISCEX 2000)*, pages 207–221. IEEE Press, 2000.

[26] Data Encryption Standard (DES). FIPS Pub 46-2, National Bureau of Standards, 1993.

[27] D. Dolev, C. Dwork, and M. Naor. Non-malleable cryptography. In *Proceedings of the 23rd Symposium on Theory of Computing*, pages 542–552, 1991.

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

[29] U. Engberg and G. Winskel. Linear logic on petri nets. In *REX, a decade of concurrency*, volume 803 of *LNCS*. Springer Verlag, 1994.

[30] 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 Press, November 1983.

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

[32] J. Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[33] S. Goldwasser and M. Bellare. Lecture notes on cryptography. Summer Course "Cryptography and Computer Security" at MIT, 1996–1999, 1999.

[34] A. Gordon and A. Jeffrey. Authenticiy by Typing for Security Protocols. In *Proceedings of the 14th Computer Security Foundations Workshop*, pages 145–159. IEEE Computer Society Press, 2001.

[35] J. Guttman. Key Compromise, Strand Spaces, and the Authentication Tests. In *Proceedings of Mathematical Foundations of Programming Semantics*, volume 17 of *Electronic Notes in Theoretical Computer Science*, 2001.

[36] J. Guttman. Security Protocol Design via Authentication Tests. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 2002.

[37] J. Y. Halpern and R. Pucella. On the relationship between strand spaces and multi-agent systems. In *Proceedings of the Eight ACM Conference on Computer and Communications Security*, Philadelphia, November 2001. ACM Press.

[38] J. Heather. Strand Spaces and Rank Functions: More than Distant Cousins. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 2002.

[39] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[40] ISO/IEC. Information Technology. *Security techniques – Entity Authentication Mechanisms Part 2: Entity authentication using symmetric techniques*, 1993.

[41] ITU-TS. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC).* ITU-TS, Geneva, 1997.

[42] K. Jensen. Coloured Petri nets and the invariant method. *TCS*, 14, 1981.

[43] C. Lakos and S. Christensen. A General Systematic Approach to Arc Extensions for Coloured Petri Nets. In *Application and Theory of Petri nets, 15th International Conference*, volume 815 of *LNCS*, pages 338–355. Springer-Verlag, 1994.

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

[45] P. D. Lincoln, J. C. Mitchell, M. Mitchell, and A. Scedrov. A Probabilistic Poly-time Framework for Protocol Analysis. In *Proceedings of the Fifth ACM Conference on Computer and Communications Security*, 1998.

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

[47] 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.

[48] G. Lowe. A Hierarchy of Authentication Specifications. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop*, pages 31–43. IEEE Computer Society Press, 1997.

[49] W. Mao and C. Boyd. Towards the Formal Analysis of Security Protocols. In *Proceedings of the 6th Computer Security Foundations Workshop*, pages 147–158. IEEE Computer Society Press, 1993.

[50] A. Mazurkiewicz. Basic notions of trace theory. In *Linear Time, Branching Time and Partial Orders in Logics and Models for Concurrency*, volume 354 of *LNCS*, pages 285–363. Springer-Verlag, 1988.

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

[52] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applies Cryptography.* CRC Press, 1996.

[53] J. Millen and F. Muller. Cryptographic protocol generation from CAPSL. Technical Report SRI-CSL-01-07, SRI International, December 2001.

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

[55] J. C. Mitchell, M. Mitchell, and U. Stern. Automated Analysis of Cryptographic Protocols Using Murphi. In *1997 IEEE Symposium on Security and Privacy.* IEEE Press, 1997.

[56] U. Montanari and F. Rossi. Contextual nets. *Acta Informatica*, 32(6), 1995.

[57] 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.

[58] M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, Event structures and Domains. *Theoretical Computer Science*, 13, 1981.

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

[60] L. C. Paulson. Mechanized proofs for a recursive authentication protocol. In *10th Computer Security Foundations Workshop*, pages 84–95. IEEE Computer Society Press, 1997.

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

[62] L. C. Paulson. Proving Security Protocols Correct. In *Proceedings of the 14th Symposium on Logic in Computer Science*, July 1999.

[63] L. C. Paulson. Relations Between Secrets: Two Formal Analyses of the Yahalom Protocol. *Journal of Computer Security*, 9(3):197–216, 2001.

[64] 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, 1993.

[65] V. R. Pratt. Modelling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986.

