Protocol Composition Logic

Anupam DATTA [a], John C. MITCHELL [b], Arnab ROY [c] and Stephan Hyeonjun STILLER [b]

[a] *CyLab, Carnegie Mellon University*
[b] *Department of Computer Science, Stanford University*
[c] *IBM Thomas J. Watson Research Center*

Abstract. Protocol Composition Logic (PCL) is a logic for proving authentication and secrecy properties of network protocols. This chapter presents the central concepts of PCL, including a protocol programming language, the semantics of protocol execution in the presence of a network attacker, the syntax and semantics of PCL assertions, and axioms and proof rules for proving authentication properties. The presentation draws on a logical framework enhanced with subtyping, setting the stage for mechanizing PCL proofs. and gives a new presentation of PCL semantics involving honest and unconstrained principals. Other papers on PCL provide additional axioms, proof rules, and case studies of standardized protocols in common use.

1. Introduction

Protocol Composition Logic (PCL) [5,6,7,8,9,10,11,12,17,18,23,38,39] is a formal logic for stating and proving security properties of network protocols. The logic was originally proposed in 2001 and has evolved over time through case studies of widely-used protocols such as SSL/TLS, IEEE 802.11i (WPA2), and variants of Kerberos. In typical case studies, model checking [31,30,36] is used to find bugs in a protocol design. After bugs found using model checking are eliminated, PCL can be used to find additional problems or prove that no further vulnerabilities exist. While model checking generally only analyzes protocol execution up to some finite bound on the number of protocol participants, PCL is designed to reason about the unbounded set of possible protocol executions. While the version of PCL considered in this chapter is based on a symbolic computation model of protocol execution and attack, variants of PCL have been developed and proved sound over conventional cryptographic computational semantics [11,12,37,38,40].

The central research question addressed by PCL is whether it is possible to prove properties of practical network security protocols compositionally, using direct reasoning that does not explicitly mention the actions of a network attacker. Intuitively, "direct reasoning" means that we draw conclusions about the effect of a protocol from the individual steps in it, without formulating the kind of reduction argument that is used to derive a contradiction from the assumption that the protocol is vulnerable to attack. For example, an axiom of PCL says, in effect, that if a principal creates a nonce (unguessable random number), and another principal receives a message containing it, then the first principal must have first sent a message containing the nonce.

The PCL assertions about protocols are similar to Hoare logic [24] and dynamic logic [21], stating before-after conditions about actions of one protocol participant. In conventional terminology adopted by PCL, a *thread* is an instance of a protocol role executed by a principal, such as Alice executing the client role of SSL. The PCL formula $\varphi\,[\mathtt{actseq}]_{\mathtt{T}}\,\psi$ states that if thread T starts in a state where precondition φ is true, then after actions actseq are executed by T, postcondition ψ must be true in any resulting state. While this formula only mentions the actions actseq of thread T, states which are reached after T executes actseq may arise as the result of these actions and any additional actions performed by other threads, including arbitrary actions by an attacker.

PCL preconditions and postconditions may include any of a number of action predicates, such as Send(T, msg), Receive(T, msg), NewNonce(T, msg), $\mathsf{Dec}(\mathtt{T}, \mathtt{msg}_{\mathrm{encr}}, \mathtt{K})$, $\mathsf{Verify}(\mathtt{T}, \mathtt{msg}_{\mathrm{signed}}, \mathtt{K})$, which assert that the named thread has performed the indicated action. For example, Send(T, msg) holds in a run if thread T sent the term msg as a message. One class of secrecy properties can be specified using the predicate Has(T, msg), which intuitively means that msg is built from constituents that T either generated or received in messages that do not hide parts of msg from T by encryption. One formula that is novel to PCL is Honest(Pname), which asserts that all actions of principal Pname are actions prescribed by the protocol. Honest is used primarily to assume that one party has followed the prescribed steps of the protocol. For example, if Alice initiates a transaction with Bob and wishes to conclude that only Bob knows the data she sends, she may be able to do so by explicitly assuming that Bob is honest. If Bob is not honest, Bob may make his private key known to the attacker, allowing the attacker to decrypt intercepted messages. Therefore, Alice may not have any guarantees if Bob is dishonest.

In comparison with a previous generation of protocol logics such as BAN logic [4], PCL was also initially designed as a logic of authentication. Furthermore, it involves annotating programs with assertions, does not require explicit reasoning about the actions of an attacker, and uses formulas for freshness, for sending and receiving messages, and for expressing that two agents have a shared secret. In contrast to BAN and related logics, PCL avoids the need for an "abstraction" phase because PCL formulas contain the protocol programs. PCL also addresses temporal concepts directly, both through modal formulas that refer specifically to particular points in the execution of a protocol, and through temporal operators in pre- and post-conditions. PCL is also formulated using standard logical concepts (predicate logic and modal operators), does not involve "jurisdiction" or "belief", and has a direct connection with the execution semantics of network protocols that is used in explicit reasoning about actions of a protocol and an attacker, such as with Paulson's inductive method [34] and Schneider's rank function method **??** and [42].

An advantage of PCL is that each proof component identifies not only the local reasoning that guarantees the security goal of that component, but also the environmental conditions that are needed to avoid destructive interference from other protocols that may use the same certificates or key materials. These environment assumptions are then proved for the steps that require them, giving us an invariant that is respected by the protocol. In formulating a PCL proof, we therefore identify the precise conditions that will allow other protocols to be executed in the same environment without interference.

In this book chapter, we explain the use of PCL using the simplified handshake protocol used in other chapters. We present PCL syntax and its symbolic-execution semantics in a semi-formal way, as if we were encoding PCL in a logical framework. The

$$\begin{aligned} A \rightarrow B &: \left\{\!\!\left| [\![A, B, k]\!]_{sk(A)} \right|\!\!\right\}^{\mathsf{a}}_{pk(B)} \\ B \rightarrow A &: \{\!|s|\!\}^{\mathsf{s}}_{k} \end{aligned}$$

Figure 1. Handshake protocol, in arrows-and-messages form

logical framework we use is a typed language of expressions which includes a form of subtyping that allows us to make implicit conversions between related types. For example, different types of keys are subtypes of the type *key* of all keys. While PCL includes formal axioms and proof rules for proving properties of protocols, there is a portion of the logic (the entailment relation between precondition/postcondition formulas) that is not formalized. However, when PCL is encoded in a logical framework, this unformalized portion of PCL is handled by the underlying logic of the logical framework. Finally, it should be noted explicitly that PCL is a logical approach that has been developed in different ways in different papers, as part of both a basic research project to find effective ways of proving properties of protocols, and as part of an ongoing effort to carry out case studies on practical protocols, each of which may require different cryptographic functions or make different infrastructure assumptions. Therefore, this chapter is intended to explain PCL, show how its syntax and semantics may be defined, describe representative axioms and proof rules, and show how they are used for sample protocols. This chapter is not a comprehensive presentation of all previous work on PCL or its applications.

2. Example

2.1. Sample protocol

The running example protocol used in this book is a simple handshake protocol between two parties. This is shown using a common "arrows-and-messages" notation in Figure 1. Note that we use $[\![\mathtt{msg}]\!]_{\mathtt{K}}$ for the digital signature of `msg` with the signing key `K`. For protocols that assume a public-key infrastructure, we write pk(Alice) for Alice's public encryption key, dk(Alice) for the corresponding private key known only to Alice, sk(Alice) for Alice's private signing key, and vk(Alice) for the corresponding publicly known signature verification key. When needed, $\langle m_1, \ldots, m_n \rangle$ is a tuple of length n.

As discussed in other chapters, this protocol is vulnerable to attack if the identity of the intended responder, B, is omitted from the first message. We will focus on the corrected protocol because the primary purpose of PCL is to prove security properties of correct protocols. However, we will also discuss how the proof fails if B is omitted from the first message. The way that the proof fails for an insecure protocol can often be used to understand how the protocol may be vulnerable to attack.

2.2. Expressing the protocol in PCL

Since PCL is a logic for proving the correctness of a protocol from its definition, we must express the protocol explicitly, with enough detail to support a rigorous proof. In PCL, we express the protocol as a set of programs, in a high-level language, one program for each participant in the protocol.

The PCL proof system allows us to reason about each program, called a protocol *role*, individually and to combine properties of different roles. Because each of the operations used to construct, send, receive, or process an incoming message may be important to the correctness of a protocol, we represent each of these steps as a separate action.

One way to understand why a more detailed representation of the protocol is needed is to look back at Figure 1, and think about what is clear from the figure itself and what needs to be clarified in English. For example, the English description says, "the initiator A creates a fresh session key *k*", but there is nothing in Figure 1 that indicates whether *k* is created by A, or known previously to both parties. The same figure would be used for a protocol in which A and B both know *k* in advance and B only accepts the first message if the shared value *k* is sent. Part of the rigor of PCL is that we use a precise protocol language that unambiguously determines the set of possible protocol runs.

The handshake protocol in Figure 1 has two roles, the **Init** role for the initiator and the **Resp** role for responder in the protocol. These roles are written out in detail in Figure 2. While we will later introduce syntactic sugar that simplifies some of the notation, Figure 2 shows every cryptographic action and every step in the construction and manipulation of each message explicitly. Instead of assuming that a received message has a specific format, for example, the responder role uses projection functions (π_1, π_2, ...) on tuples to separate a message into parts.

For clarity, we write many actions in the form $\ell oc := exp$, where the form on the right-hand side of the assignment symbol := determines the value stored in location ℓoc. The assignable locations used in a role are local to that role, but may be used to compute values sent in messages. We write $!\ell oc$ for the value stored in location ℓoc. Since all of our roles are straight-line programs, it is easy to choose locations so that each location is assigned a value only once in any thread; in fact we make this a requirement. This single-assignment requirement simplifies reasoning about a protocol role.

For each pair of principals A and B, the program executed by initiator A to communicate with intended responder B is the sequence of actions **Init**(A, B) shown in Figure 2. In words, the actions performed by the initiator are the following: choose a new nonce and store it in local storage location k, sign the message $\langle \mathrm{A}, \mathrm{B}, !\mathtt{k} \rangle$ containing the principal names A and B, and send the encryption of this message. Because digital signatures do not provide confidentiality, we assume that there is a function **unsign** that returns the plaintext of a signed message. In this handshake protocol, the receiver uses parts of the plaintext to identify the verification key of the sender and verify the signature. We call a principal executing a role a *thread.*

If principal B executes the responder role **Resp**(B), the actions of the responder thread are exactly the actions given in Figure 2. Specifically, the responder thread receives a message `enca` that may be an encrypted message created by an initiator thread. In order to determine whether the message has the correct form to be an initiator's message, the responder decrypts the message with its private decryption key $dk(\mathrm{B})$ and then extracts the plaintext from what it expects to be a signed message. If decryption fails, or if the result of decryption does not have the correct form to have the plaintext removed from a signed message, then the thread stops at the action that fails and does not proceed further. If possible, the responder thread then retrieves the first, second, and third components of the triple that is the message plaintext, and verifies the digital signature using the principal name A sent in the message. If the signature verifies, and the second component of the plaintext tuple matches the responder's principal name, then the responder generates a new nonce *s*, encrypts it under *k*, and sends the result. An assert

```
Init(A, B : principal_name) = {
    k := newnonce;
    siga := sign ⟨A, B, !k⟩, sk(A);
    enca := enc !siga, pk(B);
    send !enca;

    encb := receive;
    s := sd !encb, !k;
}
```

```
Resp(B : principal_name) = {
    enca := receive;
    siga := dec !enca, dk(B);
    texta := unsign !siga;
    idA := π1 !texta;
    idB := π2 !texta;
    k := π3 !texta;
    assert: !texta = ⟨!idA, !idB, !k⟩;
    verify !siga, vk(!idA);
    assert: !idB = B;
    s := newnonce;
    encb := se !s, !k;
    send !encb;
}
```

Figure 2. Explicit presentation of handshake protocol

action **assert:** msg = msg′, such as the one used to check idB, succeeds only if the two messages are equal.

Roles are typically parameterized by the names of principals relevant to execution of the role, such as parameters A and B in Figure 2. When a role is defined, the first argument is treated as the principal who executes the role. The difference between the first parameter and the other parameters is that private keys associated with the first parameter may be used in the actions of the role. Technically, the information available to each role is limited to its role parameters, values explicitly named in the role, and the result of applying allowed key functions such as $dk(\cdot)$, which names a private decryption key. Visibility to key functions is determined by a part of the protocol definition called its *setup assumptions*. Our formalization of setup assumptions is described in Section 3.1.

While Figure 2 is completely precise, it is sometimes convenient to adopt abbreviations and notational conventions that make the roles easier to read. For example, Figure 3 uses a few syntactical simplifications to define the same roles as Figure 2. Specifically, we use a pattern-matching form $\langle \mathtt{idA}, \mathtt{idB}, \mathtt{k}\rangle := \mathtt{texta}$ to set three locations to the three components of a tuple, as abbreviation for the three assignments using projection functions given in Figure 2 and the subsequent structural assertion that texta is a tuple of arity 3. We also omit ! and write ℓoc instead of $!\ell oc$ in expressions. Since all identifiers are typed, and the type of locations is different from other types, this is unambiguous; occurrences of ! can always be inserted mechanically based on the types of expressions. Some papers on PCL use a form "**match** texta **as** ⟨A, B, k⟩" that combines structural checks and assignment. While succinct, a potentially confusing aspect of **match** is that its effect depends on the context: because A and B are previously defined, this example **match** is an **assert** about them, while because k is not previously defined, the example **match** sets k to the third component of texta. Thus, writing **match** texta **as** ⟨A, B, k⟩ in the middle of **Resp**(B) would be equivalent to **assert:** $\pi_1$texta = A; **assert:** $\pi_2$texta = B; k := $\pi_3$texta, together with a structural assertion for checking the arity of texta (see above). For simplicity, we will avoid **match** in this presentation of PCL, although protocols written using **match** may be understood using this presentation of PCL as just explained.

```
Init(A, B : principal_name) = {
    k := newnonce;
    siga := sign ⟨A, B, k⟩, $sk$(A)
    enca := enc siga, $pk$(B)
    send enca;

    encb := receive;
    s := sd encb, k;
}

Resp(B : principal_name) = {
    enca := receive;
    siga := dec enca, $dk$(B);
    texta := unsign siga;
    ⟨idA, idB, k⟩ := texta;
    verify siga, $vk$(idA);
    assert: idB = B;
    s := newnonce;
    encb := se s, k
    send encb;
}
```

Figure 3. Explicit presentation of handshake protocol, with abbreviations

$$Auth_{Resp} : true[\mathbf{Resp}(B)]_{T} \left(\begin{array}{c} \mathsf{Honest}(\mathtt{idA}^{[T]}) \wedge \mathtt{idA}^{[T]} \neq B \\ \Rightarrow \left(\begin{array}{c} \exists T' : T'.pname = \mathtt{idA}^{[T]} \\ \wedge \mathsf{Send}(T', \mathsf{enca}^{[T]}) \lhd \mathsf{Receive}(T, \mathsf{enca}^{[T]}) \\ \wedge \mathsf{Receive}(T, \mathsf{enca}^{[T]}) \lhd \mathsf{Send}(T, \mathsf{encb}^{[T]}) \end{array} \right) \end{array} \right)$$

Figure 4. Property $Auth_{Resp}$

2.3. The protocol attacker

A protocol determines a set of runs, each with a sequence of actions by one or more principals. Since we are concerned with the behavior of protocols under attack, not all of the principals are assumed to follow the protocol. More specifically, we define a run of a protocol by choosing some number of principals, assigning one or more roles to each of them, choosing keys, and considering the effect of actions that may or may not follow the roles. If a principal follows the roles assigned to it, then we consider the principal *honest.* The attacker is represented by the set of dishonest principals, which may act arbitrarily using the cryptographic material assigned to them.

2.4. Authentication properties

Our formulation of authentication is based on the concept of *matching conversations* [3]. This requires that whenever Alice and Bob accept each other's identities at the end of a run, their records of the run *match*, i.e., each message that Alice sent was received by Bob and vice versa, each send event happened before the corresponding receive event, and the messages sent by each principal appear in the same order in both the records. An example of such an authentication property formalized as the PCL modal formula $Auth_{Resp}$ is shown in Figure 4. Note that the last line on the right of $Auth_{Resp}$ seems trivial; while not important for authentication, we are leaving it in to illustrate that PCL can prove ordering properties of individual roles (whose veracity really follows from honesty). (An analogous property $Auth_{Init}$ can be defined easily.)

Intuitively, $Auth_{Resp}$ says that starting from any state (since the precondition is *true*), if B executes the actions in the responder role purportedly with idA, then B is guaranteed that idA was involved in the protocol at some point (purportedly with B) and messages were sent and received in the expected order, provided that idA is honest (meaning that she always faithfully executes some role of the protocol and does not, for example, send out her private keys) and is a different principal from B.

3. Syntax

The syntax for expressing the roles of a protocol, PCL assertions for reasoning about them, and a symbolic framework for modeling protocol execution, may all be presented as a single typed language. We present this language as a set of type names, subtype relations, and typed constants (symbols that are used to write expressions) in a logical framework with explicit typing and subtyping. Through subtyping, we are able to consider different types of keys as subtypes of the type *key* of keys, for example, and similarly consider *key* a subtype of the type *message* of messages that may be sent on the network.

3.1. Protocols, roles, and setup assumptions

Protocols. A *protocol* is defined by a set of roles and setup assumptions. In Table 2, we define the type `protocol` of protocols as a set of parameterized roles (discussed below), using the logical framework described in section 3.2. The set of protocol executions are then defined using a set of instantiated roles and an initial state satisfying the setup assumptions.

Roles. A *role* is a sequence of actions, divided into subsequences that we refer to as *basic sequences.* The division of roles into basic sequences allows the protocol analyst to express when a thread may proceed internally, and when a thread may pause and possibly fail to continue. As defined in Table 2, a basic sequence is a sequence of actions. A role may have one or more parameters that are instantiated for each thread that executes the role. These parameters generally include the name of the principal executing the role and the names of other principals with whom execution of this role involves communicating.

Setup assumptions. The initial conditions that must be established before a protocol can be executed are called setup assumptions. In this chapter, we consider initialization conditions that determine the set of keys that may be used by each principal in the roles they execute. For example, a protocol may assume that each principal has a public key-pair, and that the public key of this pair is known to every other principal. Additional assumptions about protocol execution may also be stated in PCL preconditions.

The key assumptions we consider in this book chapter are expressed by giving a set of functions that are used to name keys, and specifying which principals have access to each key. The key assumptions may also restrict the ways that keys provided by setup assumptions can be sent by honest principals.

Every PCL key belongs to one of five disjoint types: *sym_key* for symmetric keys, *asym_enc_key* for asymmetric encryption keys, *asym_dec_key* for asymmetric decryption keys, *sgn_key* for signing keys, and *ver_key* for signature verification keys.

The technical machinery for restricting use of certain setup keys involves an additional type *conf_key* of confidential keys that cannot be released by honest principals, and confidential subtypes of each of the five key types. For example, the type *conf_asym_dec_key* of *confidential asymmetric decryption keys* is a subtype of both *asym_dec_key* and *conf_key*. Similarly, the type *conf_sgn_key* of *confidential signing keys* is a subtype of both *sgn_key* and *conf_key*. More generally, any confidential key type *conf_xyz_key* is a subtype of both the corresponding unrestricted key type *xyz_key* and *conf_key*.

The specific setup assumptions we use for the example protocols considered in this chapter provide public-key encryption and signature infrastructure. These assumptions are expressed using the following key functions (mapping principal names to various types of keys), key-pair relations, and key possession properties (stating which principals have access to which keys):

$$
\begin{aligned}
&pk : \mathtt{principal\_name} \rightarrow \mathtt{asym\_enc\_key}\\
&dk : \mathtt{principal\_name} \rightarrow \mathtt{conf\_asym\_dec\_key}\\
&sk : \mathtt{principal\_name} \rightarrow \mathtt{conf\_sgn\_key}\\
&vk : \mathtt{principal\_name} \rightarrow \mathtt{ver\_key}\\
&\forall \mathtt{Pname} : \mathit{AKeyPair}(pk(\mathtt{Pname}), dk(\mathtt{Pname})) \wedge \mathit{SKeyPair}(sk(\mathtt{Pname}), vk(\mathtt{Pname}))\\
&\forall (\mathtt{T}, \mathtt{Pname}) : \mathit{Setup}(\mathtt{T}, pk(\mathtt{Pname}))\\
&\forall (\mathtt{T}, \mathtt{Pname}) : \mathtt{T}.\mathit{pname} = \mathtt{Pname} \Rightarrow \mathit{Setup}(\mathtt{T}, dk(\mathtt{Pname}))\\
&\forall (\mathtt{T}, \mathtt{Pname}) : \mathtt{T}.\mathit{pname} = \mathtt{Pname} \Rightarrow \mathit{Setup}(\mathtt{T}, sk(\mathtt{Pname}))\\
&\forall (\mathtt{T}, \mathtt{Pname}) : \mathit{Setup}(\mathtt{T}, vk(\mathtt{Pname}))
\end{aligned}
$$

These specifications are used to choose and distribute keys in the setup phase of protocol execution. Note that for the form of public-key infrastructure expressed here, an honest principal $\mathtt{Princ}$ will be prevented from sending $dk(\mathtt{Princ}.\mathit{pname})$ or $sk(\mathtt{Princ}.\mathit{pname})$ in any message. (This restriction on behavior of honest principals will be enforced by a syntactic condition on roles.) It is possible to change the setup assumptions to allow an honest principal to send the decryption key of its asymmetric key-pair by changing the type of $dk(\cdot)$ above. However, since many protocols do not send messages that reveal these keys, we usually find it convenient to make this assumption from the outset and eliminate the need to prove separately that the protocol preserves confidentiality of selected keys. It is possible to carry such proofs out in PCL if desired.

Setup assumptions for symmetric keys can be expressed similarly, as needed for protocols that may require shared symmetric keys. For example, one form of key setup assumptions for Kerberos can be expressed by stating that there is a symmetric key $ckdc(X)$ for each client X, known to X and the Key-Distribution Center (KDC), a symmetric key $stgs(Y)$ for each server Y, known to Y and the Ticket-Granting Service (TGS), and a key shared between the KDC and the TGS. While other forms of setup assumptions are possible in principle, the present formulation using a set of key functions (and key constants) and a set of sharing assumptions is sufficient for many protocols.

3.2. Logical framework

A *logical framework* is a framework that provides a uniform way of encoding a logical language, its inference rules, and its proofs [22]. Without going into detail about logical frameworks and their use in formalizing specific logical systems, we discuss a specific framework briefly because it allows us to define the syntax, core aspects of the symbolic model of protocol execution, and the proof system of PCL succinctly and precisely.

The logical framework we use is a typed language of expressions that includes pairing, functions (to represent variable binding, as in quantified formulas), finite lists, and finite sets. We also assume a form of subtyping that is similar to order-sorted algebra [19]. This provides a convenient formalism for giving every expression a type, but allowing expressions of one type to be used as expressions of another without syntactic conversion functions. The logical framework we use, which might be called an *order-sorted logical framework,* has the following grammar:

$$
\begin{aligned}
\mathit{type} &::= \mathit{basic\_type} \,|\, \mathit{type} \times \ldots \times \mathit{type} \,|\, \mathit{type} \rightarrow \mathit{type} \,|\, \mathit{List}(\mathit{type}) \,|\, \mathit{Set}_{\mathit{fin}}(\mathit{type})\\
\mathit{term} &::= \mathit{fct\_symb} : \mathit{type} \,|\, \mathit{variable} : \mathit{type}\\
&\quad |\ \langle \mathit{term}_1 \ \ldots\ \mathit{term}_n \rangle \,|\, \pi_i\ \mathit{term}\\
&\quad |\ \lambda\, \mathit{variable} : \mathit{type}_v \,.\, \mathit{term} \,|\, \mathit{term}\ \mathit{term}_1\ \ldots\ \mathit{term}_n\ (n = 0, 1, 2, \ldots)\\
&\quad |\ \ldots
\end{aligned}
$$

In words, types are built from basic type names (chosen below for PCL) using product types (tuples), function types, and types of finite lists and finite sets. Terms, also called "expressions", are either basic constants (symbols chosen below for PCL), lambda expressions, function application, or additional forms for defining or using finite lists or finite sets.

