A Proof-Based Approach to Formalizing Protocols in Linear Epistemic Logic

Elizabeth McBryde Davis

May 6, 2014

School of Computer Science

Carnegie Mellon University

Pittsburgh, PA 15213

Research Advisors:

Frank Pfenning, Advisor

Chris Martens, Graduate Student Advisor

Submitted in partial fulfillment of the requirements

for the Senior Research Honors Thesis.

Copyright © 2014 Elizabeth McBryde Davis

Partially supported by the Qatar National Research Fund under grant NPRP 09-1107-1-168

Keywords: Formalisms, Public Key Cryptography, Needham-Schröder, Needham-Schröder-Lowe, Linear Logic, Epistemic Logic, Linear Epistemic Logic

Abstract

Linear epistemic logic can be used to reason about changing knowledge states of agents acting in a system. Here we use it to formalize the Needham-Schröder-Lowe public-key authentication protocol for establishing secure communication sessions. We have developed a notion of *adequacy* to refer to the formal compositional correspondence between the protocol and the formalism. Through the iterative process of attempting to prove adequacy theorems and noting where and how the proof breaks down, we have been able to refine the formalism so that it adheres to the structure and semantics of the protocol as it was originally specified. This work is the first step towards showing that rigorous formal reasoning can be applied to protocols and processes followed in the wild.

Contents

List of Figures

Chapter 1

Introduction

The goal of this thesis is to give a formalism for the Needham-Schröder-Lowe public-key authentication protocol[5] using DeYoung and Pfenning's linear epistemic logic[1]. We demonstrate the formalism's adequacy, or formal compositional correspondence to the protocol as originally given, by proving both soundness and completeness with respect to the possible state transitions. Ultimately, our hope is that this work will show the potential to use linear epistemic logic as a tool to formalize and reason about protocols.

1.1 Expanding on Prior Work

Protocols are generally specified as a set of rules that participants agree on between one another to follow. Consider the example in Figure 1.1. It depicts A sending B a message containing M in step 1, and B responding with a message containing $f(M)$ in step 2.

$$\begin{array}{ll} 1. & A \to B : \quad M \\ 2. & B \to A : \quad f(M) \end{array}$$

Figure 1.1: A simple protocol for two agents.

It has been noted in [3] that such a presentation relies heavily on honest participants and gives no explicit indication as to what information a participant has at various steps of the protocol.

For instance, it is not clear whether or not B knew of M before receiving it from A, as well as whether or not B waits to receive M before sending $f(M)$. Thus we are forced to rely on accompanying prose to tell us how the protocol is supposed to work.

Dolev and Yao addressed this problem in [2], noting the difficulty of reasoning about a protocol's security in informal language. They provide a formalism for public key encryption, as well as algorithms to determine just how secure a given protocol is against imposter and replay attacks. The formalism provides encryption and decryption functions as primitives, abstracting away from the implementation so that the protocol can be handled symbolically.

Durgin et al. expand on the Dolev-Yao model with multiset rewriting (MSR), in [3], formalizing it as as state transitions under the application of rules, where states are represented as multisets of first-order atomic formulas, and rules are transitions between multisets of atomic formulas. Their approach involves using the existential quantifier to introduce unique terms such as nonces or keys, which many protocols rely on to guard against replay attacks or intruders "guessing" the values. Their running example is of the Needham-Schröder public key authentication protocol [7], the predecessor to the Needham-Schröder-Lowe protocol and nearly identical.

Our formalism is similar to that of Durgin et al. in that we take advantage of the existential quantifier to generate fresh terms. Barring a translation from MSR to Horn fragments in linear logic, however, our approach differs from that of Durgin et al. in three important ways. First, we provide a formalism not only for the communication between the agents seeking to establish a secure session, but also for the interaction between principals and a key server as they request and learn the public key of their session partners. Second, in using linear epistemic logic, we make use of the modalities to force a distinction between persistent knowledge, linear knowledge, and communication. In making this distinction, our logic introduces reasoning about local state, in which we explicitly consider propositions with respect to the principals they are associated with. Third, we make the actions between protocol steps, such as encryption, transmission, receipt, and decryption of messages explicit. This finer level of detail allows us to formally examine how principals obtain and use their knowledge of the protocol state when carrying out an instance.

1.2 Approach and Definitions

1.2.1 Linear Logic

We use logic to model domains and derive new conclusions from facts we already know within those domains. Most logics present truth as persistent. For example, suppose you know A and $A \supset B$[1] to be true, and modus ponens[2] is a valid inference rule. Then upon applying the rule to these propositions, you know A, $A \supset B$, and now B. This is a perfectly acceptable notion of truth if the propositions you're reasoning about stay true. For example, letting $A =$ *XYZW is a square* and $B =$ *XYZW is a rhombus*, it is entirely acceptable for all three propositions to be true at once.

Logics which treat truth as persistent by default do not allow us to easily reason about propositions which might be true at one point but false at another. For example, suppose we are trying to model the transaction involved in purchasing a beverage. Consider the previous example, where we know A and $A \supset B$, this time letting $A =$ *You have $1* and $B =$ *You purchase a soda*. Applying modus ponens allows us to conclude that we have purchased a soda, but we aren't able to capture the fact that we no longer have the $1 that we used to purchase the soda. Additionally, we aren't able to express the fact that, if we have $2, we can purchase two sodas.

Linear logic, first invented by Girard[4], solves this problem. It is a logic which treats truth as ephemeral and propositions as consumable resources, lending itself nicely to the paradigm of reasoning about changing truth values. When we say that a proposition A is consumable, we mean that if we use A in a derivation $\mathcal{D}$, we can only use it once, and we are not permitted to use it in a separate derivation $\mathcal{D}'$. So taking our example from before about purchasing a soda, we can construct a derivation which accurately models the one-time exchange of $1 for a soda. We replace $A \supset B$ with $A \multimap B$, where $\multimap$ (pronounced "lolly") is the symbol for linear implication, which behaves in the same modus-ponens-like way, except the implication is consumed. Now

[1] $A \supset B$ represents logical implication, interpreted as *if A is true then B is true*.
[2] Modus ponens says that if you know A is true and you know $A \supset B$ is true, then you can deduce that B is true.

when we apply modus ponens to A and $A \multimap B$, we are left only with B, meaning that we no longer have the \$1 or any knowledge of the fact that \$1 will get you a soda.

Suppose we extend our example and you have, in fact, two dollars. We would represent this in linear logic as $A \otimes A$, where $\otimes$ (pronounced "tensor") represents multiplicative linear conjunction. Having $A \otimes A$ means that you have two copies of A in the same state. So when we apply modus ponens to $A \multimap B$ and one of our copies of A, we have $A \otimes B$. Suppose we wanted to buy two sodas with our \$2. This is no longer possible, since we consumed our instance of $A \multimap B$. We could fix this by making some number n copies of $A \multimap B$ and just consuming one copy of $A \multimap B$ for every \$1 we spend on soda. But linear logic permits a more elegant solution.

In addition to the linear one-time-use mode of handling propositions, linear logic also supports the notion of persistent truth. If you have a proposition C that is persistent, as opposed to the linear propositions we have been considering up to now, then you can make use of C arbitrarily many times in the construction of a proof. If the soda vendor in our example has an infinite stock of soda, then we could make $A \multimap B$ into a persistent proposition. We refer to persistent propositions in linear logic by marking them with a ! (pronounced "bang"). So, if we have A and $!(A \multimap B)$, then we can derive B and still have knowledge of $A \multimap B$. Further, if we have $A \otimes A$ and $!(A \multimap B)$, then we can derive $B \otimes B$ and, similarly, still be aware of $A \multimap B$.

Now that we have an intuition for the kind of expressiveness that linear logic gives us, we provide a formalism with which to construct derivations. We use the multiplicative and exponential fragment of linear logic, sometimes referred to as MELL. The formalism we use is a sequent calculus, with left and right rules to specify how we treat connectives in our proof search.

To say that a set of persistent formulas Γ and linear formulas Δ *prove* a proposition $\mathcal{C}$ means that every formula in Δ is used exactly once in the derivation, with no extra constraints on how the formulas in Γ are used. We express it with the notation below in 1.1.

$$\Gamma; \Delta \vdash \mathcal{C} \tag{1.1}$$

For our example where we have \$1, we can construct a derivation of our single soda purchase by letting $\Gamma = \cdot$, $\Delta = A \otimes\, !(A \multimap B)$, and $\mathcal{C} = B$. Of course, we also want to show the derivation that led us to this conclusion. For this purpose, we give left and right rules of interest for a sequent calculus as defined in [8] in Figure 1.2. There are more connectives in linear logic, but they are not used in this thesis, so we omit their rules for brevity. Note that while we have not talked about the use of existential quantification, as handled in the last two rules in Figure 1.2, it is used in the formalism that we present in later sections. Essentially, when propositions require terms as arguments, the first-order existential quantifier over terms of type τ allows us to generate fresh terms which are distinct from all other terms in the proof.

$$\frac{}{\Gamma; A \vdash A}\ \mathsf{init} \qquad \frac{\Gamma, A; \Delta, A \vdash C}{\Gamma, A; \Delta \vdash C}\ \mathsf{copy}$$

$$\frac{\Gamma; \Delta, A \vdash B}{\Gamma; \Delta \vdash A \multimap B}\ \multimap_R \qquad \frac{\Gamma; \Delta_1 \vdash A \quad \Delta_2, B \vdash C}{\Gamma; \Delta_1, \Delta_2, A \multimap B \vdash C}\ \multimap_L$$

$$\frac{\Gamma; \Delta_1 \vdash A \quad \Gamma; \Delta_2 \vdash B}{\Gamma; \Delta_1, \Delta_2 \vdash A \otimes B}\ \otimes_R \qquad \frac{\Gamma; \Delta, A, B \vdash C}{\Gamma; \Delta, A \otimes B \vdash C}\ \otimes_L$$

$$\frac{\Gamma; \cdot \vdash A}{\Gamma; \cdot \vdash\, !A}\ !_R \qquad \frac{\Gamma, A; \Delta \vdash C}{\Gamma; \Delta, !A \vdash C}\ !_L$$

$$\frac{\Gamma; \Delta \vdash [t/x]A}{\Gamma; \Delta \vdash \exists x.A}\ \exists R \qquad \frac{\Gamma; \Delta, [a/x]A \vdash C}{\Gamma; \Delta, \exists x.A \vdash C}\ \exists L^a$$

Figure 1.2: Left and right rules for connectives in linear logic.