[66] W. Reisig. *Petri Nets. An introduction*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.

[67] 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.

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

[69] A. W. Roscoe. Intensional specification of security protocols. In *Proceedings of the 9th IEEE Computer Security Foundations Workshop*, pages 28–38. IEEE Computer Society Press, 1996.

[70] 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.

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

[72] P. Y. A. Ryan and S. A. Schneider. Process algebra and Non-interference. In *Proceedings of the 12th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1999.

[73] S. Schneider. Modelling security protocols with CSP. Technical Report CSD-TR-96-04, Royal Holloway, University of London, 1996.

[74] S. Schneider. Security properties and CSP. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 174–187. IEEE Press, 1996.

[75] S. Schneider. Using CSP for protocol analysis: the Needham-Schroeder Public-Key Protocol. Technical Report CSD-TR-96-14, Royal Holloway, University of London, 1996.

[76] S. Schneider. Verifying authentication protocols with CSP. In *Proceedings of the 10th Computer Security Foundations Workshop*, pages 3–17. IEEE Computer Society Press, 1997.

[77] S. Schneider. Verifying authentication protocols in CSP. *IEEE TSE*, 24(9), September 1998.

[78] S. Schneider and A. Sidiropoulos. CSP and anonymity. In *ESORICS*, pages 198–218, 1996.

[79] B. Schneier. *Applied cryptography (2nd ed.): protocols, algorithms, and source code in C*. John Wiley & Sons, Inc., 1995.

[80] 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.

[81] D. Song, S. Berzin, and A. Perrig. Athena, a Novel Approach to Efficient Automatic Security Protocol Analysis. *Jurnal of Computer Security*, 9(1,2):47–74, 2001.

[82] D. Song, A. Perrig, and D. Phan. AGVI – Automatic Generation, Verification, and Implementation of Security Protocols. In *Proceedings of the 13th Conference on Computer Aided Verification (CAV)*, 2001.

[83] P. Syverson. Towards a Strand Semantics for Authentication Logic. *Electronic Notes in Theoretical Computer Science*, 20, 1999.

[84] J. Thayer and J. Guttman. Authentication tests. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, Oakland CA, May 2000.

[85] J. Thayer and J. Guttman. Protocol Independence through Disjoint Encryption. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 2000.

[86] J. Thayer and J. Guttman. Authentication Tests and the Structure of Bundles. *Theoretical Computer Science*, 283(2), 2002.

[87] J. Thayer, J. Guttman, and L. Zuck. The Faithfulness of Abstract Protocol Analysis: Message Authentication. In *Proceedings of the Eight ACM Conference on Computer and Communications Security*. ACM Press, 2001.

[88] 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.

[89] J. Thayer, J. Herzog, and J. Guttman. Strand space pictures. In *Proceedings of the Workshop on Formal Methods and Security Protocols*, Indianapolis, June 1998.

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

[91] J. Thayer, J. Herzog, and J. Guttman. Strand Spaces: Proving Security Protocol Correct. *Journal of Computer Security*, 7:191,230, 1999.

[92] G. Winskel. *Events in computation*. PhD thesis, Comp. Sc. Dept. University of Edinburgh, 1980.

[93] G. Winskel. Event structure semantics of CCS and related languages. In *Proceedings of the International Colloquium on Automata, Languages and Programming - ICALP 82*, volume 140 of *LNCS*. Springer Verlag, 1982. Extended version, Computer Science Report, University of Aarhus, 1982.

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

[95] G. Winskel. Petri Nets, Algebras, Morphisms, and Compositionality. *Information and Computation*, 72:197–238, 1987.

[96] G. Winskel. An introduction to event structures. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *LNCS*. Springer Verlag, 1988.

[97] G. Winskel and M. Nielsen. *Handbook of Logic and the Foundations of Computer Science*, volume IV, chapter Models for concurrency. Oxford University Press, 1995.

[98] T. Y. Woo and S. S. Lam. A semantic model for authentication protocols. In *RSP: IEEE Computer Society Symposium on Research in Security and Privacy*, 1993.

[99] T. Y. C. Woo and S. S. Lam. Authentication for Distributed Systems. *Computer*, 25(1):39–52, January 1992.

[100] T. Y. C. Woo and S. S. Lam. A Lesson on Authentication Protocol Design. *Operating Systems Review*, pages 24–37, 1994.