The logical framework we use has standard typing rules. For example, writing $\mathit{term} : \mathit{type}$ to mean a term has a specific type, if $f : \mathit{type}_1 \rightarrow \mathit{type}_2$ and $x : \mathit{type}_1$, then $f\,x : \mathit{type}_2$. A lambda expression $\lambda \mathit{var}{:}\mathit{type}_v \,.\, \mathit{exp}$ defines the function that, given argument *var*, has value *exp*. If *var* has type type_v and *term* has type type_t, then $(\lambda \mathit{var} : \mathit{type}_v \,.\, \mathit{term})$ has type $\mathit{type}_v \rightarrow \mathit{type}_t$. We will not use lambda expressions in PCL, except that lambda is used as a binding operator to express parameterization (of protocol roles) and quantification (for assertions).

We use two operator families, *tupling* $(\langle \cdot, \ldots, \cdot \rangle)$ and *projection* (π_i), to construct and deconstruct tuple $(\times)$ types. For instance, $\langle$'message1', 'message2', 'message3'$\rangle$ is a triple of messages. To extract the third component from it, we write $\pi_3\,\langle$'message1', 'message2', 'message3'$\rangle$. We use the element symbol $\in$ to also denote membership in a list and the operator : for concatenation of elements of a list type.

We write $\mathit{type}_1 \sqsubseteq \mathit{type}_2$ to indicate that type_1 is a subtype of type_2. In this case, an expression of type type_1 can be used as an expression of type type_2.

A particular detail that we adopt from order-sorted algebra [19] for convenience is the use of "type retracts". If $\mathit{type}_1 \sqsubseteq \mathit{type}_2$, then an expression of the supertype type_2 can sometimes be used as an expression of the subtype type_1, which is the opposite of what is normally associated with subtyping. However, this is semantically reasonable if we regard the "retracted" expression of the smaller type as the result of applying a "cast" (as in common programming languages) mapping the supertype to the subtype. Intuitively, the meaning of this retract (or cast) is to check the value of the expression, and leave the value of the expression unchanged if it is semantically of the smaller type and otherwise disallow the operation containing the expression with the retract.

3.3. PCL types and subtyping

The PCL types are presented in Table 1. For clarity and flexibility, we use separate types for nonces (unguessable fresh values generated during protocol execution), and different types of keys. A `principal_name` is a name given to a principal who may execute one or more roles of a protocol, a `role_name` is a name of a role, used to distinguish between different roles of a protocol, `action`s are steps executed during a run of protocol. We use `thread_id`s to give each thread (principal executing a sequence of actions) a unique identifier. Each thread may assign values to local variables that we give type `location`, so that we may distinguish assignable variables from logical variables ("program constants") whose values are not changed by actions of principals. The type of `message`s includes nonces, keys, the result of cryptographic operations, and any other values that may be sent from one thread to another. Types `action_formula`, `nonmodal_formula`, `modal_formula`, `formula`, and `statement` are different types of assertions about pro-

Table 1. PCL Types

type name	meta-variables
nonce	$N_1, N_2, \ldots$
sym_key, *conf_sym_key*	$K_1, K_2, \ldots$
asym_enc_key, *conf_asym_enc_key*	$K_1, K_2, \ldots$
asym_dec_key, *conf_asym_dec_key*	$K_1, K_2, \ldots$
sgn_key, *conf_sgn_key*	$K_1, K_2, \ldots$
ver_key, *conf_ver_key*	$K_1, K_2, \ldots$
conf_key	$K_1, K_2, \ldots$
key	$K_1, K_2, \ldots$
principal_name	$\mathtt{Pname}_1, \mathtt{Pname}_2, \ldots; X_1, Y_2, \ldots; A_1, B_2, \ldots$
role_name	$\mathtt{rname}_1, \mathtt{rname}_2, \ldots$
action	$\mathtt{act}_1, \mathtt{act}_2, \ldots$
thread_id	$\mathtt{tid}_1, \mathtt{tid}_2, \ldots$
location	$\ell oc_1, \ell oc_2, \ldots$
message	$\mathtt{msg}_1, \mathtt{msg}_2, \ldots$
action_formula	$\mathtt{af}_1, \mathtt{af}_2, \ldots$
nonmodal_formula	$\varphi_1, \varphi_2, \ldots; \psi_1, \psi_2, \ldots$
modal_formula	$\Phi_1, \Phi_2, \ldots; \Psi_1, \Psi_2, \ldots$
formula	$\mathtt{fml}_1, \mathtt{fml}_2, \ldots$
statement	$\mathtt{stm}_1, \mathtt{stm}_2, \ldots$

Table 2. PCL Type abbreviations

type name : abbreviation	meta-variables
thread : *principal_name* $\times$ *role_name* $\times$ *List*(*principal_name*) $\times$ *thread_id*	$T_1, T_2, \ldots$
principal : *principal_name* $\times$ *List*(*key*)	$\mathtt{Princ}_1, \mathtt{Princ}_2, \ldots$
actionseq : *List*(*action*)	$\mathtt{actseq}_1, \mathtt{actseq}_2, \ldots$
basicseq : *List*(*action*)	$\mathtt{basicseq}_1, \mathtt{basicseq}_2, \ldots$
role : *role_name* $\times$ *List*(*basicseq*)	$\mathtt{role}_1, \mathtt{role}_2, \ldots$
protocol : Set_{fin}(*List*(*principal_name*) $\rightarrow$ *role*)	$\mathcal{P}_1, \mathcal{P}_2, \ldots$
event : *thread_id* $\times$ *action*	$\mathtt{ev}_1, \mathtt{ev}_2, \ldots$
store : *thread_id* $\times$ *location* $\rightarrow$ *message*	$\mathtt{st}_1, \mathtt{st}_2, \ldots$
run : Set_{fin}(*principal*) $\times$ Set_{fin}(*thread*) $\times$ *List*(*event*) $\times$ *store*	$\mathcal{R}_1, \mathcal{R}_2, \ldots$

tocols and their execution. The types used in PCL are either basic types given in the table, or composite types constructed from them using the four operators $\times$, $\rightarrow$, *List*$(\cdot)$, and $Set_{fin}(\cdot)$. The binary infix constructors $\times$ and $\rightarrow$ create product types and functional mappings, respectively; the unary constructors *List*$(\cdot)$ and $Set_{fin}(\cdot)$ create lists and finite sets, respectively.

PCL also uses *type abbreviations*, presented in Table 2. These are non-constructive definitions, also known as "syntactic sugar". An example of a composite type is *thread*, which is constructed from the basic types *principal_name*, *role_name*, and *thread_id* using the two type constructor operators $\times$ and *List*$(\cdot)$.

The subtype relation $\sqsubseteq$ is a partial order: it is reflexive, transitive, and antisymmetric. PCL uses the following subtype relationships:

$$\begin{aligned}
\mathtt{conf\_sym\_key} &\sqsubseteq \mathtt{sym\_key} \sqsubseteq \mathtt{key}\\
\mathtt{conf\_asym\_enc\_key} &\sqsubseteq \mathtt{asym\_enc\_key} \sqsubseteq \mathtt{key}\\
\mathtt{conf\_asym\_dec\_key} &\sqsubseteq \mathtt{asym\_dec\_key} \sqsubseteq \mathtt{key}\\
\mathtt{conf\_sgn\_key} &\sqsubseteq \mathtt{sgn\_key} \sqsubseteq \mathtt{key}\\
\mathtt{conf\_ver\_key} &\sqsubseteq \mathtt{ver\_key} \sqsubseteq \mathtt{key}\\
\mathtt{conf\_sym\_key}, \mathtt{conf\_asym\_enc\_key}, \ldots, \mathtt{conf\_ver\_key} &\sqsubseteq \mathtt{conf\_key}\\
\mathtt{conf\_key} &\sqsubseteq \mathtt{key}\\
\mathtt{principal\_name}, \mathtt{nonce}, \mathtt{key}, \times_{i=1}^{n} \mathtt{message} &\sqsubseteq \mathtt{message}\\
\mathtt{action\_formula} &\sqsubseteq \mathtt{nonmodal\_formula}\\
\mathtt{nonmodal\_formula}, \mathtt{modal\_formula} &\sqsubseteq \mathtt{formula}
\end{aligned}$$

Here, n may be any non-negative integer. Some clarification may be useful regarding product types: $\times_{i=1}^{0}$ `message` denotes the empty product type for `message`; its only member is $\langle\,\rangle$:`message`; thus the empty tuple $\langle\,\rangle$ is a `message`.

The types defined in this section include types used in writing PCL protocols, types used in writing assertions about them, and types used in defining the symbolic execution semantics of protocols. The type `protocol` itself is used to characterize the definition of a protocol. As it appears in Table 2, the type `protocol` consists of a set of parameterized roles. We call a function *List*(`principal_name`) $\rightarrow$ `role` from principal names to roles a *parameterized role*, and the result of applying a parameterized role to a list of principal names a *role instantiation*. The first argument is assumed to be the principal executing the role; this is relevant to determining which keys can be used in executing the role. Although a protocol may also specify *setup assumptions*, we consider them a separate set of typed constants and formulas that is not part of the type `protocol`.

Although `basicseq` and `actionseq` are both lists of actions, we use the two types differently. In the semantics section we define *honesty* of principals by requiring honest principals to execute zero or more basic sequences. In effect, PCL reasons about an abstraction of the protocol in which each basic sequence, once started by an honest principal, runs to completion. An implementation may respect this abstraction by testing whether the conditions exist to finish the basic sequence before performing any externally visible action (such as sending a message) of that sequence. We consider the division of a role into basic sequences part of the definition of the PCL model of the protocol, but impose a restriction on **receive** `action`s. Because we assume progress within a basic sequence, and a **receive** `action` may block waiting for a message from the network, a **receive** `action`s may only occur as the first action of the basic sequence. In figures, we use blank lines to indicate boundaries between basic sequences.

3.4. Protocol actions and roles

As indicated by the types in Table 2, protocols are defined using lists of actions. These actions, as illustrated by example in Section 2, create, send, and receive messages. PCL constants and functions for naming messages, and actions are listed in Tables 3 and 4, resp. User-friendly PCL notation for cryptographic functions is given in Table 5.

The terms used in PCL messages are taken modulo a set of equations axiomatized in Table 6. (There are also equivalences induced by *stores*, which are mappings from assignable variables of type location, as discussed in the section on semantics below.) Instead of the functional notation based on the presentation of PCL in a logical framework with subtyping, we use what we refer to as *PCL syntax*, presented in Table 7.

There are some details about the relationship between nonces and dynamically generated keys that merit discussion. In principle, there are several ways that cryptographic

Table 3. Constant symbols in PCL

symbol	type
$ky_1, ky_2, \ldots; k_1, k_2, \ldots$	$\mathtt{key}$
Alice, Bob, Charlie, ...; A, B, C, ...	$\mathtt{principal\_name}$
Init, **Resp**, ...	$\mathtt{role\_name}$
1, 2, ...	$\mathtt{thread\_id}$
x, y, siga, encb, texta, idB, k, s, ...	$\mathtt{location}$
‘’, ‘a’, ‘ca’, ‘hello’, ... (all strings)	$\mathtt{message}$
true, *false*	$\mathtt{nonmodal\_formula}$

Table 4. Function symbols for PCL actions

function symbol	type
!	$\mathtt{location} \to \mathtt{message}$
encmsg	$\mathtt{message} \times \mathtt{asym\_enc\_key} \to \mathtt{message}$
decmsg	$\mathtt{message} \times \mathtt{asym\_dec\_key} \to \mathtt{message}$
semsg	$\mathtt{message} \times \mathtt{nonce} \to \mathtt{message}$
semsg	$\mathtt{message} \times \mathtt{sym\_key} \to \mathtt{message}$
sdmsg	$\mathtt{message} \times \mathtt{nonce} \to \mathtt{message}$
sdmsg	$\mathtt{message} \times \mathtt{sym\_key} \to \mathtt{message}$
sgnmsg	$\mathtt{message} \times \mathtt{sgn\_key} \to \mathtt{message}$
unsgnmsg	$\mathtt{message} \to \mathtt{message}$
send	$\mathtt{message} \to \mathtt{action}$
receive	$\mathtt{location} \to \mathtt{action}$
newnonce	$\mathtt{location} \to \mathtt{action}$
enc	$\mathtt{location} \times \mathtt{message} \times \mathtt{asym\_enc\_key} \to \mathtt{action}$
dec	$\mathtt{location} \times \mathtt{message} \times \mathtt{asym\_dec\_key} \to \mathtt{action}$
se	$\mathtt{location} \times \mathtt{message} \times \mathtt{nonce} \to \mathtt{action}$
se	$\mathtt{location} \times \mathtt{message} \times \mathtt{sym\_key} \to \mathtt{action}$
sd	$\mathtt{location} \times \mathtt{message} \times \mathtt{nonce} \to \mathtt{action}$
sd	$\mathtt{location} \times \mathtt{message} \times \mathtt{sym\_key} \to \mathtt{action}$
sign	$\mathtt{location} \times \mathtt{message} \times \mathtt{sgn\_key} \to \mathtt{action}$
unsign	$\mathtt{location} \times \mathtt{message} \to \mathtt{action}$
verify	$\mathtt{message} \times \mathtt{ver\_key} \to \mathtt{action}$
assign	$\mathtt{location} \times \mathtt{message} \to \mathtt{action}$
assert	$\mathtt{message} \times \mathtt{message} \to \mathtt{action}$

operations could be modeled using symbolic computation. Cryptographic keys must be generated using some source of randomness. For dynamically generated keys, we separate randomness from key generation. We consider nonce generation $\mathtt{n} :=$ **newnonce** as a symbolic version of choosing a bitstring uniformly at random from bitstrings of appropriate length. A nonce $\mathtt{n}$ may then be used as an unguessable value or as a random seed to various deterministic key generation algorithms.

Encryption and signature algorithms typically apply a key generation algorithm (KeyGen) to deterministically transform a bitstring chosen from one probability distribution to a bitstring chosen from a distribution appropriate for the encryption/signature algorithm. For symmetric key operations the KeyGen algorithm generates just one key;

Table 5. Fully functional syntax vs. PCL syntax for cryptographic functions

Fully functional syntax	**PCL syntax**	**Description**
$\mathit{encmsg}(\mathtt{msg},\mathtt{K})$	$\{\!\vert\mathtt{msg}\vert\!\}^{\mathtt{a}}_{\mathtt{K}}$	Asymmetric encryption
$\mathit{decmsg}(\mathtt{msg}_{\mathrm{encr}},\mathtt{K})$	$\left\{\!\vert\mathtt{msg}_{\mathrm{encr}}\vert\!\right\}^{-\mathtt{a}}_{\mathtt{K}}$	Asymmetric decryption
$\mathit{semsg}(\mathtt{msg},\mathtt{N})$	$\{\!\vert\mathtt{msg}\vert\!\}^{\mathtt{s}}_{\mathtt{N}}$	Symmetric encryption by nonce
$\mathit{semsg}(\mathtt{msg},\mathtt{K})$	$\{\!\vert\mathtt{msg}\vert\!\}^{\mathtt{s}}_{\mathtt{K}}$	Symmetric encryption by pre-shared key
$\mathit{sdmsg}(\mathtt{msg}_{\mathrm{encr}},\mathtt{N})$	$\left\{\!\vert\mathtt{msg}_{\mathrm{encr}}\vert\!\right\}^{-\mathtt{s}}_{\mathtt{N}}$	Symmetric decryption by nonce
$\mathit{sdmsg}(\mathtt{msg}_{\mathrm{encr}},\mathtt{K})$	$\left\{\!\vert\mathtt{msg}_{\mathrm{encr}}\vert\!\right\}^{-\mathtt{s}}_{\mathtt{K}}$	Symmetric decryption by pre-shared key
$\mathit{sgnmsg}(\mathtt{msg},\mathtt{K})$	$[\![\mathtt{msg}]\!]_{\mathtt{K}}$	Signature
$\mathit{unsgnmsg}(\mathtt{msg}_{\mathrm{signed}})$	$[\![\mathtt{msg}_{\mathrm{signed}}]\!]^{-}$	Stripping off the signature

Table 6. Equations in PCL

Equation	**Types of variables used in equation**
$\pi_i\langle v_1, v_2, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, v_k\rangle = v_i$	$[k \in \mathbb{N}^+, i \in \{1, 2, \ldots, k\}; v_1 : \mathit{type}_1, v_2 : \mathit{type}_2, \ldots, v_k : \mathit{type}_k]$
$\left\{\!\vert\{\!\vert\mathtt{msg}\vert\!\}^{\mathtt{a}}_{\mathtt{K}_1}\vert\!\right\}^{-\mathtt{a}}_{\mathtt{K}_2} = \mathtt{msg}$	$[\mathrm{K}_1 : \mathit{asym\_enc\_key}, \mathrm{K}_2 : \mathit{asym\_dec\_key}, \mathtt{msg} : \mathit{message}$, where $(\mathrm{K}_1,\mathrm{K}_2)$ is an asymmetric encryption-decryption keypair]
$\left\{\!\vert\{\!\vert\mathtt{msg}\vert\!\}^{\mathtt{s}}_{\mathtt{N}}\vert\!\right\}^{-\mathtt{s}}_{\mathtt{N}} = \mathtt{msg}$	$[\mathtt{N} : \mathit{nonce}, \mathtt{msg} : \mathit{message}]$
$\left\{\!\vert\{\!\vert\mathtt{msg}\vert\!\}^{\mathtt{s}}_{\mathtt{K}}\vert\!\right\}^{-\mathtt{s}}_{\mathtt{K}} = \mathtt{msg}$	$[\mathrm{K} : \mathit{sym\_key}, \mathtt{msg} : \mathit{message}]$
$[\![[\![\mathtt{msg}]\!]_{\mathtt{K}}]\!]^{-} = \mathtt{msg}$	$[\mathrm{K} : \mathit{sgn\_key}, \mathtt{msg} : \mathit{message}]$

for public key algorithms the KeyGen algorithm generates two keys – one public, one private.

We will only consider dynamic generation of symmetric encryption keys in this chapter. For protocols with dynamically generated symmetric keys, the role that generates a key may send a nonce, relying on the receiving role to apply the same KeyGen algorithm to the nonce as the role that generated the nonce. Since the operations of key generation and encryption/decryption always go together, we can model this situation symbolically by having composite encryption/decryption operations, $[\cdot := \mathbf{se}\ \cdot, \mathtt{N}]/[\cdot := \mathbf{sd}\ \cdot, \mathtt{N}]$, that first apply KeyGen and then encrypt/decrypt. As a result, these encryption and decryption functions use a nonce instead of key to encrypt or decrypt a message.

Many protocols like Kerberos use pre-shared symmetric keys that are not sent around in the protocol. To model such protocols, in a way that is consistent with our treatment of dynamically-generated keys, we assume separate encryption/decryption operations that do not incorporate KeyGen. The threads themselves are configured with these keys according to protocol requirements. For example, in Kerberos, there are three types of symmetric keys: shared between two principals in Client and KAS roles, in KAS and TGS roles, and in TGS and Server roles. Since a principal can be operating in any one of these roles with another principal in any one of the peer roles, we have to explicitly specify the relation in the symbolic representation of the key. In one of our earlier papers [38,37] we denoted these keys as $k^{c\to k}_{X,Y}$, $k^{k\to t}_{X,Y}$ and $k^{t\to s}_{X,Y}$ respectively, between the two principals X and Y. Such pre-shared symmetric keys have a type that allows them to be used internally by threads but not sent on the network by honest principals.

Syntactic restrictions on roles. The type system places a number of restrictions on the syntax of roles. For example, events and stores are used in the semantics of protocol ex-

Table 7. Fully functional syntax vs. PCL syntax for actions

fully functional syntax	**PCL syntax**	**Description**
$!(\ell oc)$	$!\ell oc$	Contents of assignable location
send($\mathtt{msg}$)	**send** $\mathtt{msg}$	Send message
receive(ℓoc)	$\ell oc :=$ **receive**	Receive message and store in ℓoc
newnonce(ℓoc)	$\ell oc :=$ **newnonce**	Generate nonce and store in ℓoc
enc($\ell oc, \mathtt{msg}, \mathtt{K}$)	$\ell oc :=$ **enc** $\mathtt{msg}, \mathtt{K}$	Asymmetric encrypt and store in ℓoc
dec($\ell oc, \mathtt{msg}_{\mathrm{encr}}, \mathtt{K}$)	$\ell oc :=$ **dec** $\mathtt{msg}_{\mathrm{encr}}, \mathtt{K}$	Asymmetric decrypt and store in ℓoc
se($\ell oc, \mathtt{msg}, \mathtt{N}$)	$\ell oc :=$ **se** $\mathtt{msg}, \mathtt{N}$	Symmetric encrypt and store in ℓoc
se($\ell oc, \mathtt{msg}, \mathtt{K}$)	$\ell oc :=$ **se** $\mathtt{msg}, \mathtt{K}$	Symmetric encrypt and store in ℓoc
sd($\ell oc, \mathtt{msg}_{\mathrm{encr}}, \mathtt{N}$)	$\ell oc :=$ **sd** $\mathtt{msg}_{\mathrm{encr}}, \mathtt{N}$	Symmetric decrypt and store in ℓoc
sd($\ell oc, \mathtt{msg}_{\mathrm{encr}}, \mathtt{K}$)	$\ell oc :=$ **sd** $\mathtt{msg}_{\mathrm{encr}}, \mathtt{K}$	Symmetric decrypt and store in ℓoc
sign($\ell oc, \mathtt{msg}, \mathtt{K}$)	$\ell oc :=$ **sign** $\mathtt{msg}, \mathtt{K}$	Sign message and store in ℓoc
unsign($\ell oc, \mathtt{msg}_{\mathrm{signed}}$)	$\ell oc :=$ **unsign** $\mathtt{msg}_{\mathrm{signed}}$	Store plaintext of signed message in ℓoc
verify($\mathtt{msg}_{\mathrm{signed}}, \mathtt{K}$)	**verify** $\mathtt{msg}_{\mathrm{signed}}, \mathtt{K}$	Verify signed message
assign($\ell oc, \mathtt{msg}$)	$\ell oc := \mathtt{msg}$	Assign to storable ℓoc
assert($\mathtt{msg}_1, \mathtt{msg}_2$)	**assert:** $\mathtt{msg}_1 = \mathtt{msg}_2$	Equality check

ecution and cannot occur in roles because of the typing discipline. Due to typing constraints (specifically, $\mathtt{location} \not\sqsubseteq \mathtt{message}$), a location itself is not a message, but the contents of a location are a message. Therefore, if a protocol sends a message that depends on the contents of a location ℓoc, the role must use $!\ell oc$ to refer to the contents of the location. In addition, we impose the following restrictions on roles, beyond what follows from typing and subtyping:

- *No free variables.* A parameterized role of type $List(\mathtt{principal\_name}) \rightarrow \mathtt{role}$ appearing in a protocol must not have any free variables. In particular, all keys must be $\mathtt{key}$ constants, key expressions such as $pk(\mathtt{Pname})$ for principal names that are either constants or parameters of the role, or keys received in messages.
- *Single-assignment.* Each location must be a location constant, assigned to only once, and used only after it has been assigned. Since roles are given by loop-free programs, location names can easily be chosen to satisfy the single-assignment condition.
- *Key confidentiality.* If $\mathtt{K}$ has type $\mathtt{conf\_key}$, then $\mathtt{K}$ cannot occur in any assignment action $\ell oc := \mathtt{K}$ or in any expression (or argument to any action) except as an argument that is required to have type $\mathtt{key}$.
- *Local store operations.* The ! operator may occur in $\mathtt{role}$s, to obtain the value stored in a location. However, ! may not be used outside of $\mathtt{role}$s. The reason is that because locations are local to threads, $!\ell oc$ is ambiguous unless this expression occurs in a role assigned to a specific thread. The $\mathtt{Sto}(\cdot,\cdot)$ function (in Table 8) has a thread parameter and may occur in $\mathtt{formula}$s but not in $\mathtt{role}$s.