An example proof of $A \multimap B; A \vdash B$ is given in Figure 1.3. Note that the derivation is constructed from the bottom up, building what we call a proof tree. We can say that we have a complete proof when all branches are closed off on top, as with the init rule.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \multimap B; A \vdash A}\ \mathsf{init} \quad \dfrac{}{A \multimap B; B \vdash B}\ \mathsf{init}}{A \multimap B; A, A \multimap B \vdash B}\ \multimap_L}{A \multimap B; A \vdash B}\ \mathsf{copy}}{\cdot; A,\, !(A \multimap B) \vdash B}\ !_L}{\cdot; A \otimes\, !(A \multimap B) \vdash B}\ \otimes_L$$

Figure 1.3: A proof in linear logic.

Having provided an introduction to linear logic, we present the actual logic of interest for our formalism, DeYoung and Pfenning's focused linear epistemic logic.

1.2.2 DeYoung and Pfenning's Focused Linear Epistemic Logic

An extension of Girard's linear logic, DeYoung and Pfenning's logic adds two new important features. First of all, it makes use of a methodology known as focusing, a proof search strategy developed by Andreoli in [6], which limits the number of nondeterministic choices made in a derivation. This logic is weakly focused, meaning that certain eager rule applications are not mandatory. Secondly, it adds the notion of acting principals' knowledge and their ability to affirm statements that are true.

In a focused sequent calculus, propositions A are polarized. Positive propositions have invertible left rules and non-invertible right rules, and negative propositions have invertible right rules and non-invertible left rules. Their syntax is as follows:

$$A^+ ::= p^+ \mid A^+ \otimes B^+ \mid \mathbf{1} \mid \exists x : \tau . A^+ \mid\ !A^- \mid [K]A \mid [\![K]\!]A \mid A^-$$
$$A^- ::= p^- \mid A^+ \multimap B^- \mid \forall x : \tau . A^- \mid \langle K \rangle A \mid \{A^+\}$$

Note the operators carried over from standard linear logic as presented in Figure 1.2. We also introduce some new operators. Where $\otimes$ is multiplicative conjunction, $\mathbf{1}$ is its unit, representing the empty conjunction. $\forall x : \tau . A^-$ is first-order universal quantification over terms of type τ. $\{A\}$ is a monad, marking the main branch of a derivation.

We introduce three new modalities of interest in this logic: affirmation, knowledge, and possession. They are indexed to principals and are used to distinguish the different ways that they can interact with propositions.

Affirmation, represented propositionally as $\langle K \rangle A$ is read as "a principal K says A." We use it to model message passing between principals.

Knowledge, represented as $[\![K]\!]A$ is read as "a principal K knows A." We use it to model knowledge that principals have about other principals.

Possession, represented propositionally as $[K]A$ is read as "a principal K has A." We use it to refer to expendable resources which principals consume in a derivation. Essentially, $[K]A$ is a linear form of $[\![K]\!]A$.

Before presenting the sequent calculus, we discuss the judgments that it comprises. We have a set of judgments for our persistent context Γ and our linear context Δ, as well as our consequents:

$$\text{Unrestricted Assumptions } \Gamma ::= \cdot \mid \Gamma, A^- \,\mathsf{valid} \mid \Gamma, K \,\mathsf{knows}\, A^-$$

$$\text{Linear Assumptions } \Delta ::= \cdot \mid \Delta, A^+ \,\mathsf{hyp} \mid \Delta, K \,\mathsf{has}\, A^-$$

$$\text{Consequents } J ::= A^- \,\mathsf{true} \mid K \,\mathsf{says}\, A^+ \mid A^+ \mathsf{lax}$$

Given this introduction to the logic, we present the inference rules for the weakly focused sequent calculus.

Initial Sequents

$$\frac{}{\Gamma; p^+ \vdash [p^+]}\ \mathtt{atom}^+ \qquad \frac{}{\Gamma; [p^+] \vdash p^-}\ \mathtt{atom}^-$$

Positive Connectives

$$\frac{\Gamma; \Delta_1 \vdash [A^+] \quad \Gamma; \Delta_2 \vdash [B^+]}{\Gamma; \Delta_1, \Delta_2 \vdash [A^+ \otimes B^+]}\ \otimes_R \qquad \frac{\Gamma; \Delta, A^+, B^+ \vdash J}{\Gamma; \Delta, A^+ \otimes B^+ \vdash J}\ \otimes_L$$

$$\frac{}{\Gamma; \cdot \vdash [\mathbf{1}]}\ \mathbf{1_R} \qquad \frac{\Gamma; \Delta \vdash J}{\Gamma; \Delta, \mathbf{1} \vdash J}\ \mathbf{1_L}$$

$$\frac{\Gamma; \Delta \vdash [[t/x]A^+]}{\Gamma; \Delta \vdash [\exists x : \tau . A^+]}\ \exists_R \ \exists_L^a$$

$$\frac{\Gamma, A^-; \Delta, [A^-] \vdash J}{\Gamma, A^-; \Delta \vdash J}\ \mathsf{copy} \qquad \frac{\Gamma; \cdot \vdash A^-}{\Gamma; \cdot \vdash [!A^-]}\ !_R \qquad \frac{\Gamma, A^-; \Delta \vdash J}{\Gamma; \Delta, !A^- \vdash J}\ !_L$$

$$\frac{\Gamma; \Delta, [A^-] \vdash J}{\Gamma; \Delta, K\ \mathsf{has}\ A^- \vdash J}\ \mathsf{has}_L \qquad \frac{\Gamma|_K; \Delta|_K \vdash A^-}{\Gamma; \Delta|_K \vdash [[K]A^-]}\ []_R \qquad \frac{\Gamma; \Delta, K\ \mathsf{has}\ A^- \vdash J}{\Gamma; \Delta, [K]A^- \vdash J}\ []_L$$

$$\frac{\Gamma; K\ \mathsf{knows}\ A^-; \Delta, [A^-] \vdash J}{\Gamma; K\ \mathsf{knows}\ A^-; \Delta \vdash J}\ \mathsf{knows}_L \qquad \frac{\Gamma|_K; \cdot \vdash A^-}{\Gamma; \cdot \vdash [[\![K]\!]A^-]}\ [\![]\!]_R$$

$$\frac{\Gamma, K\ \mathsf{knows}\ A^-; \Delta \vdash J}{\Gamma; \Delta, [\![K]\!]A^- \vdash J}\ [\![]\!]_L$$

$$\frac{\Gamma; \Delta \vdash A^-}{\Gamma; \Delta \vdash [A^-]}\ \mathtt{blur} \qquad \frac{\Gamma; \Delta, [A^-] \vdash J}{\Gamma; \Delta, A^- \vdash J}\ \mathtt{lfoc}$$

Negative Connectives

$$\frac{\Gamma; \Delta, A^+ \vdash B^-}{\Gamma; \Delta \vdash A^+ \multimap B^-}\ \multimap_R \qquad \frac{\Gamma, \Delta_1 \vdash [A^+] \quad \Gamma, \Delta_2, [B^-] \vdash J}{\Gamma; \Delta_1, \Delta_2, [A^+ \multimap B^-] \vdash J}\ \multimap_L$$

$$\frac{\Gamma; \Delta \vdash [a/x]A^-}{\Gamma; \Delta \vdash \forall x : \tau . A^-}\ \forall^a_R \qquad \frac{\Gamma; \Delta, [[t/x]A^-] \vdash J}{\Gamma; \Delta, [\forall x : \tau . A^-] \vdash J}\ \forall_L$$

$$\frac{\Gamma; \Delta \vdash [A^+]}{\Gamma; \Delta \vdash K\ \mathsf{says}\ A^+}\ \mathsf{says}_R \qquad \frac{\Gamma; \Delta \vdash K\ \mathsf{says}\ A^+}{\Gamma; \Delta \vdash \langle K \rangle A^+}\ \langle\rangle_R$$

$$\frac{\Gamma; \Delta, A^+ \vdash K\ \mathsf{says}\ C^+}{\Gamma; \Delta, [\langle K \rangle A^+] \vdash K\ \mathsf{says}\ C^+}\ \langle\rangle_L$$

$$\frac{\Gamma; \Delta \vdash [A^+]}{\Gamma; \Delta \vdash A^+ \mathsf{lax}}\ \mathsf{lax}_R \qquad \frac{\Gamma; \Delta \vdash A^+ \mathsf{lax}}{\Gamma; \Delta \vdash \{A^+\}}\ \{\}_R \qquad \frac{\Gamma; \Delta, A^+ \vdash C^+ \mathsf{lax}}{\Gamma; \Delta, [\{A^+\}] \vdash C^+ \mathsf{lax}}\ \{\}_L$$