The key confidentiality condition given above allows a role to encrypt a message with a confidential key and send it, as in the sequence of actions $\ell oc :=$ **enc** $\mathtt{msg}, \mathtt{K}$; **send** $!\ell oc$. The reason this conforms to the syntactic *key confidentiality* condition is that **enc**'s argument signature is $\mathtt{location} \times \mathtt{message} \times \mathtt{asym\_enc\_key}$; i.e., the third argument is required to have a key type. Semantically, this preserves confidentiality of the key because encrypting a message with a key does not produce a message from which the key can be recovered. The key confidentiality condition does not allow $\ell oc := \textbf{enc}\ \mathtt{K}_1, \mathtt{K}_2$, in which a confidential key $\mathtt{K}_1$ is encrypted under another key, because the contents of ℓoc could then be used to construct a message that is sent. In other words, the key confidentiality condition is designed to be a conservative syntactic restriction on roles (to be followed by honest principals) that prevents export of confidential keys.

Further abbreviations. For most product types, we use record notation, such as `record.`*`recordelement`*, to access components of a pair. This aids readability because we can write `T.`*`rpars`*, using the potentially meaningful abbreviation *`.rpars`* (for "role parameters") instead of $\pi_3\mathtt{T}$. The abbreviations we use are:

- $\mathtt{T}.\mathit{pname} := \pi_1\mathtt{T}$, $\mathtt{T}.\mathit{rname} := \pi_2\mathtt{T}$, $\mathtt{T}.\mathit{rpars} := \pi_3\mathtt{T}$, $\mathtt{T}.\mathit{tid} := \pi_4\mathtt{T}$
- $\mathtt{Princ}.\mathit{pname} := \pi_1\mathtt{Princ}$, $\mathtt{Princ}.\mathit{klist} := \pi_2\mathtt{Princ}$
- $\mathtt{role}.\mathit{rname} := \pi_1\mathtt{role}$, $\mathtt{role}.\mathit{bseqs} := \pi_2\mathtt{role}$
- $\mathtt{ev}.\mathit{tid} := \pi_1\mathtt{ev}$, $\mathtt{ev}.\mathit{act} := \pi_2\mathtt{ev}$
- $\mathcal{R}.\mathit{pset} := \pi_1\mathcal{R}$, $\mathcal{R}.\mathit{tset} := \pi_2\mathcal{R}$, $\mathcal{R}.\mathit{elist} := \pi_3\mathcal{R}$, $\mathcal{R}.\mathit{st} := \pi_4\mathcal{R}$

The key functions are similarly projection functions that select keys out of the list of keys associated with the principal. Here, `Princ` refers to the contextually unique *`principal`* with *`principal_name`* `Pname` (that principals differ in their principal names is a condition on *feasibility of runs*, to be defined later), and we use (1-based) numbered indices in record notation for list members for legibility's sake:

- $\mathit{pk}(\mathtt{Pname}) = \mathtt{Princ}.\mathit{klist.1}$ $\mathit{dk}(\mathtt{Pname}) = \mathtt{Princ}.\mathit{klist.2}$
- $\mathit{sk}(\mathtt{Pname}) = \mathtt{Princ}.\mathit{klist.3}$ $\mathit{vk}(\mathtt{Pname}) = \mathtt{Princ}.\mathit{klist.4}$

Of course, this last set of definitions related to keys needs to be adapted to the respective *setup assumptions*. For role parameters we may also use indices:

- $\mathtt{T}.\mathit{rpars} = \langle \mathtt{T}.\mathit{rpars.1}, \mathtt{T}.\mathit{rpars.2}, \ldots \rangle$

3.5. PCL assertions

The syntax of PCL assertions is presented in Table 8. Quantifiers are regarded as function symbols of specific types, presented in Table 9. Relying on lambda abstraction from the underlying logical framework, we normally write, for example, $\forall_{key}\mathtt{K} : \mathtt{stm}$ instead of $\forall_{key}(\lambda\mathtt{K}.\mathtt{stm})$.

4. Semantics

This section presents the semantics of PCL. In outline, protocol execution determines a set of runs, each comprising a list of events that occur in the run. The semantics of formulas then determines which formulas are true for each run. Our semantics of protocols is based on the symbolic model [14,32], commonly called the Dolev-Yao model. In this model, an honest party or an adversary can only produce a symbolic expression representing a signature or decrypt an encrypted message if it possesses the appropriate key; the model does not involve bitstrings, probability distributions, or partial knowledge.

Recall that a protocol run $\mathcal{R} = \left\langle S_{princ}, S_{th}, eventlist, store \right\rangle$ consists of a finite set of principals, a finite set of threads, a finite list of events, and a store mapping assignable locations to the values they contain (see Table 2). Only certain combinations of principals, threads, events, and store form a run that could occur as an execution of a protocol.

Table 8. Function symbols for PCL assertions

function symbol	PCL notation	type
Sto		$\mathit{thread} \times \mathtt{location} \to \mathit{message}$
Send		$\mathit{thread} \times \mathit{message} \to \mathit{action\_formula}$
Receive		$\mathit{thread} \times \mathit{message} \to \mathit{action\_formula}$
NewNonce		$\mathit{thread} \times \mathit{message} \to \mathit{action\_formula}$
Enc		$\mathit{thread} \times \mathit{message} \times \mathit{key} \to \mathit{action\_formula}$
Dec		$\mathit{thread} \times \mathit{message} \times \mathit{key} \to \mathit{action\_formula}$
Se		$\mathit{thread} \times \mathit{message} \times \mathit{nonce} \to \mathit{action\_formula}$
Se		$\mathit{thread} \times \mathit{message} \times \mathit{key} \to \mathit{action\_formula}$
Sd		$\mathit{thread} \times \mathit{message} \times \mathit{nonce} \to \mathit{action\_formula}$
Sd		$\mathit{thread} \times \mathit{message} \times \mathit{key} \to \mathit{action\_formula}$
Sign		$\mathit{thread} \times \mathit{message} \times \mathit{key} \to \mathit{action\_formula}$
Unsign		$\mathit{thread} \times \mathit{message} \to \mathit{action\_formula}$
Verify		$\mathit{thread} \times \mathit{message} \times \mathit{key} \to \mathit{action\_formula}$
Assign		$\mathit{thread} \times \mathit{message} \to \mathit{action\_formula}$
Assert		$\mathit{thread} \times \mathit{message} \times \mathit{message} \to \mathit{action\_formula}$
$\lhd$	*infix*	$\mathit{action\_formula} \times \mathit{action\_formula} \to \mathit{nonmodal\_formula}$
Fresh		$\mathit{thread} \times \mathit{message} \to \mathit{nonmodal\_formula}$
Has		$\mathit{thread} \times \mathit{message} \to \mathit{nonmodal\_formula}$
FirstSend		$\mathit{thread} \times \mathit{message} \times \mathit{message} \to \mathit{nonmodal\_formula}$
Atomic		$\mathit{message} \to \mathit{nonmodal\_formula}$
$\Subset$	*infix*	$\mathit{message} \times \mathit{message} \to \mathit{nonmodal\_formula}$
$=$	*infix*	$\mathit{message} \times \mathit{message} \to \mathit{nonmodal\_formula}$
$=$	*infix*	$\mathit{thread} \times \mathit{thread} \to \mathit{nonmodal\_formula}$
Start		$\mathit{thread} \to \mathit{nonmodal\_formula}$
Honest		$\mathit{principal\_name} \to \mathit{nonmodal\_formula}$
$\neg$	$\neg\varphi$	$\mathit{nonmodal\_formula} \to \mathit{nonmodal\_formula}$
$\wedge$	*infix*	$\mathit{nonmodal\_formula} \times \mathit{nonmodal\_formula} \to \mathit{nonmodal\_formula}$
Modal	$\varphi\,[\mathtt{actseq}]_{\mathrm{T}}\,\psi$	$\mathit{nonmodal\_formula} \times \mathit{actionseq} \times \mathit{thread} \times \mathit{nonmodal\_formula} \to \mathit{modal\_formula}$
$\models$	$\mathcal{P} : \mathcal{R} \models \mathtt{fml}$	$\mathit{protocol} \times \mathit{run} \times \mathit{formula} \to \mathit{statement}$

Table 9. Quantifiers in PCL

quantifiers	type
$\forall_{\mathit{run}}$	$(\mathit{run} \to \mathit{statement}) \to \mathit{statement}$
$\forall_{\mathit{thread}}$	$(\mathit{thread} \to \mathit{statement}) \to \mathit{statement}$
$\forall_{\mathit{principal\_name}}$	$(\mathit{principal\_name} \to \mathit{statement}) \to \mathit{statement}$
$\forall_{\mathit{key}}$	$(\mathit{key} \to \mathit{statement}) \to \mathit{statement}$
$\forall_{\mathit{message}}$	$(\mathit{message} \to \mathit{statement}) \to \mathit{statement}$
$\forall_{\mathit{type}}$	$(\mathit{type} \to \mathit{formula}) \to \mathit{formula}$ (for all types *type*)

We therefore define the set of *feasible runs of a protocol* to characterize the well-formed runs in our model of protocol execution and attack. After preliminary definitions in Subsection 4.1, we define the semantics of protocols in Subsection 4.2. Subsection 4.3 gives an inductive definition of the semantic relation $\mathcal{P} : \mathcal{R} \models \varphi$, stating that the formula φ holds on a feasible run $\mathcal{R}$ of protocol $\mathcal{P}$.

4.1. Preliminary definitions

Properties of runs. Recall that each thread $\mathtt{T} \in S_{th}$ of a run $\mathcal{R} = \left\langle S_{princ}, S_{th}, eventlist, store\right\rangle$ has the form $\mathtt{T} = \langle \mathtt{Pname}, \mathtt{rname}, princ\_list, \mathtt{tid}\rangle$ consisting of a principal name, a role name, a list of `principal_name` parameters, and a thread id. The role name $\mathtt{T}.\mathit{rname}$ must name a role of `protocol` $\mathcal{P}$, where each role defines a sequence of actions. The events in the event list *eventlist* are pairs $\langle \mathtt{tid}, \mathtt{act}\rangle$ of actions by specific threads. We require that the actions in an eventlist do not contain locations (except in positions that require type `location`). We also require that the contents of assignable locations in the store do not contain locations either. We define runs in this way so that actions and stored values may be compared syntactically (as in the semantics of formulas) without relying on stores of the run.

For any run $\mathcal{R}$, we define the set $\mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid})$ of messages that `thread` tid can send out in `run` $\mathcal{R}$ without performing additional explicit actions. Our protocol programming language has operations such as encryption, decryption, and creating or verifying signatures as explicit actions in the run. This allows us to state formulas expressing which actions occurred and which did not; if no decryption occurred, for example, then no principal has the decrypted message. On the other hand, we treat pairing and projection as implicit operations – a thread can send a tuple or projection of values it has received or constructed, without an explicit tupling or projection action in the thread.

We let $\mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid})$ be the smallest set satisfying the following conditions, where T is the unique thread with `thread_id` `tid`:

1. Base cases:
   - *strings:*
     $\{\text{`'}, \text{`a'}, \text{`ca'}, \text{`hello'}, \ldots \text{ (all strings)}\} \subseteq \mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid})$
   - *principal names:*
     $\mathtt{Princ} \in \mathcal{R}.\mathit{pset} \Rightarrow \mathtt{Princ}.\mathit{pname} \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid})$
   - *own keys:*
     $\mathtt{T}.\mathit{pname} = \mathtt{Princ}.\mathit{pname} \wedge \mathtt{K} \in \mathtt{Princ}.\mathit{klist} \Rightarrow \mathtt{K} \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid})$
   - *other principals' keys:*
     For all $\mathtt{Princ} \in \mathcal{R}.\mathit{pset}$:
     $\mathit{Setup}(\mathtt{T}, \mathit{pk}(\mathtt{Princ}.\mathit{pname})) \Rightarrow \mathit{pk}(\mathtt{Princ}.\mathit{pname}) \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid})$
     $\mathit{Setup}(\mathtt{T}, \mathit{dk}(\mathtt{Princ}.\mathit{pname})) \Rightarrow \mathit{dk}(\mathtt{Princ}.\mathit{pname}) \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid})$
     $\mathit{Setup}(\mathtt{T}, \mathit{sk}(\mathtt{Princ}.\mathit{pname})) \Rightarrow \mathit{sk}(\mathtt{Princ}.\mathit{pname}) \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid})$
     $\mathit{Setup}(\mathtt{T}, \mathit{vk}(\mathtt{Princ}.\mathit{pname})) \Rightarrow \mathit{vk}(\mathtt{Princ}.\mathit{pname}) \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid})$
   - *stored messages:*
     $\mathcal{R}.\mathtt{st}(\mathtt{tid}, \ell oc) \text{ defined} \Rightarrow \mathcal{R}.\mathtt{st}(\mathtt{tid}, \ell oc) \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid})$

2. Inductive cases:
   - *tupling:*
     $\mathtt{msg}_1 \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid}) \wedge \mathtt{msg}_2 \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid}) \wedge \ldots \wedge \mathtt{msg}_k \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid})$
     $\Rightarrow \langle \mathtt{msg}_1, \mathtt{msg}_2, \ldots, \mathtt{msg}_k\rangle \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid})$
   - *projection:*
     $\langle \ldots, \mathtt{msg}, \ldots\rangle \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid}) \Rightarrow \mathtt{msg} \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid})$

We define the so-called Dolev-Yao closure of a set of `message`s. Although we do not use the Dolev-Yao closure to define the set of runs, we do use it to characterize the semantics of the Has predicate and to establish soundness of the proof system. The capability of the Dolev-Yao attacker are characterized by a deduction system. In the following rules, we use $\mathtt{msg} : DY$ to indicate that *message* $\mathtt{msg}$ is in the Dolev-Yao closure. The following ten inference rules characterize the Dolev-Yao attacker of protocols with asymmetric encryption, symmetric encryption, and signatures.

$$\frac{\mathtt{msg} : DY \quad \mathtt{K} : DY}{\{\!|\mathtt{msg}|\!\}^{\mathsf{a}}_{\mathtt{K}} : DY}\ \mathtt{K} : \mathit{asym\_enc\_key} \qquad \frac{\{\!|\mathtt{msg}|\!\}^{\mathsf{a}}_{\mathtt{K'}} : DY \quad \mathtt{K} : DY}{\mathtt{msg} : DY}\ \mathit{AKeyPair}(\mathtt{K'},\mathtt{K})$$

$$\frac{\mathtt{msg} : DY \quad \mathtt{N} : DY}{\{\!|\mathtt{msg}|\!\}^{\mathsf{s}}_{\mathtt{N}} : DY}\ \mathtt{N} : \mathit{nonce} \qquad \frac{\{\!|\mathtt{msg}|\!\}^{\mathsf{s}}_{\mathtt{N}} : DY \quad \mathtt{N} : DY}{\mathtt{msg} : DY}\ \mathtt{N} : \mathit{nonce}$$

$$\frac{\mathtt{msg} : DY \quad \mathtt{K} : DY}{\{\!|\mathtt{msg}|\!\}^{\mathsf{s}}_{\mathtt{K}} : DY}\ \mathtt{K} : \mathit{sym\_key} \qquad \frac{\{\!|\mathtt{msg}|\!\}^{\mathsf{s}}_{\mathtt{K}} : DY \quad \mathtt{K} : DY}{\mathtt{msg} : DY}\ \mathtt{K} : \mathit{sym\_key}$$

$$\frac{\mathtt{msg} : DY \quad \mathtt{K} : DY}{\llbracket \mathtt{msg} \rrbracket_{\mathtt{K}} : DY}\ \mathtt{K} : \mathit{sgn\_key} \qquad \frac{\llbracket \mathtt{msg} \rrbracket_{\mathtt{K}} : DY}{\mathtt{msg} : DY}\ \mathtt{K} : \mathit{sgn\_key}$$

$$\frac{\mathtt{msg}_1, \ldots, \mathtt{msg}_n \ : DY}{\langle \mathtt{msg}_1, \ldots, \mathtt{msg}_n \rangle : DY} \qquad \frac{\langle \mathtt{msg}_1, \ldots, \mathtt{msg}_n \rangle : DY}{\mathtt{msg}_i : DY}$$

The side condition $\mathit{AKeyPair}(\mathtt{K'}, \mathtt{K})$ above means that $(\mathtt{K'},\mathtt{K})$ is an asymmetric encryption-decryption keypair. Note that signing a messages does not hide the plaintext (that is, no *ver_key* is needed to see the unsigned message).

Given a set M of *message*s, we define the set *DolevYao*(M) of *message*s that a symbolic Dolev-Yao [14] attacker can generate from M as follows:

$\mathtt{msg}_0 \in$ *DolevYao*(M) *iff* $\mathtt{msg}_0 : DY$ can be derived using the inference rules above from the following hypotheses:

(1) $\mathtt{msg} : DY$ (for all $\mathtt{msg} \in M$)
(2) $\mathtt{N} : DY$ (for all $\mathtt{N} \in M$)
(3) $\mathtt{K} : DY$ (for all $\mathtt{K} \in M$)

Messages and protocol state. Our protocol programming language uses assignable locations to receive messages and perform internal actions such as encryption or decryption. If a protocol sends a message that depends on the contents of a location ℓoc, then the syntax of the role must use $!\ell oc$ to refer to the contents of the location, and the result is to send the contents of the location, written as $\mathcal{R}.\mathtt{st}(\mathtt{tid}, \ell oc)$ for *run* $\mathcal{R}$ and *thread_id* $\mathtt{tid}$.

For simplicity, we use the notation $\mathcal{R}[\mathtt{msg}]$ to refer to the *meaning of message msg in run* $\mathcal{R}$. Since we are using a symbolic semantics of protocols and their properties, the meaning of an expression that appears in a formula is a symbolic expression, but without locations or dependence on stores. The symbolic meaning $\mathcal{R}[\mathtt{msg}]$ has a straightforward inductive definition, with

$$\mathcal{R}[\mathtt{Sto}(\mathtt{T}, \ell oc)] = \mathcal{R}.\mathtt{st}(\mathtt{tid}, \ell oc)$$

as one base case. Since the other base cases and all inductive cases are trivial, the meaning of a message is the result of substituting values for locations. For actions, which may contain $!\ell oc$, we write $[\frac{\lambda \ell oc.\mathcal{R}.\mathtt{st}(\mathtt{tid},\ell oc)}{!}]\mathtt{act}$, where the substitution of a lambda expression for ! indicates the result of replacing ! by an explicit function and performing β-reduction to obtain an expression for the value stored in a given location.

Since threads generate new nonces and keys, we need a definition that allows us to see which symbols are not new relative to a run $\mathcal{R}_{prev} = \left\langle S_{princ}, S_{th}, eventlist, store\right\rangle$. Before giving this definition, we need some other definitions, concerning the $\cong$- (*congruence-*) and the $\Subset$- (*subterm-*) relations:

- Canonical form of *message*s: The equations in Table 6 form a confluent and terminating rewrite system, when oriented from left to right. The result of applying these rewrites to a message, until termination, is called the *canonical form* of the message.
- $\cong$: We use the notation $\mathtt{msg}_1 \cong \mathtt{msg}_2$ to express that the *message*s $\mathtt{msg}_1$ and $\mathtt{msg}_2$ are equal modulo the equational theory presented in Table 6, or equivalently, $\mathtt{msg}_1$ and $\mathtt{msg}_2$ have the same canonical form. (We use $\cong$ instead of $=$ to avoid confusion with *nonmodal_formula*s.)
- $\Subset$: We write $\mathtt{msg}_1 \Subset \mathtt{msg}_2$ to indicate that the *canonical form* of $\mathtt{msg}_1$ is a subterm of the *canonical form* of $\mathtt{msg}_2$.

Using the $\Subset$-relation, we can now say "*key* K occurs in $\mathcal{R}$" if:

- $\mathtt{K} \in \mathtt{Princ}.\mathit{klist}$ for some Princ occurring in S_{princ},
- K occurs in some role obtained by applying the parameterized role named $\mathtt{T}.\mathit{rname}$ to its role parameters $\mathtt{T}.\mathit{rpars}$ for thread $\mathtt{T} \in S_{th}$,
- $\mathtt{K} \Subset \mathtt{msg}$ for some msg in *eventlist*,
- $\mathtt{K} \Subset \mathtt{msg}$ for some $\langle\langle \mathtt{tid}, \ell oc\rangle, \mathtt{msg}\rangle \in \mathcal{R}.\mathit{st}$

The definition of "*nonce* N occurs in $\mathcal{R}$" is analogous.

Finally, we give a number of auxiliary definitions that we will need later to state the semantics of the Honest formula and of *modal_formula*s:

- $\mathcal{R}_1 : \mathcal{R}_2 : \ldots : \mathcal{R}_n$: We say that $\mathcal{R}_1 : \mathcal{R}_2 : \ldots : \mathcal{R}_n$ is a *division* of *run* $\mathcal{R}$ if:

  * $\mathcal{R}_i.\mathit{pset} = \mathcal{R}.\mathit{pset}$ for $1 \leq i \leq n$,
  * $\mathcal{R}_i.\mathit{tset} = \mathcal{R}.\mathit{tset}$ for $1 \leq i \leq n$,
  * $\mathcal{R}_i.\mathit{elist}$ is an initial segment of $\mathcal{R}_{i+1}.\mathit{elist}$ for $1 \leq i < n$,
  * $\mathcal{R}_i.\mathit{st}$ contains only the mappings defined by $\mathcal{R}_i.\mathit{elist}$, for $1 \leq i \leq n$.

  The run components are cumulative because this simplifies our formalism overall.
- $\mathtt{events}|_\mathtt{T}$: The *projection* $\mathtt{events}|_\mathtt{T}$ of eventlist events onto a thread T is the subsequence of *event*s ev from events with $\mathtt{ev}.\mathit{tid} = \mathtt{T}.\mathit{tid}$.
- *difference*: If $\mathtt{events}_1$ is an initial segment of $\mathtt{events}_2$, their *difference* ($\mathtt{events}_2 \setminus \mathtt{events}_1$) is the sequence of events such that $\mathtt{events}_1 : (\mathtt{events}_2 \setminus \mathtt{events}_1) = \mathtt{events}_2$.
- *matching*: We say that an *event* ev *matches* an *action* act in *run* $\mathcal{R}$ if $\mathtt{ev}.\mathit{act} \cong [\frac{\lambda \mathrm{loc}.\mathcal{R}.\mathit{st}(\mathtt{ev}.\mathit{tid}, \mathrm{loc})}{!}]\mathtt{act}$. A list of events matches a list of actions if both lists are of equal length and corresponding elements match.

4.2. *Feasibility of runs*

The *set of feasible runs* of a *protocol* $\mathcal{P}$ is defined inductively. Recall that a run $\mathcal{R}$ consists of a finite set of principals, a finite set of threads, a list of events and a store.

Intuitively, the base case of the inductive definition imposes conditions on the initial configuration before protocol execution begins. For example, the principal names and thread identifiers must be unique and each thread's role name must be one of the roles of protocol $\mathcal{P}$. The list of events and the store of the run are initially both empty.

The inductive case of the definition describes how one feasible run $\mathcal{R}_{prev}$ can be extended to form another feasible run $\mathcal{R}$, by an action performed by one of the threads. Generally, an action may occur only when certain *conditions* are satisfied. An action has the *effect* of appending the associated event to the list of events in $\mathcal{R}_{prev}$ and possibly extending the store to record values assigned to additional locations.

The condition for a thread `tid` to send a message in run $\mathcal{R}$ is that the message should be generable by the thread following the definition of $\mathit{GenMsg}_{\mathcal{R}}(\texttt{tid})$ presented earlier. When a message is sent, all occurrences of $!\ell oc$ indicating the contents of a local storage location in the message are replaced by the concrete message stored at that location, i.e., $\mathcal{R}.\mathtt{st}(\texttt{tid}, \ell oc)$. The conditions for the key generation actions ensure that the keys are fresh. An encryption action is possible only if the plaintext message and the encryption key are generable by the thread performing the action, whereas decryption succeeds only if the encrypted message and the decryption key are generable. Similarly, signature generation requires both the message and the signing key to be generable by the thread. Finally, an assignment action succeeds if the message that is being assigned to a location is generable by the thread, and an equality test on messages succeeds if the two messages are equal modulo the equational theory of the message algebra.

For convenience, we write $\mathcal{R}.\mathtt{pnames}$ for the set of principal names and $\mathcal{R}.\mathtt{tids}$ for the set of thread ids in $\mathcal{R} = \left\langle S_{princ}, S_{th}, \ldots \right\rangle$, i.e., $\mathcal{R}.\mathtt{pnames} = \{\texttt{pn} \mid \exists \texttt{Princ} \in S_{princ} : \texttt{Princ}.\mathtt{pname} = \texttt{pn}\}$ and $\mathcal{R}.\mathtt{tids} = \{\texttt{tid} \mid \exists \texttt{T} \in S_{th} : \texttt{T}.\mathtt{tid} = \texttt{tid}\}$.

Base case. An empty run $\mathcal{R} = \left\langle S_{princ}, S_{th}, \langle\,\rangle, \{\,\} \right\rangle$ is feasible for any protocol $\mathcal{P}$, where $\langle\,\rangle$ is the empty sequence of `event`s and and $\{\,\}$ is the empty `store`, provided all of the following conditions are met:

• All `principal`s in set S_{princ} differ in their `principal_name`:

$\forall \langle \texttt{Pname}_i, \mathit{keylist}_i \rangle, \left\langle \texttt{Pname}_j, \mathit{keylist}_j \right\rangle \in S_{princ}$:
$\texttt{Pname}_i = \texttt{Pname}_j \Rightarrow \mathit{keylist}_i = \mathit{keylist}_j$

• The keylist for each `principal` in S_{princ} has the keys named in the setup assumptions, each of the type specified in the setup assumptions. For the setup assumptions used for protocols in this book chapter, this means:

$pk(\texttt{Princ}.\mathtt{pname}) = \texttt{Princ}.\mathtt{klist.1} : \mathtt{asym\_enc\_key}$
$dk(\texttt{Princ}.\mathtt{pname}) = \texttt{Princ}.\mathtt{klist.2} : \mathtt{conf\_asym\_dec\_key}$
$sk(\texttt{Princ}.\mathtt{pname}) = \texttt{Princ}.\mathtt{klist.3} : \mathtt{conf\_sgn\_key}$
$vk(\texttt{Princ}.\mathtt{pname}) = \texttt{Princ}.\mathtt{klist.4} : \mathtt{ver\_key}$
$\mathtt{AKeyPair}(pk(\texttt{Princ}.\mathtt{pname}), dk(\texttt{Princ}.\mathtt{pname}))$
$\mathtt{SKeyPair}(sk(\texttt{Princ}.\mathtt{pname}), vk(\texttt{Princ}.\mathtt{pname}))$

• All `thread`s differ in their `thread_id`s:

$\forall \texttt{T}_i, \texttt{T}_j \in S_{th}$: $(\texttt{T}_i \neq \texttt{T}_j) \Rightarrow \texttt{T}_i.\mathtt{tid} \neq \texttt{T}_j.\mathtt{tid}$

• Every parameterized `role` of any `thread` (instantiated with the respective role parameters) is in `protocol` $\mathcal{P}$:

$\forall \texttt{T} \in S_{th} : \exists \texttt{prm\_role} \in \mathcal{P}$: $(\texttt{prm\_role}(\texttt{T}.\mathtt{rpars})).\mathtt{rname} = \texttt{T}.\mathtt{rname}$

• The set of `principal_name`s in S_{princ} is exactly the set of `principal`s of `thread`s in S_{th}:

$\mathcal{R}.\mathtt{pnames} = \{\texttt{Pname} \mid \exists \texttt{T} \in S_{th} : \texttt{T}.\mathtt{pname} = \texttt{Pname}\}$

Table 10. *Feasibility of runs* conditions (inductive case)

act	*newassgns*	conditions
$\ell oc := \mathbf{receive}$	$\{\langle\langle \mathtt{tid}, \ell oc\rangle, \mathtt{msg}\rangle\}$	msg was sent out previously: $\exists \mathtt{tid}' : \langle \mathtt{tid}', \mathbf{send}\ \mathtt{msg}\rangle \in \mathit{eventlist}$
$\mathbf{send}\ \mathtt{msg}$	$\{\}$	$\mathtt{msg} \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid})$
$\ell oc := \mathbf{newnonce}$	$\{\langle\langle \mathtt{tid}, \ell oc\rangle, \mathtt{N}\rangle\}$	$\mathtt{N}: \mathit{nonce}$ does not occur in $\mathcal{R}$.
$\ell oc := \mathbf{enc}\ \mathtt{msg}, \mathtt{K}$	$\{\left\langle\langle \mathtt{tid}, \ell oc\rangle, \{\!\|\mathtt{msg}\|\!\}^{\mathtt{a}}_{\mathtt{K}}\right\rangle\}$	$\mathtt{K} : \mathit{asym\_enc\_key} \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid})$ $\mathtt{msg} \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid})$
$\ell oc := \mathbf{dec}\ \{\!\|\mathtt{msg}\|\!\}^{\mathtt{a}}_{\mathtt{K}'}, \mathtt{K}$	$\{\langle\langle \mathtt{tid}, \ell oc\rangle, \mathtt{msg}\rangle\}$	$\mathtt{K} : \mathit{asym\_dec\_key} \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid})$ ((K′,K) is an asymmetric encr.-decr. keypair) $\{\!\|\mathtt{msg}\|\!\}^{\mathtt{a}}_{\mathtt{K}'} \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid})$
$\ell oc := \mathbf{se}\ \mathtt{msg}, \mathtt{N}$	$\{\left\langle\langle \mathtt{tid}, \ell oc\rangle, \{\!\|\mathtt{msg}\|\!\}^{\mathtt{s}}_{\mathtt{N}}\right\rangle\}$	$\mathtt{N} : \mathit{nonce} \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid})$ $\mathtt{msg} \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid})$
$\ell oc := \mathbf{se}\ \mathtt{msg}, \mathtt{K}$	$\{\left\langle\langle \mathtt{tid}, \ell oc\rangle, \{\!\|\mathtt{msg}\|\!\}^{\mathtt{s}}_{\mathtt{K}}\right\rangle\}$	$\mathtt{K} : \mathit{sym\_key} \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid})$ $\mathtt{msg} \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid})$
$\ell oc := \mathbf{sd}\ \{\!\|\mathtt{msg}\|\!\}^{\mathtt{s}}_{\mathtt{N}}, \mathtt{N}$	$\{\langle\langle \mathtt{tid}, \ell oc\rangle, \mathtt{msg}\rangle\}$	$\mathtt{N} : \mathit{nonce} \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid})$ $\{\!\|\mathtt{msg}\|\!\}^{\mathtt{s}}_{\mathtt{N}} \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid})$
$\ell oc := \mathbf{sd}\ \{\!\|\mathtt{msg}\|\!\}^{\mathtt{s}}_{\mathtt{K}}, \mathtt{K}$	$\{\langle\langle \mathtt{tid}, \ell oc\rangle, \mathtt{msg}\rangle\}$	$\mathtt{K} : \mathit{sym\_key} \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid})$ $\{\!\|\mathtt{msg}\|\!\}^{\mathtt{s}}_{\mathtt{K}} \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid})$
$\ell oc := \mathbf{sign}\ \mathtt{msg}, \mathtt{K}$	$\{\langle\langle \mathtt{tid}, \ell oc\rangle, [\![\mathtt{msg}]\!]_{\mathtt{K}}\rangle\}$	$\mathtt{K} : \mathit{sgn\_key} \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid})$ $\mathtt{msg} \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid})$
$\ell oc := \mathbf{unsign}\ [\![\mathtt{msg}]\!]_{\mathtt{K}}$	$\{\langle\langle \mathtt{tid}, \ell oc\rangle, \mathtt{msg}\rangle\}$	$[\![\mathtt{msg}]\!]_{\mathtt{K}} \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid})$
$\mathbf{verify}\ [\![\mathtt{msg}]\!]_{\mathtt{K}'}, \mathtt{K}$	$\{\}$	$\mathtt{K} : \mathit{ver\_key} \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid})$ ((K′,K) is a sig.-verif. keypair) $[\![\mathtt{msg}]\!]_{\mathtt{K}'} \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid})$
$\ell oc := \mathtt{msg}$	$\{\langle\langle \mathtt{tid}, \ell oc\rangle, \mathtt{msg}\rangle\}$	$\mathtt{msg} \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid})$
$\mathbf{assert:}\ \mathtt{msg}_1 = \mathtt{msg}_2$	$\{\}$	$\mathtt{msg}_1 \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid}) \wedge \mathtt{msg}_2 \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{tid})$ $\mathtt{msg}_1 \cong \mathtt{msg}_2$.

Inductive case. If $\mathcal{R}_{prev} = \left\langle S_{princ}, S_{th}, \mathit{eventlist}, \mathit{store}\right\rangle$ is a feasible *run* for protocol $\mathcal{P}$, then $\mathcal{R} = \left\langle S_{princ}, S_{th}, \mathit{eventlist}{:}\langle \mathtt{tid}, \mathtt{act}\rangle, \mathit{store} \cup \mathit{newassgns}\right\rangle$ is a feasible *run* for $\mathcal{P}$, where the symbol : indicates list concatenation, $\mathtt{tid} \in \mathcal{R}.\mathtt{tids}$, and tid, act, and *newassgns* satisfy the conditions shown in Table 10.

Finally, a quick note on how the *attacker* is modeled: the attacker is a principal who executes a thread but does not necessarily follow the action sequence required by the notion of honesty (to be defined below in Section 4.3).

4.3. *Truth of statements*

In this section, we present an inductive definition of the semantic relation $\mathcal{P} : \mathcal{R} \models \varphi$ stating that the formula φ holds on a feasible run $\mathcal{R}$ of protocol $\mathcal{P}$.

Truth of *statement*s involving nonmodal formulas is defined as follows:

- $\underline{\mathcal{P}: \mathcal{R} \models \mathsf{Send}(\mathtt{T}, \mathtt{msg})}$: (1) T is a *thread* of $\mathcal{R}$. (2) T executed a **send** msg′ *action* such that $\mathcal{R}[\mathtt{msg}] \cong \mathtt{msg}'$.
- $\underline{\mathcal{P}: \mathcal{R} \models \mathsf{Receive}(\mathtt{T}, \mathtt{msg})}$: (1) T is a *thread* of $\mathcal{R}$. (2) T executed a $\ell oc := \mathbf{receive}$ *action* such that $\mathcal{R}[\mathtt{msg}] \cong \mathcal{R}.\mathtt{st}(\mathtt{T}.\mathtt{tid}, \ell oc)$.

- $\mathcal{P}\colon \mathcal{R} \models \mathsf{NewNonce}(\mathtt{T},\mathtt{msg})$: (1) $\mathtt{T}$ is a *thread* of $\mathcal{R}$. (2) $\mathtt{T}$ executed a $\ell oc := \textbf{newnonce}$ *action* such that $\mathcal{R}[\mathtt{msg}] \cong \mathcal{R}.\mathit{st}(\mathtt{T}.\mathit{tid}, \ell oc)$.
- $\mathcal{P}\colon \mathcal{R} \models \mathsf{Enc}(\mathtt{T},\mathtt{msg},\mathtt{K})$: (1) $\mathtt{T}$ is a *thread* of $\mathcal{R}$. (2) $\mathtt{T}$ executed a $\ell oc := \textbf{enc}\ \mathtt{msg}', \mathtt{K}'$ *action* such that $\mathcal{R}[\mathtt{msg}] \cong \mathtt{msg}'$ and $\mathcal{R}[\mathtt{K}] \cong \mathtt{K}'$.
- $\mathcal{P}\colon \mathcal{R} \models \mathsf{Dec}(\mathtt{T},\mathtt{msg}_{\text{encr}},\mathtt{K})$: (1) $\mathtt{T}$ is a *thread* of $\mathcal{R}$. (2) $\mathtt{T}$ executed a $\ell oc := \textbf{dec}\ \mathtt{msg}'_{\text{encr}}, \mathtt{K}'$ *action* such that $\mathcal{R}[\mathtt{msg}_{\text{encr}}] \cong \mathtt{msg}'_{\text{encr}}$ and $\mathcal{R}[\mathtt{K}] \cong \mathtt{K}'$.
- $\mathcal{P}\colon \mathcal{R} \models \mathsf{Se}(\mathtt{T},\mathtt{msg},\mathtt{N})$: (1) $\mathtt{T}$ is a *thread* of $\mathcal{R}$. (2) $\mathtt{T}$ executed a $\ell oc := \textbf{se}\ \mathtt{msg}', \mathtt{N}'$ *action* such that $\mathcal{R}[\mathtt{msg}] \cong \mathtt{msg}'$ and $\mathcal{R}[\mathtt{N}] \cong \mathtt{N}'$.
- $\mathcal{P}\colon \mathcal{R} \models \mathsf{Se}(\mathtt{T},\mathtt{msg},\mathtt{K})$: (1) $\mathtt{T}$ is a *thread* of $\mathcal{R}$. (2) $\mathtt{T}$ executed a $\ell oc := \textbf{se}\ \mathtt{msg}', \mathtt{K}'$ *action* such that $\mathcal{R}[\mathtt{msg}] \cong \mathtt{msg}'$ and $\mathcal{R}[\mathtt{K}] \cong \mathtt{K}'$.
- $\mathcal{P}\colon \mathcal{R} \models \mathsf{Sd}(\mathtt{T},\mathtt{msg}_{\text{encr}},\mathtt{N})$: (1) $\mathtt{T}$ is a *thread* of $\mathcal{R}$. (2) $\mathtt{T}$ executed a $\ell oc := \textbf{sd}\ \mathtt{msg}'_{\text{encr}}, \mathtt{N}'$ *action* such that $\mathcal{R}[\mathtt{msg}_{\text{encr}}] \cong \mathtt{msg}'_{\text{encr}}$ and $\mathcal{R}[\mathtt{N}] \cong \mathtt{N}'$.
- $\mathcal{P}\colon \mathcal{R} \models \mathsf{Sd}(\mathtt{T},\mathtt{msg}_{\text{encr}},\mathtt{K})$: (1) $\mathtt{T}$ is a *thread* of $\mathcal{R}$. (2) $\mathtt{T}$ executed a $\ell oc := \textbf{sd}\ \mathtt{msg}'_{\text{encr}}, \mathtt{K}'$ *action* such that $\mathcal{R}[\mathtt{msg}_{\text{encr}}] \cong \mathtt{msg}'_{\text{encr}}$ and $\mathcal{R}[\mathtt{K}] \cong \mathtt{K}'$ and $\mathcal{R}[\mathtt{K}] \cong \mathtt{K}'$.
- $\mathcal{P}\colon \mathcal{R} \models \mathsf{Sign}(\mathtt{T},\mathtt{msg},\mathtt{K})$: (1) $\mathtt{T}$ is a *thread* of $\mathcal{R}$. (2) $\mathtt{T}$ executed a $\ell oc := \textbf{sign}\ \mathtt{msg}', \mathtt{K}'$ *action* such that $\mathcal{R}[\mathtt{msg}] \cong \mathtt{msg}'$ and $\mathcal{R}[\mathtt{K}] \cong \mathtt{K}'$.
- $\mathcal{P}\colon \mathcal{R} \models \mathsf{Unsign}(\mathtt{T},\mathtt{msg}_{\text{signed}})$: (1) $\mathtt{T}$ is a *thread* of $\mathcal{R}$. (2) $\mathtt{T}$ executed a $\ell oc := \textbf{unsign}\ \mathtt{msg}'_{\text{signed}}$ *action* such that $\mathcal{R}[\mathtt{msg}_{\text{signed}}] \cong \mathtt{msg}'_{\text{signed}}$.
- $\mathcal{P}\colon \mathcal{R} \models \mathsf{Verify}(\mathtt{T},\mathtt{msg}_{\text{signed}},\mathtt{K})$: (1) $\mathtt{T}$ is a *thread* of $\mathcal{R}$. (2) $\mathtt{T}$ executed a $\textbf{verify}\ \mathtt{msg}'_{\text{signed}}, \mathtt{K}'$ *action* such that $\mathcal{R}[\mathtt{msg}_{\text{signed}}] \cong \mathtt{msg}'_{\text{signed}}$ and $\mathcal{R}[\mathtt{K}] \cong \mathtt{K}'$.
- $\mathcal{P}\colon \mathcal{R} \models \mathsf{Assign}(\mathtt{T},\mathtt{msg})$: (1) $\mathtt{T}$ is a *thread* of $\mathcal{R}$. (2) $\mathtt{T}$ executed a $\ell oc{:=}\mathtt{msg}'$ *action* such that $\mathcal{R}[\mathtt{msg}] \cong \mathtt{msg}'$.
- $\mathcal{P}\colon \mathcal{R} \models \mathsf{Assert}(\mathtt{T},\mathtt{msg}_1,\mathtt{msg}_2)$: (1) $\mathtt{T}$ is a *thread* of $\mathcal{R}$. (2) $\mathtt{T}$ executed an $\textbf{assert:}\ \mathtt{msg}'_1 = \mathtt{msg}'_2$ *action* such that $\mathcal{R}[\mathtt{msg}_1] \cong \mathtt{msg}'_1$ as well as $\mathcal{R}[\mathtt{msg}_2] \cong \mathtt{msg}'_2$.
- $\mathcal{P}\colon \mathcal{R} \models \mathtt{af}_1 \lhd \mathtt{af}_2$: (1) $\mathcal{P}\colon \mathcal{R} \models \mathtt{af}_1$ and $\mathcal{P}\colon \mathcal{R} \models \mathtt{af}_2$ are both true. (2) Both $\mathtt{af}_1$ and $\mathtt{af}_2$ are *action_formula*s (Send, Receive, NewNonce, ...), not necessarily of the same type. (3) There exist (**send**, **receive**, **newnonce**, ...) *action*s, labeled $\mathtt{act}_1$ and $\mathtt{act}_2$, in (possibly distinct) *thread*s $\mathtt{tid}_1$ and $\mathtt{tid}_2$ corresponding to $\mathtt{af}_1$ and $\mathtt{af}_2$, respectively (with "correspondence" as naturally defined in accordance with the truth conditions of the respective (Send, Receive, NewNonce, ...) *action_formula*s above), such that $\langle \mathtt{tid}_1, \mathtt{act}_1 \rangle$ occurs before $\langle \mathtt{tid}_2, \mathtt{act}_2 \rangle$ within $\mathcal{R}.\mathit{elist}$.
- $\mathcal{P}\colon \mathcal{R} \models \mathsf{Fresh}(\mathtt{T},\mathtt{msg})$: (1) $\mathtt{T}$ is a *thread* of $\mathcal{R}$. (2) $\mathcal{P}\colon \mathcal{R} \models \mathsf{NewNonce}(\mathtt{T},\mathtt{msg})$ is true. (3) $\forall \mathtt{msg}'\colon (\mathcal{P}\colon \mathcal{R} \models \mathsf{Send}(\mathtt{T},\mathtt{msg}')) \Rightarrow \mathcal{R}[\mathtt{msg}] \not\Subset \mathcal{R}[\mathtt{msg}']$.
- $\mathcal{P}\colon \mathcal{R} \models \mathsf{Has}(\mathtt{T},\mathtt{msg})$: (1) $\mathtt{T}$ is a *thread* of $\mathcal{R}$. (2) $\mathcal{R}[\mathtt{msg}] \in \mathit{DolevYao}(\mathit{GenMsg}_{\mathcal{R}}(\mathtt{T}.\mathit{tid}))$, where $\mathtt{Princ}$ is the *principal* executing *thread* $\mathtt{tid}$.
- $\mathcal{P}\colon \mathcal{R} \models \mathsf{FirstSend}(\mathtt{T},\mathtt{msg}_1,\mathtt{msg}_2)$: (1) $\mathtt{T}$ is a *thread* of $\mathcal{R}$. (2) $\mathcal{R}[\mathtt{msg}_1] \Subset \mathcal{R}[\mathtt{msg}_2] \wedge (\mathcal{P} : \mathcal{R} \models \mathsf{Send}(\mathtt{T},\mathtt{msg}_2))$. (3) For all messages $\mathtt{msg}_3$ such that $\mathcal{R}[\mathtt{msg}_1] \Subset \mathtt{msg}_3$ and $\langle \mathtt{T}.\mathit{tid}, \textbf{send}\ \mathtt{msg}_3 \rangle \in \mathcal{R}.\mathit{elist}$, there must be an earlier event $\left\langle \mathtt{T}.\mathit{tid}, \textbf{send}\ \mathtt{msg}'_2 \right\rangle$ with $\mathcal{R}[\mathtt{msg}_2] \cong \mathtt{msg}'_2$. In words, while $\mathtt{T}$ need *not* be the first *thread* to send out a message containing $\mathtt{msg}_1$, the first send by $\mathtt{T}$ containing $\mathtt{msg}_1$ must be a send of $\mathtt{msg}_2$.
- $\mathcal{P}\colon \mathcal{R} \models \mathsf{Atomic}(\mathtt{msg})$: $\mathcal{R}[\mathtt{msg}]$ is $\cong$-congruent to an atomic *message*. An atomic message is one in which neither of tupling, projection, encryption, decryption, signing, or unsigning occurs; i.e., the canonical form of $\mathcal{R}[\mathtt{msg}]$ is atomic.
- $\mathcal{P}\colon \mathcal{R} \models \mathtt{msg}_1 \Subset \mathtt{msg}_2$: $\mathcal{R}[\mathtt{msg}_1] \Subset \mathcal{R}[\mathtt{msg}_2]$.
- $\mathcal{P}\colon \mathcal{R} \models \mathtt{msg}_1 = \mathtt{msg}_2$: $\mathcal{R}[\mathtt{msg}_1] \cong \mathcal{R}[\mathtt{msg}_2]$.
- $\mathcal{P}\colon \mathcal{R} \models \mathtt{T}_1 = \mathtt{T}_2$: $\mathtt{T}_1$ and $\mathtt{T}_2$ are identical.
- $\mathcal{P}\colon \mathcal{R} \models \mathsf{Start}(\mathtt{T})$: (1) $\mathtt{T}$ is a *thread* of $\mathcal{R}$. (2) *thread* $\mathtt{T}$ did not execute any *action*s: $\nexists \mathtt{act}\colon \langle \mathtt{T}.\mathit{tid}, \mathtt{act} \rangle \in \mathcal{R}.\mathit{elist}$.
- $\mathcal{P}\colon \mathcal{R} \models \mathsf{Honest}(\mathtt{Pname})$: (1) Pname is a *principal_name* of $\mathcal{R}$: Pname $\in \mathcal{R}.\mathit{pnames}$. (2) For any *thread* $\mathtt{T}$ of Pname, let $\mathtt{role}$ be the unique *role* with $\mathtt{role}.\mathit{rname} = \mathtt{T}.\mathit{rname}$. There must be some $k \geq 0$ such that $[\mathtt{basicseq}_1; \mathtt{basicseq}_2; \ldots; \mathtt{basicseq}_k]$ is an ini-

tial sequence of `role.`*`bseqs`* and $\mathcal{R}$`.`*`elist`*$|_T$ matches (`basicseq`$_1$: `basicseq`$_2$: ... : `basicseq`$_k$), the concatenation of these basic sequences of the role. In other words, each thread of *`principal`* has executed precisely some number of basic sequences of its designated role. In particular, no thread can deviate from or stop in the middle of a basic sequence. (3) The canonical forms of all messages that `Pname` sent out may only have keys of type *`conf_key`* in positions that require type *`key`*. In other words, *`conf_key`*s are only used for encryption or signing messages, not as the "payload". (This is guaranteed by the syntactic condition on *`conf_key`* in roles.)