Restriction Operator

$$(\cdot)|_K = \cdot \qquad (\cdot)|_K = \cdot$$

$$(\Delta, A^+\mathsf{hyp})|_K = \Delta|_K \qquad (\Gamma, A^-\mathsf{valid})|_K = \Gamma|_K$$

$$(\Delta, K\ \mathsf{has}\ A^-)|_K = \Delta|_K, K\ \mathsf{has}\ A^- \qquad (\Gamma, K\ \mathsf{knows}\ A^-)|_K = \Gamma|_K, K\ \mathsf{knows}\ A^-$$

$$(\Delta, L\ \mathsf{has}\ A^-)|_K = \Delta|_K, \text{ if } L \neq K \qquad (\Gamma, L\ \mathsf{knows}\ A^-)|_K = \Gamma|_K, \text{ if } L \neq K$$

Our use of this logic and, more importantly, the proofs we supply, depend on some extra definitions from [1]. We present the following two definitions of state and rewrite step:

Definition 1.1. A pair of contexts $\Gamma; \Delta$ is a state if and only if Δ fits the following grammar:

$$\Delta ::= \cdot \mid \Delta, A^-\ \mathsf{hyp} \mid \Delta, p^+\ \mathsf{hyp} \mid \Delta, K\ \mathsf{has}\ A^-$$

Definition 1.2. The rewrite step $\Gamma; \Delta \longrightarrow \Gamma'; \Delta'$ holds if and only if there exists a derivation of the following form, universally quantified over C:

$$\Gamma'; \Delta' \vdash C^+\ \mathsf{lax}$$

$$\vdots$$

$$\frac{\Gamma; \Delta_2, A_2^+\ \mathsf{hyp} \vdash C^+\ \mathsf{lax}}{\Gamma; \Delta_2, [\{A_2^+\}] \vdash C^+\ \mathsf{lax}}\ \{\}_L$$

$$\vdots$$

$$\frac{\Gamma; \Delta_1, [A_1^-] \vdash C^+\ \mathsf{lax}}{\Gamma; \Delta \vdash C^+\ \mathsf{lax}}\ *$$

In addition, we make use of a tool which we call state transition sequences, which we notate $\Gamma; \Delta \xrightarrow{\epsilon} \Gamma'; \Delta'$. Define $\longrightarrow^*$ to be the reflexive, transitive closure of $\longrightarrow$. Then we define $\xrightarrow{\epsilon}$ to be a labeled subset of $\longrightarrow^*$, allowing us to refer to chains of rewrite derivations $\Gamma; \Delta \longrightarrow^* \Gamma'; \Delta'$

that follow the structure of the sequence designated by ϵ.

1.2.3 Modes

Our formalism makes use of some concepts borrowed from logic programming. Propositions in logic programming are units of computation, expressed as relations between logical terms. The computation comes from the interpreter trying to determine whether a given relation holds, what conditions it holds under, etc. by constructing derivations.

Relations can be specified in such a way that the interpreter computes arguments that make a relation sound. Which arguments are to be pre-specified versus computed is determined by the relation's mode. Assuming that a relation's arguments are represented as an n-tuple of logic variables, its mode is given as an n-tuple of mode symbols s, where s can be either $+$, meaning it must be pre-specified, $-$, meaning it must be computed, or $?$, meaning that it can have either function.

A relation can have multiple modes. Let P be a relation between n arguments and suppose it has m modes. Then there would be m n-tuples of mode symbols s to specify how the relation could be used.

For example, suppose `Plus(`N`,` M`,` S`)` is a relation between integers and that is true if and only if $N + M = S$. Suppose also that `Plus` has the modes $+, +, -$ and $+, -, +$. Then the relation could be used in two ways. With the first mode, the addends would be supplied, and their sum would be calculated. With the second mode, the first addend and the sum would be given, and the program would calculate the second addend.

1.2.4 Adequacy

Our goal is to show that our formalism fully captures the epistemic semantics of the protocol as specified by the creators. From examining the original protocol we have determined what knowledge each principal should have at each step and how their knowledge states change as the

protocol progresses. We claim that linear epistemic logic can make these changing knowledge states explicit.

To prove our formalism's adequacy, we show that it is sound and complete with respect to the protocol as it is originally specified. First we specify an initial state which characterizes the knowledge that each principal should have before an instance of the protocol is initiated. Then we identify state transition sequences within our formalism which make the previously identified knowledge states and changes explicit.

To prove soundness, first we demonstrate that a protocol progression trace can be generated according to the specified state transition sequences. To prove completeness, we show that all traces which generate protocol completion predicates contain the specified state transition sequences in order.

Chapter 2

Formalizing the Needham-Schröder-Lowe Public-Key Authentication Protocol

2.1 The Protocol

The Needham-Schröder-Lowe public key authentication protocol is executed as a series of message exchanges between two agents, which we refer to as the approacher and the provokee, and a key server. The standard notation for the protocol as specified in [5] is given in Figure 2.1. As before, we interpret $1.\ X \to Y :\ \ M$ to mean that in step 1, X sends Y a message containing M. Here in Figure 2.1, A refers to the approacher, P the provokee, S the server. The symbols K_A^+ and K_A^- refer to A's public and private keys, respectively, and N_a and N_p are nonces.

1. $A \to S :\ \ A, P$
2. $S \to A :\ \ \{K_P^+, P\}_{K_S^-}$
3. $A \to P :\ \ \{N_a, A\}_{K_P^+}$
4. $P \to S :\ \ P, A$
5. $S \to P :\ \ \{K_A^+, A\}_{K_S^-}$
6. $P \to A :\ \ \{N_a, N_p, P\}_{K_A^+}$
7. $A \to P :\ \ \{N_p\}_{K_P^+}$

Figure 2.1: The Needham-Schröder-Lowe protocol.

The protocol begins in step 1, with the approacher sending the server a message requesting the provokee's public key. Upon receiving the request, in step 2, the server responds with a digitally signed[1] message containing both the provokee's public key and the provokee's identity. Once the approacher has the provokee's public key, the protocol proceeds to step 3, in which the approacher sends a message to the provokee containing a nonce they generated for the session, along with their own identity, encrypted with the provokee's public key. Upon receiving and decrypting the message, in step 4, the provokee requests the approacher's public key from the server. The server responds in step 5 with a digitally signed message containing the approacher's public key and identity. The provokee then responds to the approacher in step 6 with a message containing the approacher's nonce, a nonce that the provokee generated for the session, and the provokee's own identity, all encrypted with the approacher's public key. After receiving the provokee's response, the approacher wraps up the protocol in step 7 with a message to the provokee containing just the provokee's nonce, encrypted with the provokee's public key.

Once all the steps in the protocol have been followed, and all messages received and processed, it is to be understood that the approacher and the provokee have sufficient information to conclude that they have established a secure session[2]. The messages sent between them are openable only by the intended recipient, the nonces are fresh for the session, and the nonces and identities of the intended participants are concealed within the encrypted messages.

In the following section we give a reconstruction of the protocol in DeYoung and Pfenning's linear epistemic logic. We define the terms, propositions, and rules specific to the domain. The logic is given at the same level of abstraction as the protocol according to [5], in order to avoid introducing assumptions about the implementation while remaining cognizant of the details that are specified.

[1] A message encrypted with an agent's private key is said to be digitally signed. Agents can verify a sender's authenticity by decrypting a digitally signed message with the sender's public key, since public keys and private keys are inverses.

[2] According to Lowe in [5], the protocol is safe against attacks that do not exploit properties of the encryption method used

2.2 The Formalism

2.2.1 Terms

Keys $\phi ::= K_A^+ \mid K_A^-$

The public and private keys, respectively, associated with principal A.

Contents $M' ::= A \mid \textbf{getkey}(B, A) \mid \kappa(\phi, A) \mid N_a \mid M' \sim M'$

Message contents can be a reference to a principal (A in this case), key requests (A requesting B's public key), key associations (ϕ is A's public key), nonces, or concatenations of contents.

Cyphertexts H

Cyphertexts are abstract crypto-objects which represent a contents term that has been encrypted with some key.

Payloads $M ::= \textbf{plain}(M') \mid \textbf{encPub}(H, A) \mid \textbf{encPri}(H, A)$

Message payloads can either be plain text, cyphertexts encrypted with public keys (H is a message encrypted with A's public key), or cyphertexts encrypted with private keys (H is a message encrypted with A's private key).

Tags $T ::= \mathcal{T}_i$

Tags are generated to track messages across destructive operations which comprise encryption, decryption, concatenation, and decomposition.

Unpacking Indicators $U ::= plain \mid encPub(A) \mid encPri(A)$

These indicators mark whether a processed message was originally plain text, encrypted with A's public key, or encrypted with A's private key.