- $\mathcal{P}: \mathcal{R} \models \neg\varphi$: $\mathcal{P}: \mathcal{R} \models \varphi$ is not true.
- $\mathcal{P}: \mathcal{R} \models \varphi \wedge \psi$: $\mathcal{P}: \mathcal{R} \models \varphi$ and $\mathcal{P}: \mathcal{R} \models \psi$ are both true.
- $\mathcal{P}: \mathcal{R} \models \forall_{type} v:$ `fml`: For all ground instances **a** of v : *type* we have $\mathcal{P}: \mathcal{R} \models [\frac{\mathbf{a}}{v}]$`fml`, where $[\frac{\mathbf{a}}{v}]$`fml` is the formula obtained by substituting all occurrences of v in `fml` by **a**.

Truth of *`statement`*s involving modal formulas is defined as follows:

- $\mathcal{P}: \mathcal{R} \models \varphi\,[$`actseq`$]_T\,\psi$: For all divisions ($\mathcal{R}_1$: $\mathcal{R}_2$: $\mathcal{R}_3$) of *run* $\mathcal{R}$, the following holds: If $\mathcal{P}: \mathcal{R}_1 \models \varphi$ is true, and (($\mathcal{R}_2$`.`*`elist`*)$|_T$ \ ($\mathcal{R}_1$`.`*`elist`*)$|_T$) matches `actseq`, then $\mathcal{P}: \mathcal{R}_2 \models \psi$ is true.[1]

As usual, we employ "$\ldots \vee \ldots$", "$\ldots \Rightarrow \ldots$", and "$\exists \ldots$" as syntactic shortcuts for "$\neg(\neg \ldots \wedge \neg \ldots)$", "$\neg \ldots \vee \ldots$", and "$\neg\forall\neg \ldots$", respectively.

5. Proof system

This section describes some of the axioms and inference rules of PCL. The portion of the PCL proof system that is presented here is sufficient to prove authentication properties of the sample protocols considered in Sections 2 and 6. While we do not present secrecy axioms and proofs in this chapter, sample secrecy axioms and proofs are developed in other papers on PCL (e.g., [38,39]).

The PCL axioms and proof roles formalize reasoning about protocols, using PCL assertions. Like Hoare logic, the axioms are generally formulas about a given action, and many inference rules let us prove something about a sequence of actions from assertions about a shorter sequence of actions. If we want to use a formula $\varphi\,[$`actseq`$]_T\,\psi$ with postcondition ψ to prove a similar formula $\varphi\,[$`actseq`$]_T\,\psi'$ with postcondition ψ', it suffices to prove that $\psi \Rightarrow \psi'$. We do not give formal proof rules for deducing implications between preconditions or postconditions. In this sense, PCL could be considered a *semi-formalized* logic for proving properties of protocols. However, two points should be considered. First of all, the formulas used in preconditions and postconditions are logically simpler than modal formulas. As a result, PCL reduces formal reasoning about protocols to reasoning about the precondition-postcondition logic. Second, the presentation of PCL given in this chapter shows how PCL may be presented in a logical framework (or meta-logic). When PCL is embedded in a meta-logic, the axioms and proof rules given here essentially reduce proofs about protocols to proofs about preconditions and postconditions in the meta-logic. Therefore, PCL provides an approach (not fully developed in this chapter) for fully formal proofs of security properties of protocols, using a metalogic supporting the formal syntax and symbolic semantics development presented in this chapter.

[1]It is intentional that $\mathcal{R}_3$ does not appear in this definition.

For brevity in some of the proof rules, we will write $\mathtt{Sto}(\mathtt{T},\ell oc)$ as $\ell oc^{[\mathtt{T}]}$. In case there is some possible confusion, here is a summary of different forms associated with locations and their values:

- In protocol actions, $!\ell oc$ is the content of location ℓoc in the current thread.
- In formulas, $\mathtt{Sto}(\mathtt{T},\ell oc)$ is the content of location ℓoc in the local store of thread $\mathtt{T}$, with $\ell oc^{[\mathtt{T}]}$ being an abbreviation for $\mathtt{Sto}(\mathtt{T},\ell oc)$.
- In the semantics of formulas, the symbolic meaning $\mathcal{R}[\mathtt{msg}]$ of a message depends on the store of the run, with $\mathcal{R}[\mathtt{Sto}(\mathtt{T},\ell oc)] = \mathcal{R}.\mathtt{st}(\mathtt{tid},\ell oc)$ as one base case.

For convenience, we then extend $\mathtt{Sto}(\mathtt{T},\cdot)$ to messages, with $\mathtt{Sto}(\mathtt{T},\mathtt{msg})$ being the message obtained by replacing any subexpression $!\ell oc$ depending on the store with $\mathtt{Sto}(\mathtt{T},\ell oc)$.

The *soundness theorem* for the proof system is proved by induction on the length of proof: all instances of the axioms are valid formulas and all proof rules preserve validity. We state the soundness theorem upfront and give proof sketches for a few representative axiom and rule schemas while presenting them.

We write $\Gamma \vdash \gamma$ if γ is provable from the formulas in Γ and any axiom or rule schema of the proof system except the honesty rule **HON**. We write $\Gamma \vdash_{\mathcal{P}} \gamma$ if γ is provable from the formulas in Γ, the basic axioms and inference rules of the proof system and the honesty rule just for protocol $\mathcal{P}$. We write $\Gamma \models_{\mathcal{P}} \gamma$ if for any feasible run $\mathcal{R}$ of protocol $\mathcal{P}$ the following holds: If $\mathcal{P}\colon \mathcal{R} \models \Gamma$, then $\mathcal{P}\colon \mathcal{R} \models \gamma$. We write $\Gamma \models \gamma$ if for any protocol $\mathcal{P}$, we have $\Gamma \models_{\mathcal{P}} \gamma$. Here, Γ is a non-modal formula (typically, a formula proved using the honesty rule) and γ is either a modal formula or a non-modal formula.

Theorem 5.1 *If* $\Gamma \vdash_{\mathcal{P}} \gamma$*, then* $\Gamma \models_{\mathcal{P}} \gamma$*. Furthermore, if* $\Gamma \vdash \gamma$*, then* $\Gamma \models \gamma$*.*

Because of the undecidability of protocol properties expressible in PCL (see [15,16]) and the fact that invalid formulas can be enumerated by enumerating finite runs, it is not possible for the PCL proof system to be semantically complete. Therefore, we have developed PCL by adding new PCL predicates, axioms, and rules as needed to handle different classes of protocols and properties.

5.1. Axiom Schemas

The axiom schemas presented below can be divided up into the following categories: **AA0**–**AA4** state properties that hold in a state as a result of executing (or not executing) certain actions; **AN1**–**AN4** capture properties of nonces; **KEY** states which thread possesses what key, in effect, axiomatizing protocol setup assumptions, while **HAS** axiomatizes the Dolev-Yao deduction rules; **SEC** captures the hardness of breaking public key encryption while **VER** states that signatures are unforgeable; **IN** (and **NIN**) provide reasoning principles for when one message is contained in (and not contained in) another; **P1** and **P2** state that the truth of certain predicates is preserved under specific further actions; and **FS1** and **FS2** support reasoning about the temporal ordering of actions performed by different threads based on the fresh nonces they use.

Axiom schemas AA0. These are axiom schemas which infer term structure from term construction actions ($\bar{k}$ refers to the unique encr./sig. key for decr./verif. key k):

- $\mathit{true}\,[\mathtt{actseq};\ell oc := \mathbf{enc}\ \mathtt{msg},\mathtt{K};\mathtt{actseq}']_{\mathtt{T}}\ \mathtt{Sto}(\mathtt{T},\ell oc) = \{\!|\mathtt{Sto}(\mathtt{T},\mathtt{msg})|\!\}^{\mathtt{a}}_{\mathtt{Sto(T,K)}}$

- $\mathit{true}\,[\mathtt{actseq};\,\ell oc := \mathbf{dec}\ \mathtt{msg},\mathtt{K};\,\mathtt{actseq}']_{\mathtt{T}}\ \mathtt{Sto}(\mathtt{T},\mathtt{msg}) = \{\!|\mathtt{Sto}(\mathtt{T},\ell oc)|\!\}^{\mathsf{a}}_{\overline{\mathtt{Sto(T,K)}}}$
- $\mathit{true}\,[\mathtt{actseq};\,\ell oc := \mathbf{se}\ \mathtt{msg},\mathtt{N};\,\mathtt{actseq}']_{\mathtt{T}}\ \mathtt{Sto}(\mathtt{T},\ell oc) = \{\!|\mathtt{Sto}(\mathtt{T},\mathtt{msg})|\!\}^{\mathsf{s}}_{\mathtt{Sto(T,N)}}$
- $\mathit{true}\,[\mathtt{actseq};\,\ell oc := \mathbf{se}\ \mathtt{msg},\mathtt{K};\,\mathtt{actseq}']_{\mathtt{T}}\ \mathtt{Sto}(\mathtt{T},\ell oc) = \{\!|\mathtt{Sto}(\mathtt{T},\mathtt{msg})|\!\}^{\mathsf{s}}_{\mathtt{Sto(T,K)}}$
- $\mathit{true}\,[\mathtt{actseq};\,\ell oc := \mathbf{sd}\ \mathtt{msg},\mathtt{N};\,\mathtt{actseq}']_{\mathtt{T}}\ \mathtt{Sto}(\mathtt{T},\mathtt{msg}) = \{\!|\mathtt{Sto}(\mathtt{T},\ell oc)|\!\}^{\mathsf{s}}_{\mathtt{Sto(T,N)}}$
- $\mathit{true}\,[\mathtt{actseq};\,\ell oc := \mathbf{sd}\ \mathtt{msg},\mathtt{K};\,\mathtt{actseq}']_{\mathtt{T}}\ \mathtt{Sto}(\mathtt{T},\mathtt{msg}) = \{\!|\mathtt{Sto}(\mathtt{T},\ell oc)|\!\}^{\mathsf{s}}_{\mathtt{Sto(T,K)}}$
- $\mathit{true}\,[\mathtt{actseq};\,\ell oc := \mathbf{sign}\ \mathtt{msg},\mathtt{K};\,\mathtt{actseq}']_{\mathtt{T}}\ \mathtt{Sto}(\mathtt{T},\ell oc) = [\![\mathtt{Sto}(\mathtt{T},\mathtt{msg})]\!]_{\mathtt{Sto(T,K)}}$
- $\mathit{true}\,[\mathtt{actseq};\,\ell oc := \mathbf{unsign}\ \mathtt{msg};\,\mathtt{actseq}']_{\mathtt{T}}\ \mathtt{Sto}(\mathtt{T},\ell oc) = [\![\mathtt{Sto}(\mathtt{T},\mathtt{msg})]\!]^{-}$
- $\mathit{true}\,[\mathtt{actseq};\,\mathbf{verify}\ \mathtt{msg}_{\mathrm{signed}},\mathtt{K};\,\mathtt{actseq}']_{\mathtt{T}}\ \mathtt{Sto}(\mathtt{T},\mathtt{msg}_{\mathrm{signed}}) = [\![[\![\mathtt{Sto}(\mathtt{T},\mathtt{msg}_{\mathrm{signed}})]\!]^{-}]\!]_{\overline{\mathtt{Sto(T,K)}}}$
- $\mathit{true}\,[\mathtt{actseq};\,\ell oc := \mathtt{msg};\,\mathtt{actseq}']_{\mathtt{T}}\ \mathtt{Sto}(\mathtt{T},\ell oc) = \mathtt{Sto}(\mathtt{T},\mathtt{msg})$
- $\mathit{true}\,[\mathtt{actseq};\,\mathbf{assert:}\ \mathtt{msg}_1 = \mathtt{msg}_2;\,\mathtt{actseq}']_{\mathtt{T}}\ \mathtt{Sto}(\mathtt{T},\mathtt{msg}_1) = \mathtt{Sto}(\mathtt{T},\mathtt{msg}_2)$

Axiom schemas AA1. These are axiom schemas which state that after an action has occurred, the predicate asserting that the action has taken place is true:

- $\mathit{true}\,[\mathtt{actseq};\,\mathbf{send}\ \mathtt{msg};\,\mathtt{actseq}']_{\mathtt{T}}\ \mathsf{Send}(\mathtt{T},\mathtt{Sto}(\mathtt{T},\mathtt{msg}))$
- $\mathit{true}\,[\mathtt{actseq};\,\ell oc := \mathbf{receive};\,\mathtt{actseq}']_{\mathtt{T}}\ \mathsf{Receive}(\mathtt{T},\mathtt{Sto}(\mathtt{T},\ell oc))$
- $\mathit{true}\,[\mathtt{actseq};\,\ell oc := \mathbf{newnonce};\,\mathtt{actseq}']_{\mathtt{T}}\ \mathsf{NewNonce}(\mathtt{T},\mathtt{Sto}(\mathtt{T},\ell oc))$
- $\mathit{true}\,[\mathtt{actseq};\,\ell oc := \mathbf{enc}\ \mathtt{msg},\mathtt{K};\,\mathtt{actseq}']_{\mathtt{T}}\ \mathsf{Enc}(\mathtt{T},\mathtt{Sto}(\mathtt{T},\mathtt{msg}),\mathtt{Sto}(\mathtt{T},\mathtt{K}))$
- $\mathit{true}\,[\mathtt{actseq};\,\ell oc := \mathbf{dec}\ \mathtt{msg},\mathtt{K};\,\mathtt{actseq}']_{\mathtt{T}}\ \mathsf{Dec}(\mathtt{T},\mathtt{Sto}(\mathtt{T},\mathtt{msg}),\mathtt{Sto}(\mathtt{T},\mathtt{K}))$
- $\mathit{true}\,[\mathtt{actseq};\,\ell oc := \mathbf{se}\ \mathtt{msg},\mathtt{N};\,\mathtt{actseq}']_{\mathtt{T}}\ \mathsf{Se}(\mathtt{T},\mathtt{Sto}(\mathtt{T},\mathtt{msg}),\mathtt{Sto}(\mathtt{T},\mathtt{N}))$
- $\mathit{true}\,[\mathtt{actseq};\,\ell oc := \mathbf{se}\ \mathtt{msg},\mathtt{K};\,\mathtt{actseq}']_{\mathtt{T}}\ \mathsf{Se}(\mathtt{T},\mathtt{Sto}(\mathtt{T},\mathtt{msg}),\mathtt{Sto}(\mathtt{T},\mathtt{K}))$
- $\mathit{true}\,[\mathtt{actseq};\,\ell oc := \mathbf{sd}\ \mathtt{msg},\mathtt{N};\,\mathtt{actseq}']_{\mathtt{T}}\ \mathsf{Sd}(\mathtt{T},\mathtt{Sto}(\mathtt{T},\mathtt{msg}),\mathtt{Sto}(\mathtt{T},\mathtt{N}))$
- $\mathit{true}\,[\mathtt{actseq};\,\ell oc := \mathbf{sd}\ \mathtt{msg},\mathtt{K};\,\mathtt{actseq}']_{\mathtt{T}}\ \mathsf{Sd}(\mathtt{T},\mathtt{Sto}(\mathtt{T},\mathtt{msg}),\mathtt{Sto}(\mathtt{T},\mathtt{K}))$
- $\mathit{true}\,[\mathtt{actseq};\,\ell oc := \mathbf{sign}\ \mathtt{msg},\mathtt{K};\,\mathtt{actseq}']_{\mathtt{T}}\ \mathsf{Sign}(\mathtt{T},\mathtt{Sto}(\mathtt{T},\mathtt{msg}),\mathtt{Sto}(\mathtt{T},\mathtt{K}))$
- $\mathit{true}\,[\mathtt{actseq};\,\ell oc := \mathbf{unsign}\ \mathtt{msg};\,\mathtt{actseq}']_{\mathtt{T}}\ \mathsf{Unsign}(\mathtt{T},\mathtt{Sto}(\mathtt{T},\mathtt{msg}))$
- $\mathit{true}\,[\mathtt{actseq};\,\mathbf{verify}\ \mathtt{msg}_{\mathrm{signed}},\mathtt{K};\,\mathtt{actseq}']_{\mathtt{T}}\ \mathsf{Verify}(\mathtt{T},\mathtt{Sto}(\mathtt{T},\mathtt{msg}_{\mathrm{signed}}),\mathtt{Sto}(\mathtt{T},\mathtt{K}))$
- $\mathit{true}\,[\mathtt{actseq};\,\ell oc := \mathtt{msg};\,\mathtt{actseq}']_{\mathtt{T}}\ \mathsf{Assign}(\mathtt{T},\mathtt{Sto}(\mathtt{T},\mathtt{msg}))$
- $\mathit{true}\,[\mathtt{actseq};\,\mathbf{assert:}\ \mathtt{msg}_1 = \mathtt{msg}_2;\,\mathtt{actseq}']_{\mathtt{T}}\ \mathsf{Assert}(\mathtt{T},\mathtt{Sto}(\mathtt{T},\mathtt{msg}_1),\mathtt{Sto}(\mathtt{T},\mathtt{msg}_2))$

Proof: The soundness of axiom schemas **AA1** follows from the semantics of the relevant *`action`*s and *`action_formula`*s. For example, the semantics of the *`action_formula`* Receive specifies that the relevant thread must have executed a corresponding **receive** *`action`*, which is given by the way the axiom schema is stated. Furthermore, there is a single-assignment condition on *`location`*s which ensures that the value of $\mathtt{Sto}(\mathtt{T},\ell oc)$ (namely the contents of the store for location ℓoc in thread $\mathtt{T}$) is really what was received; that is, the contents of ℓoc could not have been overwritten by another *`message`*.

Axiom schemas AA2. These are axiom schemas which state that if a thread has not performed any action so far then a predicate asserting any action has taken place is false:

- $\mathsf{Start}(\mathtt{T})\,[\,]_{\mathtt{T}}\ \neg\mathsf{Send}(\mathtt{T},\mathtt{msg})$
- $\mathsf{Start}(\mathtt{T})\,[\,]_{\mathtt{T}}\ \neg\mathsf{Receive}(\mathtt{T},\mathtt{msg})$
- $\mathsf{Start}(\mathtt{T})\,[\,]_{\mathtt{T}}\ \neg\mathsf{NewNonce}(\mathtt{T},\mathtt{msg})$
- $\mathsf{Start}(\mathtt{T})\,[\,]_{\mathtt{T}}\ \neg\mathsf{Enc}(\mathtt{T},\mathtt{msg},\mathtt{K})$
- $\mathsf{Start}(\mathtt{T})\,[\,]_{\mathtt{T}}\ \neg\mathsf{Dec}(\mathtt{T},\mathtt{msg},\mathtt{K})$
- $\mathsf{Start}(\mathtt{T})\,[\,]_{\mathtt{T}}\ \neg\mathsf{Se}(\mathtt{T},\mathtt{msg},\mathtt{N})$
- $\mathsf{Start}(\mathtt{T})\,[\,]_{\mathtt{T}}\ \neg\mathsf{Se}(\mathtt{T},\mathtt{msg},\mathtt{K})$
- $\mathsf{Start}(\mathtt{T})\,[\,]_{\mathtt{T}}\ \neg\mathsf{Sd}(\mathtt{T},\mathtt{msg},\mathtt{N})$
- $\mathsf{Start}(\mathtt{T})\,[\,]_{\mathtt{T}}\ \neg\mathsf{Sd}(\mathtt{T},\mathtt{msg},\mathtt{K})$
- $\mathsf{Start}(\mathtt{T})\,[\,]_{\mathtt{T}}\ \neg\mathsf{Sign}(\mathtt{T},\mathtt{msg},\mathtt{K})$
- $\mathsf{Start}(\mathtt{T})\,[\,]_{\mathtt{T}}\ \neg\mathsf{Unsign}(\mathtt{T},\mathtt{msg})$
- $\mathsf{Start}(\mathtt{T})\,[\,]_{\mathtt{T}}\ \neg\mathsf{Verify}(\mathtt{T},\mathtt{msg},\mathtt{K})$
- $\mathsf{Start}(\mathtt{T})\,[\,]_{\mathtt{T}}\ \neg\mathsf{Assign}(\mathtt{T},\mathtt{msg})$
- $\mathsf{Start}(\mathtt{T})\,[\,]_{\mathtt{T}}\ \neg\mathsf{Assert}(\mathtt{T},\mathtt{msg}_1,\mathtt{msg}_2)$

Axiom schemas AA3. These axiom schemas allow us to prove that certain messages are not sent. Note that in this axiom schema and in some other cases below, the symbol ">" separates the "schema" part from the "axiom" (or "rule") part: For example, "$\forall \mathtt{actseq} \ldots :> \varphi\,[\mathtt{actseq}]_T\,\psi$" means that every instance of $\varphi\,[\mathtt{actseq}]_T\,\psi$ in which `actseq` satisfies the given condition is an axiom instance of the axiom schema.

- $\forall \mathtt{actseq}$ that do not contain a **send** *action* $:> \neg\mathsf{Send}(\mathtt{T},\mathtt{msg})\,[\mathtt{actseq}]_{\mathtt{T}}\,\neg\mathsf{Send}(\mathtt{T},\mathtt{msg})$
- $\neg\mathsf{Send}(\mathtt{T},\mathtt{msg})\;\;[\textbf{send}\;\mathtt{msg}']_{\mathtt{T}}\;\;\mathtt{Sto}(\mathtt{T},\mathtt{msg}') \neq \mathtt{msg} \Rightarrow \neg\mathsf{Send}(\mathtt{T},\mathtt{msg})$
- $\forall \mathtt{actseq}$ that do not contain a **sign** *action* $:> \neg\mathsf{Sign}(\mathtt{T},\mathtt{msg},\mathtt{K})\,[\mathtt{actseq}]_{\mathtt{T}}\,\neg\mathsf{Sign}(\mathtt{T},\mathtt{msg},\mathtt{K})$
- $\neg\mathsf{Sign}(\mathtt{T},\mathtt{msg},\mathtt{K})\;\;[\mathtt{loc} := \textbf{sign}\;\mathtt{msg}',\mathtt{K}']_{\mathtt{T}}\,(\mathtt{Sto}(\mathtt{T},\mathtt{msg}') \neq \mathtt{msg} \vee \mathtt{Sto}(\mathtt{T},\mathtt{K}') \neq \mathtt{K}$
  $\Rightarrow \neg\mathsf{Sign}(\mathtt{T},\mathtt{msg},\mathtt{K}))$

Axiom schemas AA4 These axiom schemas state that if one action occurs after another, they are related in the same temporal order:

- $\mathit{true}\,[\textbf{send}\;\mathtt{msg_a};\mathtt{actseq};\textbf{send}\;\mathtt{msg_b}]_{\mathtt{T}}\;\mathsf{Send}(\mathtt{T},\mathtt{Sto}(\mathtt{T},\mathtt{msg_a})) \lhd \mathsf{Send}(\mathtt{T},\mathtt{Sto}(\mathtt{T},\mathtt{msg_b}))$
- $\mathit{true}\,[\textbf{send}\;\mathtt{msg_a};\mathtt{actseq};\ell oc_{\mathrm{b}} := \textbf{receive}]_{\mathtt{T}}\;\mathsf{Send}(\mathtt{T},\mathtt{Sto}(\mathtt{T},\mathtt{msg_a}))\lhd\mathsf{Receive}(\mathtt{T},\mathtt{Sto}(\mathtt{T},\ell oc_{\mathrm{b}}))$
- $\mathit{true}\,[\textbf{send}\;\mathtt{msg_a};\mathtt{actseq};\ell oc_{\mathrm{b}} := \textbf{newnonce}]_{\mathtt{T}}\;\mathsf{Send}(\mathtt{T},\mathtt{Sto}(\mathtt{T},\mathtt{msg_a}))\lhd\mathsf{NewNonce}(\mathtt{T},\mathtt{Sto}(\mathtt{T},\ell oc_{\mathrm{b}}))$
- $\mathit{true}\,[\ell oc_{\mathrm{a}} := \textbf{receive};\mathtt{actseq};\textbf{send}\;\mathtt{msg_b}]_{\mathtt{T}}\;\mathsf{Receive}(\mathtt{T},\mathtt{Sto}(\mathtt{T},\ell oc_{\mathrm{a}}))\lhd\mathsf{Send}(\mathtt{T},\mathtt{Sto}(\mathtt{T},\mathtt{msg_b}))$
- $\mathit{true}\,[\ell oc_{\mathrm{a}} := \textbf{receive};\mathtt{actseq};\ell oc_{\mathrm{b}} := \textbf{receive}]_{\mathtt{T}}\;\mathsf{Receive}(\mathtt{T},\mathtt{Sto}(\mathtt{T},\ell oc_{\mathrm{a}}))\lhd\mathsf{Receive}(\mathtt{T},\mathtt{Sto}(\mathtt{T},\ell oc_{\mathrm{b}}))$
- $\mathit{true}\,[\ell oc_{\mathrm{a}} := \textbf{receive};\mathtt{actseq};\ell oc_{\mathrm{b}} := \textbf{newnonce}]_{\mathtt{T}}\;\mathsf{Receive}(\mathtt{T},\mathtt{Sto}(\mathtt{T},\ell oc_{\mathrm{a}}))\lhd\mathsf{NewNonce}(\mathtt{T},\mathtt{Sto}(\mathtt{T},\ell oc_{\mathrm{b}}))$
- $\mathit{true}\,[\ell oc_{\mathrm{a}} := \textbf{newnonce};\mathtt{actseq};\textbf{send}\;\mathtt{msg_b}]_{\mathtt{T}}\;\mathsf{NewNonce}(\mathtt{T},\mathtt{Sto}(\mathtt{T},\ell oc_{\mathrm{a}}))\lhd\mathsf{Send}(\mathtt{T},\mathtt{Sto}(\mathtt{T},\mathtt{msg_b}))$
- $\mathit{true}\,[\ell oc_{\mathrm{a}} := \textbf{newnonce};\mathtt{actseq};\ell oc_{\mathrm{b}} := \textbf{receive}]_{\mathtt{T}}\;\mathsf{NewNonce}(\mathtt{T},\mathtt{Sto}(\mathtt{T},\ell oc_{\mathrm{a}}))\lhd\mathsf{Receive}(\mathtt{T},\mathtt{Sto}(\mathtt{T},\ell oc_{\mathrm{b}}))$
- $\mathit{true}\,[\ell oc_{\mathrm{a}} := \textbf{newnonce};\mathtt{actseq};\ell oc_{\mathrm{b}} := \textbf{newnonce}]_{\mathtt{T}}\;\mathsf{NewNonce}(\mathtt{T},\mathtt{Sto}(\mathtt{T},\ell oc_{\mathrm{a}}))\lhd\mathsf{NewNonce}(\mathtt{T},\mathtt{Sto}(\mathtt{T},\ell oc_{\mathrm{b}}))$
- *similarly for other* `action_formula`*s*

Proof: The semantics of $\lhd$-*nonmodal_formula*s is defined by the temporal order between the indicated **newnonce**, **send**, or **receive** (or other) *action*s. However, the required order is guaranteed by any explicit action sequence that lists the actions explicitly in this order.

Axiom schema AN1. The thread generating a given nonce is unique:

- $\mathsf{NewNonce}(\mathtt{T_1},\mathtt{msg}) \wedge \mathsf{NewNonce}(\mathtt{T_2},\mathtt{msg}) \Rightarrow \mathtt{T_1} = \mathtt{T_2}$

Axiom schema AN2. Only the generating thread has the nonce just after generation:

- $\mathit{true}\,[\mathtt{actseq};\ell oc := \textbf{newnonce}]_{\mathtt{T}}\;\mathsf{Has}(\mathtt{T}',\mathtt{Sto}(\mathtt{T},\ell oc)) \Rightarrow \mathtt{T} = \mathtt{T}'$

Axiom schema AN3. A nonce is fresh just after generation:

- $\mathit{true}\,[\mathtt{actseq};\ell oc := \textbf{newnonce}]_{\mathtt{T}}\;\mathsf{Fresh}(\mathtt{T},\mathtt{Sto}(\mathtt{T},\ell oc))$

Proof: The semantic conditions on $\mathsf{Fresh}(\mathtt{T},\mathtt{Sto}(\mathtt{T},\ell oc))$ require that *thread* `T` executed a corresponding **newnonce** *action* and never sent out any *message* containing the new nonce. The former holds because $\ell oc :=$ **newnonce** is listed as an action executed by thread `T` in the modal formula; and the latter holds because this action is listed as the *only* action executed by thread `T`.

Axiom schema AN4. A thread has a fresh nonce only if it generated it:

- $\mathsf{Fresh}(\mathtt{T},\mathtt{msg}) \Rightarrow \mathsf{NewNonce}(\mathtt{T},\mathtt{msg})$

Axiom schemas KEY. The axiom schemas for possession and secrecy of keys distributed according to the protocol setup assumptions depend on the setup assumptions. For any key function xk, if the setup assumptions include the specification $\forall(\mathtt{T},\mathtt{Pname}) : \mathtt{T}.\mathit{pname} = \mathtt{Pname} \Rightarrow \mathit{Setup}(\mathtt{T}, xk(\mathtt{Pname}))$, stating that $xk(\mathtt{Pname})$ is known to threads of that principal, then we take all instances of the schema $\mathsf{Has}(\mathtt{T}, xk(\mathtt{T}.\mathit{pname}))$ as axioms. Similarly, if the setup assumptions include the specification $\forall(\mathtt{T},\mathtt{Pname}) : \mathit{Setup}(\mathtt{T}, xk(\mathtt{Pname}))$, stating that $xk(\mathtt{Pname})$ is known to threads of all principals, then we take all instances of the schema $\mathsf{Has}(\mathtt{T}, xk(\mathtt{Pname}))$ as axioms. We also take secrecy formulas as axiom schemas for each key that is known to the principal that owns it but not known to others. For the setup assumptions used for protocols in this chapter (given in Section 3.1), we therefore have the following axiom schemas about key possession and secrecy:

- $\mathsf{Has}(\mathtt{T}, pk(\mathtt{Pname}))$
- $\mathsf{Has}(\mathtt{T}, dk(\mathtt{T}.\mathit{pname}))$
- $\mathsf{Has}(\mathtt{T}, sk(\mathtt{T}.\mathit{pname}))$
- $\mathsf{Has}(\mathtt{T}, vk(\mathtt{Pname}))$
- $\mathsf{Has}(\mathtt{T}, dk(\mathtt{Pname})) \wedge \mathsf{Honest}(\mathtt{Pname}) \Rightarrow \mathtt{T}.\mathit{pname} = \mathtt{Pname}$
- $\mathsf{Has}(\mathtt{T}, sk(\mathtt{Pname})) \wedge \mathsf{Honest}(\mathtt{Pname}) \Rightarrow \mathtt{T}.\mathit{pname} = \mathtt{Pname}$

Axiom schema HAS. The axiom schemas for the Has predicate are partly based on Dolev-Yao deduction rules:

- $\forall \mathtt{msg}$ that are strings: ‘’, ‘a’, ‘ca’, ‘hello’, … $:> \mathsf{Has}(\mathtt{T}, \mathtt{msg})$
- $\mathsf{Has}(\mathtt{T}, \mathtt{Pname})$
- $\mathsf{NewNonce}(\mathtt{T}, \mathtt{N}) \Rightarrow \mathsf{Has}(\mathtt{T}, \mathtt{N})$
- $\mathsf{Receive}(\mathtt{T}, \mathtt{msg}) \Rightarrow \mathsf{Has}(\mathtt{T}, \mathtt{msg})$
- $\mathsf{Has}(\mathtt{T}, \langle\,\rangle)$
- $\forall k \in \mathbb{N} :> \mathsf{Has}(\mathtt{T}, \mathtt{msg}_1) \wedge \mathsf{Has}(\mathtt{T}, \mathtt{msg}_2) \wedge \ldots \wedge \mathsf{Has}(\mathtt{T}, \mathtt{msg}_k) \Rightarrow \mathsf{Has}(\mathtt{T}, \langle \mathtt{msg}_1, \mathtt{msg}_2, \ldots, \mathtt{msg}_k \rangle)$
- $\forall k \in \mathbb{N} :> \mathsf{Has}(\mathtt{T}, \langle \mathtt{msg}_1, \mathtt{msg}_2, \ldots, \mathtt{msg}_k \rangle) \Rightarrow \mathsf{Has}(\mathtt{T}, \mathtt{msg}_1) \wedge \mathsf{Has}(\mathtt{T}, \mathtt{msg}_2) \wedge \ldots \wedge \mathsf{Has}(\mathtt{T}, \mathtt{msg}_k)$
- $\forall \mathtt{K}{:}\, \mathit{asym\_enc\_key} :> \mathsf{Has}(\mathtt{T}, \mathtt{msg}) \wedge \mathsf{Has}(\mathtt{T}, \mathtt{K}) \Rightarrow \mathsf{Has}(\mathtt{T}, \{\!|\mathtt{msg}|\!\}^{\mathsf{a}}_{\mathtt{K}})$
- $\forall(\mathtt{K}, \mathtt{K}')$ such that $(\mathtt{K}, \mathtt{K}')$ is an asymmetric encryption-decryption *key* pair :>
  $\mathsf{Has}(\mathtt{T}, \{\!|\mathtt{msg}|\!\}^{\mathsf{a}}_{\mathtt{K}}) \wedge \mathsf{Has}(\mathtt{T}, \mathtt{K}') \Rightarrow \mathsf{Has}(\mathtt{T}, \mathtt{msg})$
- $\forall \mathtt{N}{:}\, \mathit{nonce} :> \mathsf{Has}(\mathtt{T}, \mathtt{msg}) \wedge \mathsf{Has}(\mathtt{T}, \mathtt{N}) \Rightarrow \mathsf{Has}(\mathtt{T}, \{\!|\mathtt{msg}|\!\}^{\mathsf{s}}_{\mathtt{N}})$
- $\forall \mathtt{K}{:}\, \mathit{sym\_key} :> \mathsf{Has}(\mathtt{T}, \mathtt{msg}) \wedge \mathsf{Has}(\mathtt{T}, \mathtt{K}) \Rightarrow \mathsf{Has}(\mathtt{T}, \{\!|\mathtt{msg}|\!\}^{\mathsf{s}}_{\mathtt{K}})$
- $\mathsf{Has}(\mathtt{T}, \{\!|\mathtt{msg}|\!\}^{\mathsf{s}}_{\mathtt{N}}) \wedge \mathsf{Has}(\mathtt{T}, \mathtt{N}) \Rightarrow \mathsf{Has}(\mathtt{T}, \mathtt{msg})$
- $\mathsf{Has}(\mathtt{T}, \{\!|\mathtt{msg}|\!\}^{\mathsf{s}}_{\mathtt{K}}) \wedge \mathsf{Has}(\mathtt{T}, \mathtt{K}) \Rightarrow \mathsf{Has}(\mathtt{T}, \mathtt{msg})$
- $\forall \mathtt{K}{:}\, \mathit{sgn\_key} :> \mathsf{Has}(\mathtt{T}, \mathtt{msg}) \wedge \mathsf{Has}(\mathtt{T}, \mathtt{K}) \Rightarrow \mathsf{Has}(\mathtt{T}, [\![\mathtt{msg}]\!]_{\mathtt{K}})$
- $\mathsf{Has}(\mathtt{T}, [\![\mathtt{msg}]\!]_{\mathtt{K}}) \Rightarrow \mathsf{Has}(\mathtt{T}, \mathtt{msg})$

Axiom schema SEC. For protocols with confidential private keys, i.e., $dk(A)$ declared in the setup assumptions, the only principal which can decrypt with its own private key is the principal itself, provided it is honest:

- $\mathsf{Honest}(\mathrm{B}) \wedge \mathsf{Dec}(\mathtt{T}, \{\!|\mathtt{msg}|\!\}^{\mathsf{a}}_{pk(\mathrm{B})}, dk(\mathrm{B})) \Rightarrow \mathtt{T}.\mathit{pname} = \mathrm{B}$

Axiom schema VER. For protocols with confidential signing keys, i.e., $sk(A)$ declared in the setup assumptions, the only principal which can sign with its own signing key is the principal itself, provided it is honest:

- $\mathsf{Honest}(\mathrm{A}) \wedge \mathsf{Verify}(\mathtt{T}, [\![\mathtt{msg}]\!]_{sk(\mathrm{A})}, vk(\mathrm{A})) \Rightarrow \exists \mathtt{T}' : \mathtt{T}'\texttt{.}\mathit{pname} = \mathrm{A} \wedge \mathsf{Sign}(\mathtt{T}', \mathtt{msg}, sk(\mathrm{A}))$

Proof: Let $\mathcal{R}$ be any *run*. If $\mathcal{R} \models \mathsf{Verify}(\mathtt{T}, [\![\mathtt{msg}]\!]_{sk(\mathrm{A})}, vk(\mathrm{A}))$, then by definition $\mathtt{T}$ executed a **verify** $\mathtt{msg}'_{\text{signed}}, \mathtt{K}'$ *action* such that $\mathcal{R}[[\![\mathtt{msg}]\!]_{sk(\mathrm{A})}] \cong \mathtt{msg}'_{\text{signed}}$ and $\mathcal{R}[vk(\mathrm{A})] \cong \mathtt{K}'$. By the **verify**-clause in the definition of feasibility of runs, we know that $\mathtt{msg}'_{\text{signed}}$ must have been of the form $[\![\mathtt{msg}_1]\!]_{\mathtt{K}'_1}$ with $[\![\mathtt{msg}_1]\!]_{\mathtt{K}'_1} \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{T}\texttt{.}\mathit{tid})$. Because of $[\![\mathtt{msg}_1]\!]_{\mathtt{K}'_1} = \mathtt{msg}'_{\text{signed}} \cong \mathcal{R}[[\![\mathtt{msg}]\!]_{sk(\mathrm{A})}]$, it follows that $\mathtt{msg}_1 \cong \mathtt{msg}$ and $\mathtt{K}'_1 \cong sk(\mathrm{A})$.

Let $\mathcal{R}_1$ be the minimum-length run such that $\mathcal{R} = \mathcal{R}_1 : \mathcal{R}_2$ and $[\![\mathtt{msg}_1]\!]_{\mathtt{K}'_1} \Subset \mathtt{msg}_C \in \mathit{GenMsg}_{\mathcal{R}_1}(\mathtt{T}_2\texttt{.}\mathit{tid})$ for some thread $\mathtt{T}_2$ and containing message $\mathtt{msg}_C$. By the way *GenMsg* is defined, there has to be such a first time. By analyzing the set of possible last actions in $\mathcal{R}_1$, it follows that the last action in $\mathcal{R}_1$ is some $\ell oc_2 :=$ **sign** $\mathtt{msg}_2, \mathtt{K}_2$ action such that $[\![\mathtt{msg}_1]\!]_{\mathtt{K}'_1} \cong \mathcal{R}\texttt{.}st(\mathtt{T}_2\texttt{.}\mathit{tid}, \ell oc_2) = [\![\mathtt{msg}_2]\!]_{\mathtt{K}_2}$ (*stored messages* clause, *GenMsg* definition; **sign**-clause, feasibility of runs). That the first time can't have been immediately after something else (projection, **receive**, …) can be shown rigorously through an inductive argument; for instance it couldn't have been a **receive** action for a message containing $[\![\mathtt{msg}_1]\!]_{\mathtt{K}'_1}$ because then there must have been an earlier corresponding **send** action, contradicting minimality of $\mathcal{R}_1$.

The $\ell oc_2 :=$ **sign** $\mathtt{msg}_2, \mathtt{K}_2$ action could only have been executed by *thread* $\mathtt{T}_2$ if at that time it was the case that $\mathtt{K}_2 \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{T}_2\texttt{.}\mathit{tid})$. We also know that $\mathtt{msg}_2 \cong \mathtt{msg}$ and $\mathtt{K}_2 \cong sk(\mathrm{A})$ (because of $\mathtt{msg}_1 \cong \mathtt{msg}$ and $\mathtt{K}'_1 \cong sk(\mathrm{A})$ (both established above) and $[\![\mathtt{msg}_2]\!]_{\mathtt{K}_2} \cong [\![\mathtt{msg}_1]\!]_{\mathtt{K}'_1}$). Thus, from the action execution we first get $\mathcal{R} \models \mathsf{Sign}(\mathtt{T}_2, \mathtt{msg}_2, \mathtt{K}_2)$, which leads us to $\mathcal{R} \models \mathsf{Sign}(\mathtt{T}_2, \mathtt{msg}, sk(\mathrm{A}))$ (transitivity of congruence relations and truth conditions of the Sign predicate).

From (1) $sk(\mathrm{A}) \cong \mathtt{K}_2 \in \mathit{GenMsg}_{\mathcal{R}}(\mathtt{T}_2\texttt{.}\mathit{tid})$, (2) $\mathsf{Honest}(A)$ (here, finally *honesty* comes into play), and (3) $sk(\mathrm{A}) : \mathit{conf\_sgn\_key}$ (from our setup assumptions), it follows that $\mathtt{T}_2\texttt{.}\mathit{pname} = \mathrm{A}$. Existential abstraction leads to the consequent of part of the axiom schema whose proof we are hereby concluding.

Axiom schema IN. This axiom schema provides reasoning principles for when one message is contained in another:

- $\mathtt{msg} \Subset \mathtt{msg}$
- $\mathtt{msg} \Subset \{\!|\mathtt{msg}|\!\}^{\mathsf{a}}_{\mathtt{K}} \wedge \mathtt{K} \Subset \{\!|\mathtt{msg}|\!\}^{\mathsf{a}}_{\mathtt{K}}$
- $\mathtt{msg} \Subset \{\!|\mathtt{msg}|\!\}^{\mathsf{s}}_{\mathtt{K}} \wedge \mathtt{K} \Subset \{\!|\mathtt{msg}|\!\}^{\mathsf{s}}_{\mathtt{K}}$
- $\mathtt{msg} \Subset \langle \ldots, \mathtt{msg}, \ldots \rangle$
- $\mathtt{msg} \Subset \{\!|\mathtt{msg}|\!\}^{\mathsf{s}}_{\mathtt{N}} \wedge \mathtt{N} \Subset \{\!|\mathtt{msg}|\!\}^{\mathsf{s}}_{\mathtt{N}}$
- $\mathtt{msg} \Subset [\![\mathtt{msg}]\!]_{\mathtt{K}} \wedge \mathtt{K} \Subset [\![\mathtt{msg}]\!]_{\mathtt{K}}$

Axiom schema NIN. This axiom schema provides reasoning principles for when one message is not contained in another:

- $\mathtt{msg}$ is atomic $:> \mathtt{msg}' \ncong \mathtt{msg} \Rightarrow \mathtt{msg}' \not\Subset \mathtt{msg}$
- $\mathtt{msg} \not\Subset \mathtt{msg}_1 \wedge \mathtt{msg} \not\Subset \mathtt{msg}_2 \wedge \ldots \wedge \mathtt{msg} \not\Subset \mathtt{msg}_n \wedge \mathtt{msg} \ncong \langle \mathtt{msg}_1, \mathtt{msg}_2, \ldots, \mathtt{msg}_n \rangle$
  $\Rightarrow \mathtt{msg} \not\Subset \langle \mathtt{msg}_1, \mathtt{msg}_2, \ldots, \mathtt{msg}_n \rangle$
- $\mathtt{msg}' \not\Subset \mathtt{msg} \wedge \mathtt{msg}' \not\Subset \mathtt{K} \wedge \mathtt{msg}' \ncong \{\!|\mathtt{msg}|\!\}^{\mathsf{a}}_{\mathtt{K}} \Rightarrow \mathtt{msg}' \not\Subset \{\!|\mathtt{msg}|\!\}^{\mathsf{a}}_{\mathtt{K}}$
- $\mathtt{msg}' \not\Subset \mathtt{msg} \wedge \mathtt{msg}' \not\Subset \mathtt{N} \wedge \mathtt{msg}' \ncong \{\!|\mathtt{msg}|\!\}^{\mathsf{s}}_{\mathtt{N}} \Rightarrow \mathtt{msg}' \not\Subset \{\!|\mathtt{msg}|\!\}^{\mathsf{s}}_{\mathtt{N}}$
- $\mathtt{msg}' \not\Subset \mathtt{msg} \wedge \mathtt{msg}' \not\Subset \mathtt{K} \wedge \mathtt{msg}' \ncong \{\!|\mathtt{msg}|\!\}^{\mathsf{s}}_{\mathtt{K}} \Rightarrow \mathtt{msg}' \not\Subset \{\!|\mathtt{msg}|\!\}^{\mathsf{s}}_{\mathtt{K}}$
- $\mathtt{msg}' \not\Subset \mathtt{msg} \wedge \mathtt{msg}' \not\Subset \mathtt{K} \wedge \mathtt{msg}' \ncong [\![\mathtt{msg}]\!]_{\mathtt{K}} \Rightarrow \mathtt{msg}' \not\Subset [\![\mathtt{msg}]\!]_{\mathtt{K}}$

Axiom schema P1. The predicates Has, Sign, $\mathsf{NewNonce}$, and $\mathsf{FirstSend}$ are preserved across actions:

- $\forall \varphi$ that are of type Has, Sign, $\mathsf{NewNonce}$, or $\mathsf{FirstSend}$ $:> \varphi\,[\mathtt{actseq}]_{\mathtt{T}}\,\varphi$

Axiom schema P2.1. The freshness of a message is preserved across non-send actions:

- $\forall \mathtt{actseq}$ that do not contain a **send** *action* $:> \mathsf{Fresh}(\mathtt{T},\mathtt{msg})\,[\mathtt{actseq}]_{\mathtt{T}}\,\mathsf{Fresh}(\mathtt{T},\mathtt{msg})$

Axiom schema P2.2. The freshness of a message is preserved across a send action that does not send something containing the message:

- $\mathsf{Fresh}(\mathtt{T},\mathtt{msg})\;\;[\textbf{send}\;\mathtt{msg}']_{\mathtt{T}}\,\mathtt{msg} \not\Subset \mathtt{Sto}(\mathtt{T},\mathtt{msg}') \Rightarrow \mathsf{Fresh}(\mathtt{T},\mathtt{msg})$

Axiom schema FS1. A nonce is "sent first" when it is fresh and the first send action with a message containing the nonce occurs:

- $\mathsf{Fresh}(\mathtt{T},\mathtt{msg}_1)\;\;[\textbf{send}\;\mathtt{msg}_2]_{\mathtt{T}}\,(\mathtt{msg}_1 \Subset \mathtt{Sto}(\mathtt{T},\mathtt{msg}_2)$
  $\Rightarrow \mathsf{NewNonce}(\mathtt{T},\mathtt{msg}_1) \wedge \mathsf{FirstSend}(\mathtt{T},\mathtt{msg}_1,\mathtt{Sto}(\mathtt{T},\mathtt{msg}_2)))$

Proof: For the *action_formula* FirstSend to hold, two conditions must be fulfilled: (1) $\mathtt{msg}_1 \Subset \mathtt{Sto}(\mathtt{T},\mathtt{msg}_2)$. (2) There is an instance of *thread* $\mathtt{T}$ sending out $\mathtt{Sto}(\mathtt{T},\mathtt{msg}_2)$ such that this is the first instance of any *message* being sent out such that $\mathtt{msg}_1$ is a part of it. Condition (1) holds by the way the axiom schema is stated. Condition (2) holds because before the **send** *action*, $\mathtt{msg}_1$ was fresh.