2.2.2 Propositions

Our propositions follow the convention in which persistent propositions begin with a capital letter and linear propositions begin with a lowercase letter.

Non-Indexed Relations

Crypt(H, ϕ, M) : An interface to an external cryptographic signature understood by all participants in the protocol. Presented as a relation between ciphertext H, key ϕ, and message M with two modes:

1. When used with mode $-, +, +$, H is the cyphertext generated from encrypting M with ϕ.
2. When used with mode $+, +, -$, M is the message revealed when H is decrypted with ϕ.

AgentNeq(A, B) : A and B are distinct principals.

Principal Information

Server(A) : A is a server.

PublicKey(ϕ, A) : ϕ is A's public key.

PrivateKey(ϕ, A) : ϕ is A's private key.

needkey(A) : A principal might not know of A's public key.

Message Handling

msg(M, T) : Message M has tag T.

send(M, A) : Message M is an outgoing message intended for A.

recv(M, A) : Message M is an incoming message intended for A.

unpack(U, A, T) : A message for A, received in form U, has been processed and tagged with T.

Protocol State - Server

step0a(A, B) : Recognizing a request for A's public key from B.

step0b(A, B, T) : Preparing to digitally sign a message with tag T containing A's public key.

step0c(A, B, T) : Preparing to send B the digitally signed message containing A's public key.

Protocol State - Approacher

init(A) : Recognizing intent to start a secure session with principal A.

step1a(A) : Preparing to generate a nonce for the session with A.

step1b(A, N_a) : Preparing to create the overture message to send to A after generating N_a.

step1c(A, N_a, T) : Preparing to encrypt the overture message, which has tag T.

step1d(A, N_a, T) : Preparing to send the encrypted overture message.

step1e(A, N_a, T) : Sending A the encrypted overture message.

step4a(A, N_a, T, N_b) : Recognizing A has responded to overture message and provided N_b.

step4b$(A, N_a, \mathcal{T}_i, N_b, \mathcal{T}_j)$: Preparing to send A's encrypted nonce message, which has tag $\mathcal{T}_j$.

sessionApp(A, N_a, N_b) : Concluding that a secure session with A has been established.

Protocol State - Provokee

step2a(A, N_a) : Recognizing that A is trying to start a session, with nonce N_a.

step3a(A, N_a) : Preparing to generate a nonce for the session with A.

step3b(A, N_a, N_b) : Preparing to create overture response message after generating N_b.

step3c(A, N_a, N_b, T) : Preparing to encrypt the overture response message, which has tag T.

step3d(A, N_a, N_b, T) : Preparing to send the encrypted overture response message.

step3e(A, N_a, N_b, T) : Sending A the encrypted overture response message.

sessionProv(A, N_a, N_b) : Concluding that a secure session with A has been established.

2.2.3 Operations

Each rule is universally quantified over the principals and terms in a given protocol model. We omit the explicit notation for readability.

RequestKey:

$[A]\textbf{needkey}(B) \otimes [\![A]\!]\textbf{Server}(C) \otimes !\textbf{AgentNeq}(B, C) \multimap \langle A \rangle \textbf{send}(\textbf{plain}(\textbf{getkey}(B, A)), C)$

If A needs B's public key and knows some server C that is not B, then A can send C a request for B's public key.

GetKeyRequest:

$[A]\textbf{unpack}(plain, A, T) \otimes [A]\textbf{msg}(\textbf{plain}(\textbf{getkey}(B, C)), T) \otimes [\![A]\!]\textbf{Server}(A) \multimap$

$[A]\textbf{step0a}(B, C) \otimes [A]\textbf{needkey}(B)$

If A has unpacked a plaintext message inteded for them tagged with T, if the message contents are a key request for B's key from C, and if A is a server, A begins the process of handling C's request, potentially having to reach out to other servers to obtain B's key.

ConstructKeyMessage:

$[A]\textbf{step0a}(B, C) \otimes [\![A]\!]\textbf{PublicKey}(K_B^+, B) \multimap \exists \mathcal{T}_i.[A]\textbf{step0b}(B, C, \mathcal{T}_i) \otimes$

$[A]\textbf{msg}(\textbf{plain}(\kappa(K_B^+, B)), \mathcal{T}_i)$

If A has begun the process of handling C's key request, and A knows B's public key, then A generates a plain text message containing the key association between B and B's public key and proceeds to the next state step of processing the key request.

EncryptKey:

$[A]\textbf{step0b}(B, C, \mathcal{T}_i) \otimes [A]\textbf{msg}(M, \mathcal{T}_i) \otimes [\![A]\!]\textbf{PrivateKey}(K_A^-, A) \otimes !\textbf{Crypt}(H, K_A^-, M) \multimap$

$[A]\textbf{step0c}(B, C, \mathcal{T}_i) \otimes [A]\textbf{msg}(\textbf{encPri}(H, A), \mathcal{T}_i)$

If A is in the key request processing step of having generated the plain text of the key association,

and A knows their own private key, then A can digitally sign the message and proceed to the next state step.

SendKey:

$[A]\textbf{step0c}(B, C, \mathcal{T}_i) \otimes [A]\textbf{msg}(M, \mathcal{T}_i) \multimap \langle A \rangle \textbf{send}(M, C)$

If A is at the final state step of processing a key request and has digitally signed the message containing B's key association, then A can send the message over the network to C.

LearnKey:

$[A]\textbf{unpack}(encPri(S), A, \mathcal{T}_i) \otimes [\![A]\!]\textbf{Server}(S) \otimes [A]\textbf{msg}(\textbf{plain}(\kappa(K_B^+, B)), \mathcal{T}_i) \multimap$
$[\![A]\!]\textbf{PublicKey}(K_B^+, B)$

If A has decrypted a message intended for A with S's public key, tagged with $\mathcal{T}_i$, if A knows that S is a server, and A has a message that contains B's key association whose tag matches with $\mathcal{T}_i$, then A can learn B's public key.

KeyKnown1:

$[A]\textbf{init}(B) \otimes [\![A]\!]\textbf{PublicKey}(K_B^+, B) \multimap [A]\textbf{step1a}(B)$

If A has the intent to start a session with B, and A knows B's public key, then A can proceed to the first state step of extending an overture.

GenNonce1:

$[A]\textbf{step1a}(B) \multimap \exists N_a.[A]\textbf{step1b}(B, N_a)$

If A is in the first step of beginning the protocol, then A can generate a nonce N_a for the session and proceed to the next state step. The nonce is added to the state step marker in order to differentiate it from other protocol instances which may be in progress.

CreateOverture:

$[A]\textbf{step1b}(B, N_a) \multimap \exists \mathcal{T}_i.[A]\textbf{step1c}(B, N_a, \mathcal{T}_i) \otimes [A]\textbf{msg}(\textbf{plain}(N_a \sim A), \mathcal{T}_i)$

If A has generated their nonce for the session, then they can create a plaintext message containing the nonce and their identifying information, tagged with $\mathcal{T}_i$ and proceed to the next state step. The tag is added to the state step marker in order to track the message once it has been encrypted.

EncryptOverture:

$[A]\textbf{step1c}(B, N_a, \mathcal{T}_i) \otimes [A]\textbf{msg}(\textbf{plain}(M), \mathcal{T}_i) \otimes [\![A]\!]\textbf{PublicKey}(K_B^+, B) \otimes$

$!\textbf{Crypt}(H, K_B^+, \textbf{plain}(M)) \multimap [A]\textbf{step1d}(B, N_a, \mathcal{T}_i) \otimes [A]\textbf{msg}(\textbf{encPub}(H, B), \mathcal{T}_i)$

If A has generated the plaintext overture, if A knows B's public key, and encrypting the message with B's public key generates cyphertext H, then A can carry out the encryption and proceed to the next state step.

SendOverture:

$[A]\textbf{step1d}(B, N_a, \mathcal{T}_i) \otimes [A]\textbf{msg}(M, \mathcal{T}_i) \multimap [A]\textbf{step1e}(B, N_a, \mathcal{T}_i) \otimes \langle A \rangle \textbf{send}(M, B)$

If A has encrypted an overture message for a session with B such that the tag in the message matches the tag in the state step, then A can send the message over the network to B and proceed to the next state step.

ReceiveMessage:

$\langle A \rangle \textbf{send}(M, B) \multimap [C]\textbf{recv}(M, B)$

If A sends a message over the network intended for B, then C can receive it. Essentially, messages can be intercepted by anyone, regardless of the intended recipient.

UnpackPlain:

$[A]\textbf{recv}(\textbf{plain}(M), B) \multimap \exists \mathcal{T}_i.[A]\textbf{unpack}(plain, B, \mathcal{T}_i) \otimes [A]\textbf{msg}(\textbf{plain}(M), \mathcal{T}_i)$

If A receives a message in plaintext form, then A can unpack the message, tag it with $\mathcal{T}_i$ and add

the tagged message to their linear context. The fact that the message was originally received in plain text is noted.

UnpackDecryptPublic:

$[A]$**recv**(**encPub**$(H, B), C) \otimes [\![A]\!]$**PrivateKey**$(K_B^-, B) \otimes$!**Crypt**$(H, K_B^-, M) \multimap$

$\exists \mathcal{T}_i.[A]$**unpack**$(encPub(B), C, \mathcal{T}_i) \otimes [A]$**msg**$(M, \mathcal{T}_i)$

If A has received a message intended for C which was encrypted with B's public key, if A knows B's private key, and if M is the message produced when H is decrypted with B's private key, then A can unpack and decrypt the message, tag it with $\mathcal{T}_i$, and add the tagged message to their linear context. The fact that the message was originally received as a cyphertext encrypted with a B's public key is noted.