Axiom schema FS2. If a thread first sent a nonce $\mathtt{msg}_1$ as part of $\mathtt{msg}_2$ and another thread received a $\mathtt{msg}_3$ containing $\mathtt{msg}_1$, then the receive action occurred after the send action:

- $((\mathsf{NewNonce}(\mathtt{T}_1,\mathtt{msg}_1) \wedge \mathsf{FirstSend}(\mathtt{T}_1,\mathtt{msg}_1,\mathtt{msg}_2))$
  $\wedge(\mathtt{T}_1 \neq \mathtt{T}_2 \wedge \mathtt{msg}_1 \Subset \mathtt{msg}_3 \wedge \mathsf{Receive}(\mathtt{T}_2,\mathtt{msg}_3))$
  $\Rightarrow \mathsf{Send}(\mathtt{T}_1,\mathtt{msg}_2) \lhd \mathsf{Receive}(\mathtt{T}_2,\mathtt{msg}_3))$

Proof: $\mathtt{T}_1$ is the thread in which $\mathtt{msg}_1$ originates and $\mathtt{msg}_2$ is the first term sent to the adversary that contains $\mathtt{msg}_1$. Therefore no message containing $\mathtt{msg}_1$ is in the Dolev-Yao closure of the adversary before the send event $\mathsf{Send}(\mathtt{T}_1,\mathtt{msg}_2)$. Since $\mathtt{T}_2$ is different from $\mathtt{T}_1$, $\mathtt{msg}_1$ cannot have been originated in it. Therefore, since $\mathtt{T}_2$ knows the message $\mathtt{msg}_3$ which contains $\mathtt{msg}_1$, it must have received a message $\mathtt{msg}$ from the adversary containing $\mathtt{msg}_1$. As observed before, this is only possible after $\mathtt{T}_1$ sent out $\mathtt{msg}_2$. Therefore $\mathsf{Send}(\mathtt{T}_1,\mathtt{msg}_2) \lhd \mathsf{Receive}(\mathtt{T}_2,\mathtt{msg}_3)$ is true in this $\mathcal{R}$.

5.2. Rule Schemas

The rule schemas include generic rules **G1–G4** for reasoning about program preconditions and postconditions, a sequencing rule **SEQ**, and a novel honesty rule **HON** for establishing protocol invariants in the presence of adversaries.

Rule schema G1. Conjunction of post-conditions:

$$\frac{\varphi\,[\mathtt{actseq}]_{\mathtt{T}}\,\psi_1 \qquad \varphi\,[\mathtt{actseq}]_{\mathtt{T}}\,\psi_2}{\varphi\,[\mathtt{actseq}]_{\mathtt{T}}\,\psi_1 \wedge \psi_2}$$

Rule schema G2. Disjunction of pre-conditions:

$$\frac{\varphi_1\,[\mathtt{actseq}]_{\mathtt{T}}\,\psi \qquad \varphi_2\,[\mathtt{actseq}]_{\mathtt{T}}\,\psi}{\varphi_1 \vee \varphi_2\,[\mathtt{actseq}]_{\mathtt{T}}\,\psi}$$

Rule schema G3. Strengthening pre-conditions and weakening post-conditions:

$$\frac{\varphi' \Rightarrow \varphi \qquad \varphi\,[\mathtt{actseq}]_{\mathtt{T}}\,\psi \qquad \psi \Rightarrow \psi'}{\varphi'\,[\mathtt{actseq}]_{\mathtt{T}}\,\psi'}$$

Rule schema G4. True non-modal formulas spawn modal formulas as their post-condition:

$$\frac{\psi}{\varphi\,[\mathtt{actseq}]_{\mathrm{T}}\,\psi}$$

Rule schema SEQ. Stringing together two action sequences if post-condition of the former and pre-condition of the latter agree:

$$\frac{\varphi_1\,[\mathtt{actseq}]_{\mathrm{T}}\,\varphi_2 \qquad \varphi_2\,[\mathtt{actseq}']_{\mathrm{T}}\,\varphi_3}{\varphi_1\,[\mathtt{actseq} : \mathtt{actseq}']_{\mathrm{T}}\,\varphi_3}$$

Proof: Assume that the premises are valid and that for a given division $(\mathcal{R}_1 : \mathcal{R}_2 : \mathcal{R}_3)$ of *run* $\mathcal{R}$, the following hold:

1. $\mathcal{P} : \mathcal{R}_1 \models \varphi_1$
2. $(\mathcal{R}_2.\mathit{elist}|_{\mathrm{T}} \setminus \mathcal{R}_1.\mathit{elist}|_{\mathrm{T}})$ matches $\mathtt{actseq} : \mathtt{actseq}'$

Then, it is possible to find a division $(\mathcal{R}_1 : \mathcal{R}_2' : \mathcal{R}_2'')$ of *run* $\mathcal{R}_2$ such that:

1. $(\mathcal{R}_2'.\mathit{elist}|_{\mathrm{T}} \setminus \mathcal{R}_1.\mathit{elist}|_{\mathrm{T}})$ matches $\mathtt{actseq}$.
2. $(\mathcal{R}_2''.\mathit{elist}|_{\mathrm{T}} \setminus (\mathcal{R}_1 : \mathcal{R}_2').\mathit{elist}|_{\mathrm{T}})$ matches $\mathtt{actseq}'$.

Then by the first premise we get $\mathcal{P} : \mathcal{R}_2' \models \varphi_2$, and therefore by the second premise we get $\mathcal{P} : \mathcal{R}_2'' \models \varphi_3$. That is, $\mathcal{P} : \mathcal{R}_2 \models \varphi_3$.

Rule schema HON. The honesty rule is an invariance rule for proving properties about the actions of principals that execute roles of a protocol, similar in spirit to the basic invariance rule of LTL [27] and invariance rules in other logics of programs. The honesty rule is often used to combine facts about one role with inferred actions of other roles. For example, suppose Alice receives a signed response from a message sent to Bob. Alice may use facts about Bob's role to infer that Bob must have performed certain actions before sending his reply. This form of reasoning may be sound if Bob is honest, since honest, by definition in our framework, means "follows one or more roles of the protocol". The assumption that Bob is honest is essential because the intruder may perform action sequences that do not conform to the protocol roles.

To a first approximation, the honesty rule says that if a property holds before each role starts, and the property is preserved by any sequence of actions that an honest principal may perform, then the property holds for every honest principal. An example property that can be proved by this method is that if a principal sends a signed message of a certain form, the principal must have received a request for this response. The proof of such a property depends on the protocol. For this reason, the antecedent of the honesty rule includes a set of formulas constructed from the set of roles of the protocol in a systematic way.

One semantic assumption that is effectively built into the honesty rule is that every honest thread that starts a basic sequence complete all of the actions in the basic sequence. We therefore formulate the honesty rule to prove properties that hold at the end of every basic sequence of every role.

The division of a role into basic sequences is part of the protocol specification – the protocol designer, or the protocol analyst who takes a protocol description and formulates the protocol in PCL, may choose how to partition the actions of a role into basic sequences. PCL then reasons about the protocol as if any honest principal that starts a basic sequence does so in a way that allows each projection of a message into parts, each assert, and each cryptographic operation to succeed. An actual implementation of a protocol may respect this abstraction by first determining whether the conditions exist to complete a basic sequence, and then proceeding with visible actions (such as sending messages) only under these conditions. If the conditions do not allow the basic sequence to complete, then the implementation may locally "roll back" to a state as if the basic sequence never started. Put another way, we reason about the set of protocol executions by assuming atomicity of basic sequences of actions. Since a protocol may be formulated with any desired partition of roles into basic sequences, this is not a limitation of PCL, but a feature that allows atomicity assumptions to be used in protocol proofs.

Note that all elements of a $\mathtt{protocol}\ \mathcal{P}$ are of type $\mathit{List}(\mathtt{principal\_name}) \to \mathtt{role}$. We now state the honesty rule:

$$\frac{\mathsf{Start}(\mathtt{T})\,[\,]_{\mathtt{T}}\,\varphi \qquad \forall \mathtt{p}\in\mathcal{P} : \forall \mathtt{basicseq}\in(\mathtt{p}(\mathtt{T}.\mathit{rpars})).\mathit{bseqs} :> \varphi\,[\mathtt{basicseq}]_{\mathtt{T}}\,\varphi}{\mathsf{Honest}(\mathtt{T}.\mathit{pname}) \Rightarrow \varphi}$$

(Here, we assume that the role parameters in $\mathtt{T}.\mathit{rpars}$ are all distinct symbols.) Note that the universal quantifiers in this rule expand to a finite conjunction, and thus the rule can be applied in a finite proof. An example is given in the next section.

Proof: This soundness proof is similar in spirit to the proof of **SEQ**. Consider a run $\mathcal{R}$ and a thread $\mathtt{T}$, such that the premises hold. Since $\mathtt{T}$ belongs to an honest principal, $\mathtt{T}$ must have performed a sequence of actions which is a (possibly empty) concatenation of basic sequences of some instantiated role $\mathtt{p}(\mathtt{T}.\mathit{rpars})$. Let these basic sequences be $\mathtt{basicseq}_1, \mathtt{basicseq}_2, \cdots, \mathtt{basicseq}_k$ (in the order executed). Then it is possible to find a division $(\mathcal{R}_0 : \mathcal{R}_1 : \ldots : \mathcal{R}_k)$ of run $\mathcal{R}$, such that $\mathcal{R}_0.\mathit{elist}|_{\mathtt{T}}$ is empty, and for all $i \in \{1, 2, \ldots, k\}$, the list of events $(\mathcal{R}_i.\mathit{elist}|_{\mathtt{T}} \setminus \mathcal{R}_{(i-1)}.\mathit{elist}|_{\mathtt{T}})$ *matches* the action sequence $\mathtt{basicseq}_i$.

By the first premise we have $\mathcal{P} : \mathcal{R}_0 \models \varphi$. By successively applying all parts of the second premise, we first get $\mathcal{P} : \mathcal{R}_1 \models \varphi$, then $\mathcal{P} : \mathcal{R}_2 \models \varphi$, and finally $\mathcal{P} : \mathcal{R}_k \models \varphi$. The latter is identical to $\mathcal{P} : \mathcal{R} \models \varphi$, quod erat demonstrandum.

6. Analysis of the Handshake Protocol

In this section, we present the PCL authentication proof for the handshake protocol introduced in Section 2 and written out formally in Table 2. We also discuss how the proof fails for the faulty version of the sample protocol.

6.1. Properties

Our formulation of authentication is based on the concept of *matching conversations* [3] and is similar to the idea of proving authentication using *correspondence assertions* [45]. The same basic idea is also presented in [13] where it is referred to as *matching records of runs*. Intuitively, this form of authentication requires that whenever Alice and Bob complete their roles, their views of the run must *match* in the sense that both have the same record of the actions that occurred and the temporal order between them. More specifically, each message that Alice sent was received by Bob and vice versa, each send event happened before the corresponding receive event, and so on. In general, authentication properties are formulated and proved for both the initiator and the responder. However, in this chapter, we prove authentication from the responder's point of view only because of space constraints. The authentication property $Auth_{Resp}$ is defined in Figure 4.

The actions in the modal formula $Auth_{Resp}$ are the actions of the responder role of the protocol. In this example, the precondition is simply *true*. The postcondition assumes that `idA` is honest, so that she faithfully executes portions of some role of the protocol and does not, for example, send out her private keys. Under this assumption, the postcondition means that after executing the actions in the initiator role purportedly with `idA`, B is guaranteed that `idA` was involved in the protocol and messages were sent and received in the expected order.

We also show here a secrecy property for this protocol, but omit proofs. A proof system to analyze secrecy properties has been developed for an earlier syntax and semantics of PCL [38,39] but is not covered in this chapter.

$$Secrecy_{Init}: \quad true[\mathbf{Init}(A,B)]_T\ (\mathsf{Honest}(A) \wedge \mathsf{Honest}(B) \wedge A \neq B \\ \wedge \mathsf{Has}(T', k) \Rightarrow T'.pname = A \vee T'.pname = B)$$

(An analogous property $Secrecy_{Resp}$ can be defined easily.) The secrecy formula above asserts that if A completes an initiator role, purportedly with B, then if B is honest, it is guaranteed that the only threads which can have `k` belong to principals A or B only. Similar secrecy properties can be formulated for nonce `s` as well.

6.2. Proof of $Auth_{Resp}$

The main idea of the proof is that the responder verifies a signature using the public verification key of the initiator. If the initiator is assumed honest, then by a basic property of signatures captured in one of the PCL axioms, the initiator must have signed the verified message. Using the honesty rule to consider all possible action sequences that allow an honest thread to sign a message, we show that if the initiator signed the verified message, then the initiator must have done specific actions in a specific order that lets us complete the proof.

The main steps[2] of the formal proof are presented in Table 11, with uses of rule schema G left implicit. The formal proof uses expressions such as $T.pname$ ($=T.rpars.1$) and $T.rpars.2$ for principals. For any thread T, the principal $T.pname$ is the principal who executes this thread. In addition, if T is executed by an honest principal, then $T.rpars.1$ and $T.rpars.2$ are the principals supplied as first and second parameters to the parameterized role associated with this thread. While for the formal proof it is technically precise (and convenient) to use $T.rpars.1$ and $T.rpars.2$, note that we often refer to these parameters as A (the initiator) and B (the intended responder). Using this terminology, the proof can be divided into three parts.

- Lines (1)–(3): We show that certain actions were performed by B in the responder role. Specifically, we show that B verified A's signature `siga`. We then use the signature verification axiom (**VER**) to prove that, if honest, A must have signed this message.
- In lines (4)–(5), the honesty rule is used to prove that since A signs a message of the indicated form, she first sent `k` as part of the first message of the protocol.

[2]A full proof together with a sample execution trace for the handshake protocol can be found here: `http://seclab.stanford.edu/pcl/papers/2010-06_execution-trace_and_full-proof.pdf`

1. For the second basic sequence of **Init** and the first basic sequence of **Resp** (which is the full role), there is no signing action, so we can prove[3] (writing $\mathsf{N\&FS}(\ldots)$ for $\mathsf{NewNonce}(\ldots) \wedge \mathsf{FirstSend}(\ldots)$):
   (AA3:) $\neg\mathsf{Sign}(\ldots)[\mathtt{basicseq}]\neg\mathsf{Sign}(\ldots)$
   (P1:) $\mathsf{Sign}(\ldots) \wedge \mathsf{N\&FS}(\ldots)[\mathtt{basicseq}]\mathsf{Sign}(\ldots) \wedge \mathsf{N\&FS}(\ldots)$
   We use G3 on both separately (by weakening the post-conditions), and then we combine the results using G2:
   $\mathsf{Sign}(\ldots) \Rightarrow \mathsf{N\&FS}(\ldots)[\mathtt{basicseq}]\mathsf{Sign}(\ldots) \Rightarrow \mathsf{N\&FS}(\ldots)$
2. For the first basic sequence of **Init**, we prove the following:
   (AA1, AN3, P2.1, FS1, IN, SEQ:) $\mathit{true}[\mathtt{basicseq}]\mathsf{Sign}(\ldots) \wedge \mathsf{N\&FS}(\ldots)$
   (G3:) $\mathsf{Sign}(\ldots) \Rightarrow \mathsf{N\&FS}(\ldots)[\mathtt{basicseq}]\mathsf{Sign}(\ldots) \Rightarrow \mathsf{N\&FS}(\ldots)$

- Finally, in lines (7)–(8), we again reason about actions executed by B in order to deduce that the the first message was sent by A before it was received by B. Combining the assertions, we show that the authentication property holds: If B has completed the protocol as a responder, apparently with A, then A must have sent the first message (intended for B), and then B sent the second message to A.

In many cases, attempting to prove incorrect security properties in PCL may suggest ways that a protocol may be repaired to achieve the desired property. If we attempt to carry out the proof shown in Table 11 for the flawed protocol discussed in section 2.1 (and in other chapters of this book), the attempt fails. In the failed attempt, the closest formula we can prove using the honesty rule fails to sufficiently link the Initiator and Responder. Specifically, the provable version of (4) asserts that any honest message signer will first send the signed message under a specific encryption key (written $pk\,(\mathtt{T}_{\mathrm{all}}\,\mathtt{.}\mathit{rpars.2})$):

$$\begin{aligned}&\mathsf{Honest}(\mathtt{T}_{\mathrm{all}}\,\mathtt{.}\mathit{pname}) \wedge \mathsf{Sign}(\mathtt{T}_{\mathrm{all}}, \left\langle \mathtt{T}_{\mathrm{all}}\,\mathtt{.}\mathit{pname}, \mathrm{k}^{[\mathtt{T}_{\mathrm{all}}]}\right\rangle, sk\,(\mathtt{T}_{\mathrm{all}}\,\mathtt{.}\mathit{pname}))\\ &\quad\Rightarrow \mathsf{NewNonce}(\mathtt{T}_{\mathrm{all}}, \mathrm{k}^{[\mathtt{T}_{\mathrm{all}}]})\\ &\quad\wedge\mathsf{FirstSend}\left(\mathtt{T}_{\mathrm{all}}, \mathrm{k}^{[\mathtt{T}_{\mathrm{all}}]}, \left\{\!\left|[\![\left\langle \mathtt{T}_{\mathrm{all}}\,\mathtt{.}\mathit{pname}, \mathrm{k}^{[\mathtt{T}_{\mathrm{all}}]}\right\rangle]\!]_{sk(\mathtt{T}_{\mathrm{all}}\,\mathtt{.}\mathit{pname})}\right|\!\right\}^{\mathrm{a}}_{pk(\mathtt{T}_{\mathrm{all}}\,\mathtt{.}\mathit{rpars.2})}\right)\end{aligned}$$

However, when we proceed with this formula, the analog of line (5) has an encryption key in the message sent by the initiator that is not linked to $\mathtt{T}$:

$$\begin{aligned}&\mathit{true}\;[\mathbf{Resp}(\mathtt{T.}\mathit{pname})]_{\mathtt{T}}\;\mathsf{Honest}(\mathtt{idA}^{[\mathtt{T}]})\\ &\Rightarrow \exists \mathtt{T}' :\; \mathtt{T}'\mathtt{.}\mathit{pname} = \mathtt{idA}^{[\mathtt{T}]} \wedge \mathsf{NewNonce}(\mathtt{T}', \mathrm{k}^{[\mathtt{T}]})\\ &\wedge\mathsf{FirstSend}\left(\mathtt{T}', \mathrm{k}^{[\mathtt{T}]}, \left\{\!\left|[\![\left\langle \mathtt{idA}^{[\mathtt{T}]}, \mathrm{k}^{[\mathtt{T}]}\right\rangle]\!]_{sk\left(\mathtt{idA}^{[\mathtt{T}]}\right)}\right|\!\right\}^{\mathrm{a}}_{pk(\mathtt{T}'\mathtt{.}\mathit{rpars.2})}\right)\end{aligned}$$

Instead of proving that the honest principal $\mathtt{idA}^{[\mathtt{T}]}$ encrypted the message for thread $\mathtt{T}$ ($= \mathtt{T.}\mathit{rpars.1}$), we can only conclude that some thread $\mathtt{T}'$ encrypted the signed message for some unconstrained role parameter $\mathtt{T}'\mathtt{.}\mathit{rpars.2}$ of that thread that could be different from the responder. In other words, because the initiator does not include the identity of the intended responder in the signed message, the responder that receives this message cannot deduce that the message was intended (encrypted) for him. This failure of the proof also suggests the attack scenario in which an initiator executing role $\mathbf{Init}(\mathrm{A}, \mathrm{C})$ encrypts the signed message for C, who turns out to be dishonest. Dishonest C then decrypts A's message and re-encrypts it for B. When responder B receives this message, the responder has no guarantee that A intended (encrypted) the message for B.

[3]Note that the implication we are proving is interesting only for **Init**; it is trivially true for **Resp**. Because the honesty rule is used to prove authentication when communicating with any honest principal executing any combination of roles, the honesty rule requires us to consider all roles in the proof.