UnpackDecryptPrivate:

$[A]$**recv**(**encPri**$(H, B), C) \otimes [\![A]\!]$**PublicKey**$(K_B^+, B) \otimes$!**Crypt**$(H, K_B^+, M) \multimap$

$\exists \mathcal{T}_i.[A]$**unpack**$(encPri(B), C, \mathcal{T}_i) \otimes [A]$**msg**$(M, \mathcal{T}_i)$

If A has received a message intended for C which was encrypted with B's private key, if A knows B's public key, and if M is the message produced when H is decrypted with B's public key, then A can unpack and decrypt the message, tag it with $\mathcal{T}_i$, and add the tagged message to their linear context. The fact that the message was originally received as a cyphertext encrypted with a B's private key is noted.

RealizeProvokee:

$[A]$**unpack**$(encPub(A), A, \mathcal{T}_i) \otimes [A]$**msg**(**plain**$(N_b \sim B), \mathcal{T}_i) \multimap [A]$**step2a**$(B, N_b) \otimes$

$[A]$**msg**(**plain**$(N_b), \mathcal{T}_i) \otimes [A]$**needkey**$(B)$

If A has decrypted a message tagged with $\mathcal{T}_i$, intended for them, with their own private key and they have a plaintext message also tagged with $\mathcal{T}_i$ containing a nonce and B's reference, then A can realize that they are being approached by B to start a session and proceed to the first state

step of their role. In doing so, A adds the nonce in the state step marker, decomposes B's message to just contain the nonce, and notes that they might need B's key.

KeyKnown2:

$[A]\textbf{step2a}(B, N_b) \otimes [\![A]\!]\textbf{PublicKey}(K_B^+, B) \multimap [A]\textbf{step3a}(B, N_b)$

If A has entered the first provokee state step for a session with B, and if A knows B's public key, then A can proceed to the next state step.

GenNonce2:

$[A]\textbf{step3a}(B, N_b) \multimap \exists N_a.[A]\textbf{step3b}(B, N_b, N_a)$

If A has just passed the state step where they confirm their knowledge of B's public key, then A can generate their own nonce N_b for the session and proceed to the next state step, taking note of the nonce in the state marker.

CreateOvertureResponse:

$[A]\textbf{step3b}(B, N_b, N_a) \otimes [A]\textbf{msg}(\textbf{plain}(N_b), \mathcal{T}_i) \multimap \exists \mathcal{T}_j.[A]\textbf{step3c}(B, N_b, N_a, \mathcal{T}_j) \otimes$
$[A]\textbf{msg}(\textbf{plain}(N_b \sim N_a \sim A), \mathcal{T}_j)$

If A has generated a nonce for their session with B and has a message containing the nonce that they received from B, then they can proceed to the next state step and create a new plaintext message, tagged with $\mathcal{T}_j$, containing B's nonce, their own nonce, and their own reference. The tag associated with this message is added to the state marker.

EncryptOvertureResponse:

$[A]\textbf{step3c}(B, N_b, N_a, \mathcal{T}_i) \otimes [A]\textbf{msg}(M, \mathcal{T}_i) \otimes [\![A]\!]\textbf{PublicKey}(K_B^+, B) \otimes !\textbf{Crypt}(H, K_B^+, M) \multimap$
$[A]\textbf{step3d}(B, N_b, N_a, \mathcal{T}_i) \otimes [A]\textbf{msg}(\textbf{encPub}(H, B), \mathcal{T}_i)$

If A has created the plain text overture response with a tag that matches the tag in the state marker for their session with B, if A knows B's public key, and if H is the cyphertext generated when

the message is encrypted with B's public key, then A can proceed to the next state step and encrypt the overture response message with B's public key.

SendOvertureResponse:

$[A]\textbf{step3d}(B, N_b, N_a, \mathcal{T}_i) \otimes [A]\textbf{msg}(M, \mathcal{T}_i) \multimap [A]\textbf{step3e}(B, N_b, N_a, \mathcal{T}_i) \otimes \langle A \rangle \textbf{send}(M, B)$

If A has encrypted the overture response message for their session with B, then they can proceed to the next state step and send the message over the network with the intent for it to go to B.

RealizeProvokeeResponded:

$[A]\textbf{step1e}(B, N_a, \mathcal{T}_i) \otimes [A]\textbf{unpack}(encPub(A), A, \mathcal{T}_j) \otimes [A]\textbf{msg}(\textbf{plain}(N_a \sim N_b \sim B), \mathcal{T}_j) \multimap$
$\exists \mathcal{T}_k.[A]\textbf{step4a}(B, N_a, \mathcal{T}_i, N_b) \otimes [A]\textbf{msg}(\textbf{plain}(N_b), \mathcal{T}_k)$

If A is the approacher in a session with B and has sent the initial overture over the network, if A has received and decrypted a message with tag $\mathcal{T}_j$ which was intended for them and encrypted with their own public key, and if A has a plaintext message also tagged with $\mathcal{T}_j$ which has the nonce N_a that A generated for the session, a new nonce, and B's reference, then A can realize that this is a response from B and move on to the next state step, taking note of the new nonce N_b which was presumably generated by B. In doing so, they also generate a new plaintext message containing just N_b.

EncryptProvokeeNonce:

$[A]\textbf{step4a}(B, N_a, \mathcal{T}_i, N_b) \otimes [A]\textbf{msg}(\textbf{plain}(N_b), \mathcal{T}_j) \otimes [\![A]\!]\textbf{PublicKey}(K_B^+, B) \otimes$
$!\textbf{Crypt}(H, K_B^+, \textbf{plain}(N_b)) \multimap [A]\textbf{step4b}(B, N_a, \mathcal{T}_i, N_b, \mathcal{T}_j) \otimes [A]\textbf{msg}(\textbf{encPub}(H, B), \mathcal{T}_j)$

If A has separated B's nonce from the overture response for their session with B, if A knows B's public key, and if H is the cyphertext generated from encrypting the plaintext nonce with B's public key, then A can encrypt the nonce with B's public key and proceed to the next state step, adding the tag associated with the encrypted message to the state marker.

SecureSessionApproacher:

$[A]\textbf{step4b}(B, N_a, \mathcal{T}_i, N_b, \mathcal{T}_j) \otimes [A]\textbf{msg}(M, \mathcal{T}_j) \multimap [A]\textbf{sessionApp}(B, N_a, N_b) \otimes \langle A \rangle \textbf{send}(M, B)$

If A has encrypted B's nonce in a message tagged with $\mathcal{T}_j$ for their session with B, where the second in the step marker also is also $\mathcal{T}_j$, then A can send the encrypted nonce over the network to B and conclude that they've done all that is necessary to establish a secure session with B.

SecureSessionProvokee:

$[A]\textbf{step3e}(B, N_b, N_a, \mathcal{T}_i) \otimes [A]\textbf{unpack}(encPub(A), A, \mathcal{T}_j) \otimes [A]\textbf{msg}(\textbf{plain}(N_a), \mathcal{T}_j) \multimap$
$[A]\textbf{sessionProv}(B, N_b, N_a)$

If A has received and decrypted a message from B that was encrypted with their own public key after having sent B an overture response for the session, and the decrypted message contains the nonce that they generated for the session, then they can recognize this as B having carried out the final step correctly and conclude that a secure session with B has been established.

2.3 Remarks

The only rule involving a change in multiple principals' knowledge state is **ReceiveMessage**, which models communication over the network. All other changes in a principal's knowledge state are based on from information indexed to them. This rule formulation is deliberate, as it accurately captures the inability to access another principal's knowledge without explicit communication.

Chapter 3

Proof of Adequacy

In this chapter, we prove that our formalism follows the notion of adequacy that we have developed: that it is sound and complete with respect to the actual protocol.

3.1 Correspondence of Principal Knowledge

Since our aim is to prove that the our formalism is adequate for modeling executions of the Needham-Schröder-Lowe protocol, we must characterize the knowledge states of the protocol itself and what is entailed at each step. We first express the properties of the protocol states as statements in the original description language, plain prose. We then translate those statements into propositions in our formalism which we defined to have the same semantics. Figure 3.1 shows the knowledge states we have identified as well as their propositional expression.

Note the distinction between a principal's knowledge about other principals and their knowledge about the session. In particular, we express a principal's knowledge of a key explicitly as a proposition, whereas their awareness of session details (nonces, etc.) is implicit in their knowledge of the protocol step marker, which tracks those details. Accordingly, the former's translation from protocol language to formal propositions is fairly shallow, while the latter's is more nuanced and relies on the protocol progression itself to carry any significant meaning.

			Agent Knowledge Upon Step Completion	**Agent Knowledge Expressed as Judgments**
0.	-	-	*Initial knowledge G:* A knows A's private key A knows S's public key P knows P's private key P knows S's public key S knows S's private key S knows A's public key S knows P's public key	*Corresponding knowledge set Γ:* A knows **PrivateKey**(K_A^-, A) A knows **PublicKey**(K_S^+, S) P knows **PrivateKey**(K_P^-, P) P knows **PublicKey**(K_S^+, S) S knows **PrivateKey**(K_S^-, S) S knows **PublicKey**(K_A^+, A) S knows **PublicKey**(K_P^+, P)
1.	$A \to S:$	A, P	G S **has A's key request**	Γ S has **step0a**(P, A)
2.	$S \to A:$	$\{K_P^+, P\}_{K_S^-}$	G A **knows P's public key** A **can contact P**	Γ A knows **PublicKey**(K_P^+, P) A has **step1a**(P)
3.	$A \to P:$	$\{N_a, A\}_{K_P^+}$	G A knows P's public key. A **has sent an overture to P** P **recognizes the overture**	Γ A knows **PublicKey**(K_P^+, P) A has **step1e**$(P, N_a, \mathcal{T}_i)$, $N_a, \mathcal{T}_i$ fresh P has **step2a**(A, N_a)
4.	$P \to S:$	P, A	G A knows P's public key A has sent an overture to P P recognizes the overture S **has P's key request**	Γ A knows **PublicKey**(K_P^+, P) A has **step1e**$(P, N_a, \mathcal{T}_i)$ P has **step2a**(A, N_a) S has **step0a**(A, P)
5.	$S \to P:$	$\{K_A^+, A\}_{K_S^-}$	G A knows P's public key A has sent an overture to P P **knows A's public key** P **can respond to A's overture**	Γ A knows **PublicKey**(K_P^+, P) A has **step1e**(P, N_a) P knows **PublicKey**(K_A^+, A) P has **step3a**(A, N_a)
6.	$P \to A:$	$\{N_a, N_p, P\}_{K_A^+}$	G A knows P's public key P knows A's public key P **has responded to A's overture** A **recognizes P's response**	Γ A knows **PublicKey**(K_P^+, P) P knows **PublicKey**(K_A^+, A) P has **step3e**$(A, N_a, N_b, \mathcal{T}_j)$, $\mathcal{T}_j, N_b$ fresh A has **step4a**$(P, N_a, \mathcal{T}_i, N_b)$
7.	$A \to P:$	$\{N_p\}_{K_P^+}$	G A knows P's public key P knows A's public key A **has a secure session with P** P **has a secure session with A**	Γ A knows **PublicKey**(K_P^+, P) P knows **PublicKey**(K_A^+, A) A has **sessionApp**(P, N_a, N_b) P has **sessionProv**(A, N_a, N_b)

Figure 3.1: The Needham-Schröder-Lowe protocol, annotated with principals' knowledge at each step. New knowledge is bolded for clarity.

3.2 Correspondence of Protocol State Transitions

Having defined the knowledge states upon completion of each step, we now focus on the transitions between these states, both in the protocol's description language, and in that of our formalism. In the previous section, we defined a correspondence between principal knowledge states at each step in the language of the protocol and the language of the formalism. Our goal here is to do the same for the transitions between knowledge states. Ultimately, we want to show that the formalism allows principals to arrive at knowledge states which mirror those identified in the protocol only by the same actions as specified in the protocol.

The protocol itself is sparsely specified, so we must explicitly state what actions are taken to get from step to step in the protocol. We present another annotation of the protocol in Figure 3.2, this time identifying the intermediate actions of each principal in executing the main steps. Note that the knowledge states explicitly identified in Figure 3.1 are still implicitly present, and principals may make use of that knowledge in these intermediate steps.

In order to fully express the correspondence between principals' actions in the protocol and principals' actions taken in context in our formalism, we must provide a notion of state specific to our formalism. A protocol state is a generic state that follows the grammar given by Definition 1.1, but we refine the definition by giving the exact forms of the assumptions permitted by the formalism:

Definition 3.1. A state of the protocol is a pair of contexts $\Gamma; \Delta$, satisfying:

1. Each assumption in Γ has one of the forms: (i) an operation from Section 2.2.3, (ii) A knows Q, where A is a principal represented on the network and Q is a persistent proposition from Section 2.2.2 that is not a *Non-Indexed Relation*, or (iii) a *Non-Indexed Relation* from Section 2.2.2.
2. Each assumption in Δ has one of the forms: (i) A has Q, where A is a principal represented on the network and Q is a linear proposition from Section 2.2.2 that is not a *Non-Indexed Relation*, or (ii) $!Q$, where Q is a *Non-Indexed Relation* from Section 2.2.2.

			Specified Principal Actions	Intermediate Principal Actions
1.	$A \to S:$	A, P	A sends S a key request	
				S receives A's message
				S recognizes A's message as a key request
				S encodes P's public key association
				S digitally signs the message
2.	$S \to A:$	$\{K_P^+, P\}_{K_S^-}$	S sends A P's key	
				A receives S's message
				A decrypts S's message
				A learns P's public key
				A generates a nonce N_a
				A creates an overture message with N_a and A
				A encrypts the message with P's public key
3.	$A \to P:$	$\{N_a, A\}_{K_P^+}$	A sends P the overture	
				P receives A's message
				P decryptes A's message
				P recognizes A's message as an overture
4.	$P \to S:$	P, A	P sends S a key request	
				S receives P's message
				S recognizes P's message as a key request
				S encodes A's public key association
				S digitally signs the message
5.	$S \to P:$	$\{K_A^+, A\}_{K_S^-}$	S sends P A's key	
				P receives S's message
				P decrypts S's message
				P learns A's public key
				P generates a nonce N_p
				P creates an overture response with N_a, N_p, P
				P encrypts the message with A's public key
6.	$P \to A:$	$\{N_a, N_p, P\}_{K_A^+}$	P sends A an overture response	
				A receives P's message
				A decrypts P's message
				A accepts P's response if the contents are right
				A creates a message containing N_p
				A encrypts the message with P's public key
7.	$A \to P:$	$\{N_p\}_{K_P^+}$	A sends P the final message	
				A believes the session to be secure
				P receives A's message
				P decryptes A's message
				P accepts A's message if the contents are right
				P believes the session to be secure

Figure 3.2: The Needham-Schröder-Lowe protocol, annotated with principals' actions.

Using Definition 3.1, as well as DeYoung and Pfenning's definition of Rewrite Step[1], we can construct derived inference rules which represent state transitions $\Gamma; \Delta \longrightarrow \Gamma'; \Delta'$. We present the rewrite steps for individual rules as the need arises in our proofs, but we provide an illustrative example in Figure 3.3 for the rule **EncryptOverture** from Section 2.2.3:

Operation:

$[A]\mathbf{step1c}(B, N_a, \mathcal{T}_i) \otimes [A]\mathbf{msg}(\mathbf{plain}(M), \mathcal{T}_i) \otimes [\![A]\!]\mathbf{PublicKey}(K_B^+, B) \otimes$
$!\mathbf{Crypt}(H, K_B^+, \mathbf{plain}(M)) \multimap [A]\mathbf{step1d}(B, N_a, \mathcal{T}_i) \otimes [A]\mathbf{msg}(\mathbf{encPub}(H, B), \mathcal{T}_i)$

Corresponding State Transition:

$\Gamma, A\ \mathsf{knows}\ \mathbf{PublicKey}(K_B^+, B), \mathbf{Crypt}(H, K_B^+, \mathbf{plain}(M)); \Delta, [A]\mathbf{step1c}(B, N_a, \mathcal{T}_i),$
$A\ \mathsf{has}\ \mathbf{msg}(\mathbf{plain}(M), \mathcal{T}_i) \longrightarrow \Gamma, A\ \mathsf{knows}\ \mathbf{PublicKey}(K_B^+, B), \mathbf{Crypt}(H, K_B^+, \mathbf{plain}(M)); \Delta,$
$A\ \mathsf{has}\ \mathbf{step1d}(B, N_a, \mathcal{T}_i), A\ \mathsf{has}\ \mathbf{msg}(\mathbf{encPub}(H, B), \mathcal{T}_i)$

Figure 3.3: The **EncryptOverture** rule as a formula in linear logic and a rewrite step.

Now that we have presented our notion of protocol state and rewrite step transitions, as well as given a language in which to construct derivations, we progress to proving the adequacy of our formalism.

3.3 Initial State and Theorem Statement

Our proof concerns instances of the protocol as executed between three principals, A, P, and S, who play the roles of the approacher, provokee, and server, respectively. We claim that our proof strategy generalizes to more complex server situations, where servers have to obtain keys they don't know from other servers in order to handle key requests. For the sake of simplicity, however, we restrict our attention to these three principals.

We begin by defining an initial state $\Gamma_0; \Delta_0$ that represents what we believe to be the minimum initial knowledge for each principal in order to successfully play their role in the protocol.

All principals should know their own private keys, as well as that S is a server. Additionally, A and P should know S's public key so that they can authenticate S's digitally signed messages. To get the protocol execution started, A should also have the intent to establish a secure session with P as well as an awareness of the need for P's public key. This initial state is essentially G from Figure 3.1, with the added knowledge for each of the principals that S is a server and the initial intent for A to start a session with P. Thus we define $\Gamma_0; \Delta_0$ similarly, as in Figure 3.4.