AA0	$\mathit{true}\ [\mathbf{Resp}(\mathtt{T}.\mathit{pname})]_{\mathtt{T}}\ \mathtt{siga}^{[\mathtt{T}]} = [\![\langle \mathtt{idA}^{[\mathtt{T}]}, \mathtt{T}.\mathit{rpars.1}, \mathtt{k}^{[\mathtt{T}]} \rangle]\!]_{sk(\mathtt{idA}^{[\mathtt{T}]})}$	(1)
AA1, (1)	$\mathit{true}\ [\mathbf{Resp}(\mathtt{T}.\mathit{pname})]_{\mathtt{T}}\ \mathsf{Verify}\left(\mathtt{T}, [\![\langle \mathtt{idA}^{[\mathtt{T}]}, \mathtt{T}.\mathit{rpars.1}, \mathtt{k}^{[\mathtt{T}]} \rangle]\!]_{sk(\mathtt{idA}^{[\mathtt{T}]})}, vk\left(\mathtt{idA}^{[\mathtt{T}]}\right)\right)$	(2)
VER, (2)	$\mathit{true}\ [\mathbf{Resp}(\mathtt{T}.\mathit{pname})]_{\mathtt{T}}\ \mathsf{Honest}(\mathtt{idA}^{[\mathtt{T}]})$	
	$\Rightarrow \exists \mathtt{T}' : \mathtt{T}'.\mathit{pname} = \mathtt{idA}^{[\mathtt{T}]} \wedge \mathsf{Sign}\left(\mathtt{T}', \langle \mathtt{idA}^{[\mathtt{T}]}, \mathtt{T}.\mathit{rpars.1}, \mathtt{k}^{[\mathtt{T}]} \rangle, sk\left(\mathtt{idA}^{[\mathtt{T}]}\right)\right)$	(3)
AA2, AA1, AN3,	$\mathsf{Honest}(\mathtt{T}_{\mathrm{all}}.\mathit{pname}) \wedge \mathsf{Sign}(\mathtt{T}_{\mathrm{all}}, \langle \mathtt{T}_{\mathrm{all}}.\mathit{pname}, \mathtt{T}_{\mathrm{all}}.\mathit{rpars.2}, \mathtt{k}^{[\mathtt{T}_{\mathrm{all}}]} \rangle, sk\,(\mathtt{T}_{\mathrm{all}}.\mathit{pname}))$	
P2.1, FS1, IN, SEQ,	$\Rightarrow \mathsf{NewNonce}(\mathtt{T}_{\mathrm{all}}, \mathtt{k}^{[\mathtt{T}_{\mathrm{all}}]})$	
AA3, P1, HON	$\wedge\ \mathsf{FirstSend}\left(\mathtt{T}_{\mathrm{all}}, \mathtt{k}^{[\mathtt{T}_{\mathrm{all}}]}, \{\![[\![\langle \mathtt{T}_{\mathrm{all}}.\mathit{pname}, \mathtt{T}_{\mathrm{all}}.\mathit{rpars.2}, \mathtt{k}^{[\mathtt{T}_{\mathrm{all}}]} \rangle]\!]_{sk(\mathtt{T}_{\mathrm{all}}.\mathit{pname})}]\!\}^{\mathsf{a}}_{pk(\mathtt{T}_{\mathrm{all}}.\mathit{rpars.2})}\right)$	(4)
(3), (4)	$\mathit{true}\ [\mathbf{Resp}(\mathtt{T}.\mathit{pname})]_{\mathtt{T}}\ \mathsf{Honest}(\mathtt{idA}^{[\mathtt{T}]})$	
	$\Rightarrow \exists \mathtt{T}' : \mathtt{T}'.\mathit{pname} = \mathtt{idA}^{[\mathtt{T}]} \wedge \mathsf{NewNonce}(\mathtt{T}', \mathtt{k}^{[\mathtt{T}]})$	
	$\wedge\ \mathsf{FirstSend}\left(\mathtt{T}', \mathtt{k}^{[\mathtt{T}]}, \{\![[\![\langle \mathtt{idA}^{[\mathtt{T}]}, \mathtt{T}.\mathit{rpars.1}, \mathtt{k}^{[\mathtt{T}]} \rangle]\!]_{sk(\mathtt{idA}^{[\mathtt{T}]})}]\!\}^{\mathsf{a}}_{pk(\mathtt{T}.\mathit{rpars.1})}\right)$	(5)
AA0, AA1, IN, (1)	$\mathit{true}\ [\mathbf{Resp}(\mathtt{T}.\mathit{pname})]_{\mathtt{T}}\ \mathtt{k}^{[\mathtt{T}]} \Subset \{\![[\![\langle \mathtt{idA}^{[\mathtt{T}]}, \mathtt{T}.\mathit{rpars.1}, \mathtt{k}^{[\mathtt{T}]} \rangle]\!]_{sk(\mathtt{idA}^{[\mathtt{T}]})}]\!\}^{\mathsf{a}}_{pk(\mathtt{T}.\mathit{rpars.1})}$	
	$\wedge\ \mathsf{Receive}\left(\mathtt{T}, \{\![[\![\langle \mathtt{idA}^{[\mathtt{T}]}, \mathtt{T}.\mathit{rpars.1}, \mathtt{k}^{[\mathtt{T}]} \rangle]\!]_{sk(\mathtt{idA}^{[\mathtt{T}]})}]\!\}^{\mathsf{a}}_{pk(\mathtt{T}.\mathit{rpars.1})}\right)$	(6)
FS2, AA0,	$\mathit{true}\ [\mathbf{Resp}(\mathtt{T}.\mathit{pname})]_{\mathtt{T}}\ \mathsf{Honest}(\mathtt{idA}^{[\mathtt{T}]}) \Rightarrow \exists \mathtt{T}' : (\mathtt{T}'.\mathit{pname} = \mathtt{idA}^{[\mathtt{T}]} \wedge (\mathtt{T}' \neq \mathtt{T}$	
(1), (5), (6)	$\Rightarrow \mathsf{Send}(\mathtt{T}', \mathsf{enca}^{[\mathtt{T}]}) \lhd \mathsf{Receive}(\mathtt{T}, \mathsf{enca}^{[\mathtt{T}]})))$	(7)
AA4, (7)	$\mathit{true}\ [\mathbf{Resp}(\mathtt{T}.\mathit{pname})]_{\mathtt{T}}\ \mathsf{Honest}(\mathtt{idA}^{[\mathtt{T}]}) \wedge \mathtt{idA}^{[\mathtt{T}]} \neq \mathtt{T}.\mathit{pname} \Rightarrow \exists \mathtt{T}' : \mathtt{T}'.\mathit{pname} = \mathtt{idA}^{[\mathtt{T}]}$	
	$\wedge\ \mathsf{Send}(\mathtt{T}', \mathsf{enca}^{[\mathtt{T}]}) \lhd \mathsf{Receive}(\mathtt{T}, \mathsf{enca}^{[\mathtt{T}]})$	
	$\wedge\ \mathsf{Receive}(\mathtt{T}, \mathsf{enca}^{[\mathtt{T}]}) \lhd \mathsf{Send}(\mathtt{T}, \mathsf{encb}^{[\mathtt{T}]})$	(8)

Table 11. Formal proof of Auth_{resp}

7. Protocol Composition

Principals may participate in multiple protocols in parallel (e.g. SSL and IPSec), possibly using the same keys. In this section, we define parallel composition of protocols and present a sound method for modular reasoning about their security. Because most of the reasoning principles used in PCL are simple local reasoning about the actions of a thread executing a role of a protocol, such steps are independent of the protocol that uses this role. As a result, some properties proved using PCL are preserved when additional roles are added to the protocol, or if additional actions are added to the end of one or more roles. The one exception to this characterization, and it is a big one, is the honesty rule, which is used to prove properties of one thread based on properties of other honest threads. Therefore, the composition principles of PCL revolve around properties of the honesty rule and its use. For simplicity, we only consider parallel composition, which involves adding new roles, and leave the details of sequential composition, which involves extending individual roles with actions of another role, to other presentations.

Two protocols $\mathcal{P}_1, \mathcal{P}_2$ have *compatible setup assumptions* if the same setup conditions are given in both protocols for every key function that is common to both. For example, $\mathcal{P}_1$ may assume public keys, and $\mathcal{P}_2$ may assume shared symmetric keys (of different names). Alternately, both may use the same setup assumptions. However, two protocols do not have compatible setup assumptions if one assumes a key is confidential and the other allows the key to be sent by honest principals.

Lemma 7.1 *If $\mathcal{P}_1, \mathcal{P}_2$ have compatible setup assumptions, then the PCL assertions provable about $\mathcal{P}_i$ are unchanged if the setup assumptions of $\mathcal{P}_i$ are replaced by the union of the setup assumptions of $\mathcal{P}_1$ and $\mathcal{P}_2$.*

Definition (Parallel Composition) If $\mathcal{P}_1, \mathcal{P}_2$ have compatible setup assumptions, then their parallel composition $\mathcal{P}_1 \mid \mathcal{P}_2$ is defined by the union of their roles and the union of their setup assumptions.

For example, consider the protocol obtained by parallel composition of two versions of a standard protocol, such as SSL 2.0 and SSL 3.0. By the definition above, a principal running their parallel composition may simultaneously engage in sessions of both protocols. Clearly, a property proved about either protocol individually might no longer hold when the two are run in parallel, since an adversary might use information acquired by executing one protocol to attack the other. Since all the axioms and inference rules in Section 5 except the honesty rule are protocol-independent, the only formulas used in the proof of one protocol that might no longer be sound for a composition involving that protocol are formulas proved using the honesty rule, i.e., the protocol invariants. In order to guarantee that the security properties of the individual protocols are preserved under parallel composition, it is therefore sufficient to verify that each protocol respects the invariants of the other. This observation suggests the following four-step methodology for proving properties of the parallel composition of two protocols.

1. Prove separately the security properties of protocols $\mathcal{P}_1$ and $\mathcal{P}_2$.

$$\vdash_{\mathcal{P}_1} \Psi_1 \ \ \textit{and} \ \ \vdash_{\mathcal{P}_2} \Psi_2$$

2. Identify the set of invariants used in the two proofs, Γ_1 and Γ_2. The formulas included in these sets will typically be the formulas in the two proofs which were proved using the honesty rule. The proofs from the previous step can be decomposed into two parts – the first part proves the protocol invariants using the honesty rule for the protocol, while the second proves the protocol properties using the invariants as hypotheses, but without using the honesty rule. Formally:

$$\vdash_{\mathcal{P}_1} \Gamma_1 \ \textit{and} \ \Gamma_1 \vdash \Psi_1 \ \ \textit{and} \ \ \vdash_{\mathcal{P}_2} \Gamma_2 \ \textit{and} \ \Gamma_2 \vdash \Psi_2$$

3. Notice that it is possible to weaken the hypotheses to $\Gamma_1 \cup \Gamma_2$. The proof of the protocol properties is clearly preserved under a larger set of assumptions.

$$\Gamma_1 \cup \Gamma_2 \vdash \Psi_1 \ \ \textit{and} \ \ \Gamma_1 \cup \Gamma_2 \vdash \Psi_2$$

4. Prove that the invariants, $\Gamma_1 \cup \Gamma_2$, hold for both protocols. This step uses the transitivity of entailment in the logic: if $\vdash_{\mathcal{P}} \Gamma$ and $\Gamma \vdash \gamma$, then $\vdash_{\mathcal{P}} \gamma$. Since $\vdash_{\mathcal{P}_1} \Gamma_1$ and $\vdash_{\mathcal{P}_2} \Gamma_2$ were already proved in step 1, in this step it is sufficient to show that $\vdash_{\mathcal{P}_1} \Gamma_2$ and $\vdash_{\mathcal{P}_2} \Gamma_1$. By Lemma 7.2 below, we therefore have $\vdash_{\mathcal{P}_1|\mathcal{P}_2} \Gamma_1 \cup \Gamma_2$. From this and the formulas from step 3, we can conclude that the security properties of $\mathcal{P}_1$ and $\mathcal{P}_2$ are preserved under their parallel composition.

$$\vdash_{\mathcal{P}_1|\mathcal{P}_2} \Psi_1 \ \ \textit{and} \ \ \vdash_{\mathcal{P}_1|\mathcal{P}_2} \Psi_2$$

Lemma 7.2 *If* $\vdash_{\mathcal{P}_1} \Psi$ *and* $\vdash_{\mathcal{P}_2} \Psi$*, then* $\vdash_{\mathcal{P}_1|\mathcal{P}_2} \Psi$*, where the last step in the proof of* Ψ *in both* $\mathcal{P}_1$ *and* $\mathcal{P}_2$ *uses the honesty rule and no previous step uses the honesty rule.*

Proof: Following the consequent of the honesty rule, Ψ must be of the form $\mathsf{Honest}(\mathtt{T}.\mathit{pname}) \Rightarrow \varphi$ for some formula φ. Suppose that Ψ can be proved for both $\mathcal{P}_1$ and $\mathcal{P}_2$ using the honesty rule. By the definition of the honesty rule, for $i = 1, 2$ we have $\vdash_{\mathcal{P}_i} \mathsf{Start}(\mathtt{T})\,[\,]_{\mathtt{T}}\,\varphi$, and for all $\mathtt{p} \in \mathcal{P}_i$ and for all $\mathtt{basicseq} \in (\mathtt{p}(\mathtt{T}.\mathit{rpars})).\mathit{bseqs}$ we have $\vdash_{\mathcal{P}_i} \varphi\,[\mathtt{basicseq}]_{\mathtt{T}}\,\varphi$. Every basic sequence $\mathtt{basicseq}$ of a role in $\mathcal{P}_1 \mid \mathcal{P}_2$ is a basic sequence of a role in $\mathcal{P}_i$ for $i = 1, 2$. It follows that $\vdash_{\mathcal{P}_1|\mathcal{P}_2} \varphi\,[\mathtt{basicseq}]_{\mathtt{T}}\,\varphi$ and, therefore, by the application of the honesty rule, $\vdash_{\mathcal{P}_1|\mathcal{P}_2} \mathsf{Honest}(\mathtt{T}.\mathit{pname}) \Rightarrow \varphi$. □

Theorem 7.3 *If* $\vdash_{\mathcal{P}_1} \Gamma$ *and* $\vdash_{\mathcal{P}_2} \Gamma$ *and* $\Gamma \vdash \Psi$*, then* $\vdash_{\mathcal{P}_1|\mathcal{P}_2} \Psi$.

8. Related Work

A variety of methods and tools have been developed for analyzing the security guarantees provided by network protocols. The main lines of work include specialized logics [4,44, 20], process calculi [1,2,25,35] and tools [28,43], as well as theorem-proving [34,33] and model-checking methods [26,31,36,41] using general purpose tools. (The cited papers are representative but not exhaustive; see [29] for a more comprehensive survey.)

There are several differences among these approaches. While most model-checking tools can only analyze a finite number of concurrent sessions of a protocol, some of the logics, process calculi, and theorem-proving techniques yield protocol security proofs without bounding the number of sessions. With the exception of the BAN family of logics [4], most approaches involve explicit reasoning about possible attacker actions. Finally, while security properties are interpreted over individual traces in the majority of these methods, in the process calculi-based techniques, security is defined by an equivalence relation between a real protocol and an ideal protocol, which is secure by construction. Inspite of these differences, all of these approaches use the same symbolic model of protocol execution and attack. This model seems to have developed from positions taken by Needham-Schroeder [32], Dolev-Yao [14], and subsequent work by others.

PCL shares several features with BAN [4], a specialized protocol logic. It is designed to be a logic for authentication, with relevant secrecy concepts. Both logics annotate programs with assertions and use formulas for concepts like “freshness”, “sees”, “said”, and “shared secret”. Furthermore, neither logic requires explicit reasoning about the actions of an attacker.

On the other hand, PCL differs from BAN on some aspects since it addresses known problems with BAN. BAN had an abstraction step in going from the program for the protocol to its representation as a logical formula. PCL avoids the abstraction phase since formulas contain the program for the protocol. PCL uses a dynamic logic set-up: after a sequence of actions is executed, some property holds in the resulting state. It is formulated using standard logical concepts: predicate logic and modal operators, with more or less standard semantics for many predicates and modalities. Temporal operators can be used to refer specifically to actions that have happened and the order in which they occurred. Formulas are interpreted over traces and the proof system is sound with respect to the standard symbolic model of protocol execution and attack. On the other hand, BAN was initially presented without semantics. Although subsequently, model-theoretic semantics was defined, the interpretation and use of concepts like “believes” and “jurisdiction” remained unclear. Finally, PCL formulas refer to specific states in the protocol. For example, something may be fresh at one step, but no longer fresh later. In contrast, BAN statements are persistent, making it less expressive.

PCL also shares several common points with the Inductive Method [34]. Both methods use the same trace-based model of protocol execution and attack; proofs use induction and provable protocol properties hold for an unbounded number of sessions. One difference is the level of abstraction. Paulson reasons explicitly about traces including possible intruder actions whereas basic reasoning principles are codified in PCL as axioms and proof rules. Proofs in PCL are significantly shorter and do not require any explicit reasoning about an intruder. Finally, while Paulson’s proofs are mechanized using Isabelle, most proofs in PCL are currently hand-proofs, although the steps needed to automate PCL in a meta-logic are described in this chapter.

9. Conclusions

Protocol Composition Logic (PCL), summarized and illustrated by example in this chapter, is a formal logic for stating and proving security properties of network protocols. PCL uses “direct reasoning”, in the sense that our proof steps reason directly about the effects of successive protocol steps. PCL proof steps also reason only about the actions of explicit threads and honest protocol participants. Because of the way that PCL axioms and rules are proved sound for reasoning about a semantic model that includes an attacker, there is no need to consider the attacker explicitly in the proof system.

A distinctive goal of PCL is to support compositional reasoning about security protocols. For example, PCL allows proofs of key-distribution protocols to be combined with proofs for protocols that use these keys. While the logic was originally developed for the symbolic “Dolev-Yao” model of protocol execution and attack used in this chapter, a variant of the logic with similar reasoning principles has also been developed for the computational model used by cryptographers.

Based on approximately ten years of experience with successive versions of the logic, PCL appears to scale well to industrial protocols of five to twenty messages, in part because PCL proofs appear to be relatively short (for formal proofs). The logic has been successfully applied to a number of industry standards including SSL/TLS, IEEE 802.11i (marketed as WPA2), and Kerberos V5.

While PCL could be considered a *semi-formalized* logic for proving properties of protocols, because we do not give formal proof rules for entailment between preconditions or postconditions, PCL provides an approach for fully formal proofs of security properties of protocols using any logic framework supporting the formal syntax and symbolic semantics presented in this chapter.

Acknowledgements We thank Ante Derek, Nancy Durgin, and Dusko Pavlovic for contributing towards the formulation of PCL, and Michael Backes, Changhua He, Jean-Pierre Seifert, Mukund Sundararajan and Mathieu Turuani for collaborating on case studies that improved the logic. Computational PCL began as joint work with Ante Derek, Vitaly Shmatikov, Mathieu Turuani, and Bogdan Warinschi.

References

[1] M. Abadi and A. Gordon. A calculus for cryptographic protocols: the spi calculus. *Information and Computation*, 148(1):1–70, 1999. Expanded version available as SRC Research Report 149 (January 1998).

[2] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *28th ACM Symposium on Principles of Programming Languages*, pages 104–115, 2001.

[3] M. Bellare and P. Rogaway. Entity authentication and key distribution. In *Advances in Cryrptology - Crypto '93 Proceedings*, pages 232–249. Springer-Verlag, 1994.

[4] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, 1990.

[5] Anupam Datta, Ante Derek, John C. Mitchell, and Dusko Pavlovic. A derivation system for security protocols and its logical formalization. In *Proceedings of 16th IEEE Computer Security Foundations Workshop*, pages 109–125. IEEE, 2003.

[6] Anupam Datta, Ante Derek, John C. Mitchell, and Dusko Pavlovic. Secure protocol composition (extended abstract). In *Proceedings of ACM Workshop on Formal Methods in Security Engineering*, pages 11–23, 2003.

[7] Anupam Datta, Ante Derek, John C. Mitchell, and Dusko Pavlovic. Abstraction and refinement in protocol derivation. In *Proceedings of 17th IEEE Computer Security Foundations Workshop*, pages 30–45. IEEE, 2004.

[8] Anupam Datta, Ante Derek, John C. Mitchell, and Dusko Pavlovic. Secure protocol composition. In *Proceedings of 19th Annual Conference on Mathematical Foundations of Programming Semantics*. ENTCS, 2004.

[9] Anupam Datta, Ante Derek, John C. Mitchell, and Dusko Pavlovic. A derivation system and compositional logic for security protocols. *Journal of Computer Security*, 13(3):423–482, 2005.

[10] Anupam Datta, Ante Derek, John C. Mitchell, and Arnab Roy. Protocol Composition Logic (PCL). In *Computation, Meaning end Logic: Articles dedicated to Gordon Plotkin*, volume 172, pages 311–358. Electronic Notes in Theoretical Computer Science, 2007.

[11] Anupam Datta, Ante Derek, John C. Mitchell, Vitaly Shmatikov, and Mathieu Turuani. Probabilistic polynomial-time semantics for a protocol security logic. In *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP '05)*, Lecture Notes in Computer Science, pages 16–29. Springer-Verlag, 2005.

[12] Anupam Datta, Ante Derek, John C. Mitchell, and Bogdan Warinschi. Computationally sound compositional logic for key exchange protocols. In *Proceedings of 19th IEEE Computer Security Foundations Workshop*, pages 321–334. IEEE, 2006.

[13] W. Diffie, P. C. Van Oorschot, and M. J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2:107–125, 1992.

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

[15] N. A. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov. Undecidability of bounded security protocols. In *Workshop on Formal Methods and Security Protocols*, 1999.

[16] Nancy Durgin, Patrick Lincoln, John Mitchell, and Andre Scedrov. Multiset rewriting and the complexity of bounded security protocols. *J. Comput. Secur.*, 12(2):247–311, 2004.

[17] Nancy Durgin, John C. Mitchell, and Dusko Pavlovic. A compositional logic for protocol correctness. In *Proceedings of 14th IEEE Computer Security Foundations Workshop*, pages 241–255. IEEE, 2001.

[18] Nancy Durgin, John C. Mitchell, and Dusko Pavlovic. A compositional logic for proving security properties of protocols. *Journal of Computer Security*, 11:677–721, 2003.

[19] Joseph A. Goguen and José Meseguer. Order-sorted algebra i: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theor. Comput. Sci.*, 105(2):217–273, 1992.

[20] Li Gong, Roger Needham, and Raphael Yahalom. Reasoning About Belief in Cryptographic Protocols. In Deborah Cooper and Teresa Lunt, editors, *Proceedings 1990 IEEE Symposium on Research in Security and Privacy*, pages 234–248. IEEE Computer Society, 1990.

[21] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. Foundations of Computing. MIT Press, 2000.

[22] Robert Harper, Furio Honsell, and Gordon D. Plotkin. A framework for defining logics. In *LICS*, pages 194–204. IEEE Computer Society, 1987.

[23] Changhua He, Mukund Sundararajan, Anupam Datta, Ante Derek, and John C. Mitchell. A modular correctness proof of IEEE 802.11i and TLS. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 2–15, 2005.

[24] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[25] Patrick D. Lincoln, John C. Mitchell, Mark Mitchell, and Andre Scedrov. Probabilistic polynomial-time equivalence and security protocols. In *Formal Methods World Congress, vol. I*, number 1708 in Lecture Notes in Computer Science, pages 776–793. Springer-Verlag, 1999.

[26] G. Lowe. Some new attacks upon security protocols. In *Proceedings of 9th IEEE Computer Security Foundations Workshop*, pages 162–169. IEEE, 1996.

[27] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.

[28] C. Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

[29] C. Meadows. Open issues in formal methods for cryptographic protocol analysis. In *Proceedings of DISCEX 2000*, pages 237–250. IEEE, 2000.

[30] J. C. Mitchell, V. Shmatikov, and U. Stern. Finite-state analysis of ssl 3.0. In *Proceedings of the Seventh USENIX Security Symposium*, pages 201–216, 1998.

[31] J.C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Murφ. In *Proc. IEEE Symp. Security and Privacy*, pages 141–151, 1997.

[32] R.M. Needham and M.D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[33] L.C. Paulson. Mechanized proofs for a recursive authentication protocol. In *Proceedings of 10th IEEE Computer Security Foundations Workshop*, pages 84–95, 1997.

[34] L.C. Paulson. Proving properties of security protocols by induction. In *Proceedings of 10th IEEE Computer Security Foundations Workshop*, pages 70–83, 1997.

[35] Ajith Ramanathan, John C. Mitchell, Andre Scedrov, and Vanessa Teague. Probabilistic bisimulation and equivalence for security analysis of network protocols. In *Foundations of Software Science and Computation Structures, 7th International Conference, FOSSACS 2004, Proceedings*, volume 2987 of *Lecture Notes in Computer Science*, pages 468–483. Springer-Verlag, 2004.

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

[37] Arnab Roy, Anupam Datta, Ante Derek, and John C. Mitchell. Inductive proofs of computational secrecy. In Joachim Biskup and Javier Lopez, editors, *ESORICS*, volume 4734 of *Lecture Notes in Computer Science*, pages 219–234. Springer, 2007.

[38] Arnab Roy, Anupam Datta, Ante Derek, John C. Mitchell, and Jean-Pierre Seifert. Secrecy analysis in Protocol Composition Logic. In *Proceedings of 11th Annual Asian Computing Science Conference*, 2006.

[39] Arnab Roy, Anupam Datta, Ante Derek, John C. Mitchell, and Jean-Pierre Seifert. Secrecy analysis in protocol composition logic. In *Formal Logical Methods for System Security and Correctness, IOS Press,*

2008. Volume based on presentations at Summer School 2007, Formal Logical Methods for System Security and Correctness, Marktoberdorf, Germany, 2008.

[40] Arnab Roy, Anupam Datta, and John C. Mitchell. Formal proofs of cryptographic security of Diffie-Hellman-based protocols. In *Trustworthy Global Computing (TGC)*, pages 312–329. Springer, 2007.

[41] S. Schneider. Security properties and CSP. In *IEEE Symp. Security and Privacy*, 1996.

[42] Steve Schneider. Verifying authentication protocols with csp. *IEEE Transactions on Software Engineering*, pages 741–58, 1998.

[43] D. Song. Athena: a new efficient automatic checker for security protocol analysis. In *Proceedings of 12th IEEE Computer Security Foundations Workshop*, pages 192–202. IEEE, 1999.

[44] P. Syverson and P.C. van Oorschot. On unifying some cryptographic protocol logics. In *Proceedings of 7th IEEE Computer Security Foundations Workshop*, pages 14–29, 1994.

[45] T. Y. C. Woo and S. C. Lam. A semantic model for authentication protocols. In *Proceedings IEEE Symposium on Research in Security and Privacy*, 1993.