$\Gamma_0 = A \,\mathsf{knows}\, \textbf{PrivateKey}(K_A^-, A), A \,\mathsf{knows}\, \textbf{Server}(S), A \,\mathsf{knows}\, \textbf{PublicKey}(K_S^+, S),$
$S \,\mathsf{knows}\, \textbf{Server}(S), S \,\mathsf{knows}\, \textbf{PublicKey}(K_A^+, A), S \,\mathsf{knows}\, \textbf{PublicKey}(K_P^+, P),$
$S \,\mathsf{knows}\, \textbf{PrivateKey}(K_S^-, S), P \,\mathsf{knows}\, \textbf{PrivateKey}(K_P^-, P), P \,\mathsf{knows}\, \textbf{Server}(S),$
$P \,\mathsf{knows}\, \textbf{PublicKey}(K_S^+, S);$
$\Delta_0 = A \,\mathsf{has}\, \textbf{needkey}(P), A \,\mathsf{has}\, \textbf{init}(P)$

Figure 3.4: The initial state $\Gamma_0; \Delta_0$.

In Section 2.2.2, we defined propositions **sessionApp**(A, N_a, N_b) and **sessionProv**(A, N_a, N_b). We use the former to indicate that the principal who possesses it has successfully played the role of the approacher in an instance of the protocol. The latter is to indicate that the principal who possesses it has successfully played the role of the provokee. Proving adequacy means proving the correctness of this use. We say correctness of use to mean that for all ϵ such that $\Gamma_0; \Delta_0 \xrightarrow{\epsilon} \Gamma; \Delta, A \,\mathsf{has}\, \textbf{sessionApp}(B, N_a, N_b), P \,\mathsf{has}\, \textbf{sessionProv}(A, N_a, N_b)$, we have that ϵ characterizes a state transition sequence that transitions through all knowledge states in the formalism which correspond to the knowledge states identified in the protocol (in Figure 3.1) in the order of the protocol's progression. Specifically, we give a definition for a state transition sequence $\Gamma_0; \Delta_0 \xrightarrow{\epsilon} \Gamma_0, A \,\mathsf{knows}\, \textbf{PublicKey}(K_P^+, P), P \,\mathsf{knows}\, \textbf{PublicKey}(K_A^+, A); S \,\mathsf{has}\, \textbf{needkey}(P),$ $S \,\mathsf{has}\, \textbf{needkey}(A), A \,\mathsf{has}\, \textbf{sessionApp}(B, N_a, N_b), P \,\mathsf{has}\, \textbf{sessionProv}(A, N_a, N_b)$ which we claim to be a transition sequence which adequately corresponds to an execution of the Needham-Schröder-Lowe protocol. That definition is shown in Figure 3.5.

$\Gamma_0; \Delta_0 \xrightarrow{\epsilon_a}$

$\Gamma_1; \Delta_1, S \text{ has } \mathbf{step0a}(P, A) \xrightarrow{\epsilon_b}$

$\Gamma_2, A \text{ knows } \mathbf{PublicKey}(K_P^+, P); \Delta_2, A \text{ has } \mathbf{step1a}(P) \xrightarrow{\epsilon_c}$

$\Gamma_3, A \text{ knows } \mathbf{PublicKey}(K_P^+, P); \Delta_3, A \text{ has } \mathbf{step1e}(P, N_a, \mathcal{T}_i), P \text{ has } \mathbf{step2a}(A, N_a) \xrightarrow{\epsilon_d}$

$\Gamma_4, A \text{ knows } \mathbf{PublicKey}(K_P^+, P); \Delta_4, A \text{ has } \mathbf{step1e}(P, N_a, \mathcal{T}_i), P \text{ has } \mathbf{step2a}(A, N_a),$
$S \text{ has } \mathbf{step0a}(A, P) \xrightarrow{\epsilon_e}$

$\Gamma_5, A \text{ knows } \mathbf{PublicKey}(K_P^+, P), P \text{ knows } \mathbf{PublicKey}(K_A^+, A); \Delta_5, A \text{ has } \mathbf{step1e}(P, N_a, \mathcal{T}_i),$
$P \text{ has } \mathbf{step3a}(A, N_a) \xrightarrow{\epsilon_f}$

$\Gamma_6, A \text{ knows } \mathbf{PublicKey}(K_P^+, P), P \text{ knows } \mathbf{PublicKey}(K_A^+, A); \Delta_6, P \text{ has } \mathbf{step3e}(A, N_a, N_b, \mathcal{T}_j),$
$A \text{ has } \mathbf{step4a}(P, N_a, \mathcal{T}_i, N_b) \xrightarrow{\epsilon_g}$

$\Gamma_0, A \text{ knows } \mathbf{PublicKey}(K_P^+, P), P \text{ knows } \mathbf{PublicKey}(K_A^+, A); S \text{ has } \mathbf{needkey}(P), S \text{ has } \mathbf{needkey}(A),$
$A \text{ has } \mathbf{sessionApp}(P, N_a, N_b), P \text{ has } \mathbf{sessionProv}(A, N_a, N_b)$

Figure 3.5: The definition for the state transition sequence under ϵ.

Now that we have definitions for $\Gamma_0; \Delta_0$ and ϵ, we commence proving our adequacy theorem:

Theorem 3.1. $\Gamma_0; \Delta_0 \xrightarrow{\epsilon'} \Gamma_0, A \text{ knows } \boldsymbol{PublicKey}(K_P^+, P), P \text{ knows } \boldsymbol{PublicKey}(K_A^+, A);$
$S \text{ has } \boldsymbol{needkey}(P), S \text{ has } \boldsymbol{needkey}(A), A \text{ has } \boldsymbol{sessionApp}(P, N_a, N_b), P \text{ has } \boldsymbol{sessionProv}(A, N_a, N_b)$
corresponds to a correct execution of the Needham-Schröder-Lowe protocol if and only if $\epsilon' = \epsilon$

We prove our adequacy theorem (3.1) in two parts. First, we prove soundness, in which we give a derivation of $\Gamma_0; \Delta_0 \xrightarrow{\epsilon} \Gamma_0, A \text{ knows } \mathbf{PublicKey}(K_P^+, P), P \text{ knows } \mathbf{PublicKey}(K_A^+, A);$
$S \text{ has } \mathbf{needkey}(P), S \text{ has } \mathbf{needkey}(A), A \text{ has } \mathbf{sessionApp}(B, N_a, N_b), P \text{ has } \mathbf{sessionProv}(A, N_a, N_b)$
which corresponds to a correct execution of the protocol. Second we prove completeness, that for a given derivation $\Gamma_0; \Delta_0 \xrightarrow{\epsilon'} \Gamma_0, A \text{ knows } \mathbf{PublicKey}(K_P^+, P), P \text{ knows } \mathbf{PublicKey}(K_A^+, A);$
$S \text{ has } \mathbf{needkey}(P), S \text{ has } \mathbf{needkey}(A), A \text{ has } \mathbf{sessionApp}(P, N_a, N_b), P \text{ has } \mathbf{sessionProv}(A, N_a, N_b),$
it must be the case that ϵ' has the form of ϵ.

3.4 Soundness

Theorem 3.2. *There exists a rewrite derivation for the state transition sequence* $\Gamma_0; \Delta_0 \xrightarrow{\epsilon} \Gamma_0, A \text{ knows } \textbf{\textit{PublicKey}}(K_P^+, P), P \text{ knows } \textbf{\textit{PublicKey}}(K_A^+, A); S \text{ has } \textbf{\textit{needkey}}(P), S \text{ has } \textbf{\textit{needkey}}(A), A \text{ has } \textbf{\textit{sessionApp}}(P, N_a, N_b), P \text{ has } \textbf{\textit{sessionProv}}(A, N_a, N_b)$ *which corresponds to a correct execution of the Needham-Schröder-Lowe protocol.*

Proof: We construct a derivation structured around the intermediate state transition sequences $\Gamma_i; \Delta_i \xrightarrow{\epsilon'} \Gamma_{i+1}; \Delta_{i+1}$, showing how ϵ' corresponds to the intermediate steps between explicit protocol steps. For clarity, when presenting our state transitions, we highlight newly introduced propositions.

First, we show the derivation $\Gamma_0; \Delta_0 \xrightarrow{\epsilon_a} \Gamma_1; \Delta_1, S \text{ has } \textbf{step0a}(P, A)$

We start out at the beginning of the protocol execution, with A wanting to start a session with P. A doesn't have P's public key, given by $A \text{ has } \textbf{needkey}(P)$, but they do know that S is a server, represented as $A \text{ knows } \textbf{Server}(S)$. A also knows that P is not the same principal as S. Since agent inequality is valid in our formalism, it can also be derived that $!\textbf{AgentNeq}(P, S)$. Given this information about the state, A can decide to send a message over the network to S, requesting P's public key. So we combine these propositions and apply **RequestKey**, which has the following rewrite rule:

$\Gamma, A \text{ knows } \textbf{Server}(B); \Delta, A \text{ has } \textbf{needkey}(C), !\textbf{AgentNeq}(B, C) \longrightarrow \Gamma, A \text{ knows } \textbf{Server}(B); \Delta, \langle A \rangle \textbf{send}(\textbf{plain}(\textbf{getkey}(B, A)), C)$

Doing so puts us at the state $\Gamma_0; A \text{ has } \textbf{init}(P), \langle A \rangle \textbf{send}(\textbf{plain}(\textbf{getkey}(P, A)), S)$. From here, S can receive A's message. We are in a position to apply **ReceiveMessage**, which has the following rewrite rule:

$\Gamma; \Delta, \langle A \rangle \textbf{send}(M, B) \longrightarrow \Gamma; \Delta, C \text{ has } \textbf{recv}(M, B)$

After the rule is applied, we are at the state $\Gamma_0; A \text{ has } \textbf{init}(P), S \text{ has } \textbf{recv}(\textbf{plain}(\textbf{getkey}(P, A)), S)$. Before S can respond to the message, they first have to process the message and realize that it

is a key request. Since the contents of the message are plaintext, we can apply **UnpackPlain**, which has the following rewrite rule:

$\Gamma; \Delta, A$ has **recv**(**plain**$(M), B) \longrightarrow \Gamma; \Delta, A$ has **unpack**$(plain, B, \mathcal{T}_i), A$ has **msg**$(M, \mathcal{T}_i)$, with $\mathcal{T}_i$ being fresh

Applying the rule brings us to the state $\Gamma_0; A$ has **init**(P), S has **unpack**$(plain, S, \mathcal{T}_i)$, S has **msg**(**plain**(**getkey**$(P, A)), \mathcal{T}_i)$. Upon unpacking the message, S should realize that what they have received is a key request, and that they need to fulfill it. Here we can apply **GetKeyRequest**, which has the following rewrite rule:

Γ, A knows **Server**$(A); \Delta, A$ has **unpack**$(plain, A, T), A$ has **msg**(**plain**(**getkey**$(B, C)), T) \longrightarrow$ Γ, A knows **Server**$(A); \Delta, A$ has **step0a**$(B, C), A$ has **needkey**(B)

Upon application of the rule, we arrive at the state $\Gamma_0; A$ has **init**(P), S has **step0a**(P, A), S has **needkey**(P). So far, we have succeeded in creating a derivation which follows the state transition sequence $\Gamma_0; \Delta_0 \xrightarrow{\epsilon_a} \Gamma_1; \Delta_1, S$ has **step0a**(P, A)

Next we give a derivation for $\Gamma_1; \Delta_1, S$ has **step0a**$(P, A) \xrightarrow{\epsilon_b} \Gamma_2, A$ knows **PublicKey**(K_P^+, P); Δ_2, A has **step1a**(P). In this case, $\Gamma_1 = \Gamma_0$ and $\Delta_1 = S$ has **needkey**$(P), A$ has **init**(P).

Since S has recognized the request and knows P's public key, they can construct a message encoding that information. At this step, we can apply **ConstructKeyMessage**, which has the following rewrite rule:

Γ, A knows **PublicKey**$(K_B^+, B); \Delta, A$ has **step0a**$(B, C) \longrightarrow \Gamma, A$ knows **PublicKey**$(K_B^+, B); \Delta,$ A has **step0b**$(B, C, \mathcal{T}_i), A$ has **msg**(**plain**$(\kappa(K_B^+, B)), \mathcal{T}_i)$, with $\mathcal{T}_i$ being fresh

Applying this rule, we arrive at the state $\Gamma_0; A$ has **init**(P), S has **step0b**$(P, A, \mathcal{T}_j)$, S has **needkey**(P), S has **msg**(**plain**$(\kappa(K_P^+, P)), \mathcal{T}_j)$ Now that S has constructed the message encoding P's public key, they have to digitally sign it before sending it to A. Here we can apply the rule **EncryptKey**, which has the following rewrite rule:

Γ, A knows **PrivateKey**$(K_A^-, A); \Delta, A$ has **step0b**$(B, C, \mathcal{T}_i), M$ has $\mathcal{T}_i$, !**Crypt**$(H, K_A^-, M) \longrightarrow$ Γ, A knows **PrivateKey**$(K_A^-, A); \Delta, A$ has **step0c**$(B, C, \mathcal{T}_i), A$ has **msg**(**encPri**$(H, A), \mathcal{T}_i)$

Note that we are applying the rule even though we do not currently have $!\textbf{Crypt}(H, K_S^-, M)$ in our state. The **Crypt** predicate was introduced in Section 2.2.2 to represent an interface to an outside cryptographic signature that principals can use for encryption and decryption. This signature is separate from the protocol state $\Gamma; \Delta$. It can be introduced into the linear context, but that nuanced separation is not evident in the rewrite rules. Here and in all other instances in which the **Crypt** predicate is introduced, we implicitly understand that it is introduced from an outside signature.

We apply the **EncryptKey** rule, using the $-, +, +$ mode of $!\textbf{Crypt}(H, K_S^-, \textbf{plain}(\kappa(K_P^+, P)))$ to arrive at the state $\Gamma_0; A$ has $\textbf{init}(P), S$ has $\textbf{needkey}(P), S$ has $\textbf{step0c}(P, A, \mathcal{T}_j)$, S has $\textbf{msg}(\textbf{encPri}(H, S), \mathcal{T}_j)$. Now S has to send the encrypted key message back to A. Since the message is digitally signed, S cannot determine which message to send based on its contents, but since their state marker includes the tag of that message, they can use the tag to find the encrypted message. The corresponding rule to apply would be **SendKey**, which has the following rewrite rule:

$\Gamma; \Delta, A$ has $\textbf{step0c}(B, C, \mathcal{T}_i), A$ has $\textbf{msg}(M, \mathcal{T}_i) \longrightarrow \Gamma; \Delta, \langle A \rangle \textbf{send}(M, C)$

Applying this rule brings us to the state $\Gamma_0; A$ has $\textbf{init}(P), S$ has $\textbf{needkey}(P)$, $\langle S \rangle \textbf{send}(\textbf{encPri}(H, S), A)$. A must receive this message, so we apply **ReceiveMessage** to transition to the state $\Gamma_0; A$ has $\textbf{init}(P), S$ has $\textbf{needkey}(P), A$ has $\textbf{recv}(\textbf{encPri}(H, S), A)$. Since this received message is encrypted with S's private key, and A knows S's public key, we can apply **UnpackDecryptPrivate**, given the following rewrite rule:

Γ, A knows $\textbf{PublicKey}(K_B^+, B); \Delta, A$ has $\textbf{recv}(\textbf{encPri}(H, B), C), !\textbf{Crypt}(H, K_B^+, M) \longrightarrow \Gamma,$ A knows $\textbf{PublicKey}(K_B^+, B); \Delta, A$ has $\textbf{unpack}(encPri(B), C, \mathcal{T}_i), A$ has $\textbf{msg}(M, \mathcal{T}_i)$, where $\mathcal{T}_i$ is fresh

Upon A decrypting the message using $!\textbf{Crypt}(H, K_B^+, M)$ in the $+, +, -$ mode, we arrive at the state $\Gamma_0; A$ has $\textbf{init}(P), S$ has $\textbf{needkey}(P), A$ has $\textbf{unpack}(encPri(S), A, \mathcal{T}_k)$, A has $\textbf{msg}(\textbf{plain}(\kappa(K_P^+, P)), \mathcal{T}_k)$. Since A has decrypted a digitally signed message from S containing an encoding of P's public key, we can conclude that A can learn P's public key. At

this point, we can apply the rule **LearnKey**, which has the following rewrite rule:

Γ, A knows **Server**$(B); \Delta, A$ has **unpack**$(encPri(B), A, \mathcal{T}_i), A$ has **msg**(**plain**$(\kappa(K_B^+, B)), \mathcal{T}_i) \longrightarrow$
Γ, A knows **Server**$(B), A$ knows **PublicKey**$(K_B^+, B); \Delta$

We apply the rule to get us to the state Γ_0, A knows **PublicKey**(K_P^+, P); S has **needkey**$(P), A$ has **init**(P). Having arrived at this state so far, we have demonstrated the derivation $\Gamma_1; \Delta_1, S$ has **step0a**(P, A) $\xrightarrow{\epsilon_b} \Gamma_2, A$ knows **PublicKey**$(K_P^+, P); \Delta_2, A$ has **step1a**(P), where $\Gamma_2 = \Gamma_0$ and $\Delta_2 = S$ has **needkey**(P), A has **init**(P).

We move forward, giving the derivation for Γ_2, A knows **PublicKey**$(K_P^+, P); \Delta_2, A$ has **step1a**(P) $\xrightarrow{\epsilon_c} \Gamma_3, A$ knows **PublicKey**$(K_P^+, P); \Delta_3, A$ has **step1e**$(P, N_a, \mathcal{T}_i), P$ has **step2a**(A, N_a). Now that A has learned P's public key, they can move forward with extending an overture to P. First, A needs to acknowledge that they know P's key. We can apply **KeyKnown1**, with the following rewrite rule:

Γ, A knows **PublicKey**$(K_B^+, B); \Delta, A$ has **init**$(B) \longrightarrow \Gamma, A$ knows **PublicKey**$(K_B^+, B); \Delta,$
A has **step1a**(B)

Applying this rule, we arrive at the state Γ_0, A knows **PublicKey**$(K_P^+, P); S$ has **needkey**(P), A has **step1a**(P). Now A can begin the process of extending an overture to P by first generating a nonce for the session. Here, we can apply **GenNonce1**, with the following rewrite rule:

$\Gamma; \Delta, A$ has **step1a**$(B) \longrightarrow \Gamma; \Delta, A$ has **step1b**(B, N_a), where N_a is fresh

We apply the rule to bring us to the state Γ_0, A knows **PublicKey**$(K_P^+, P); S$ has **needkey**(P), A has **step1b**(P, N_a). Now that A has generated a nonce, they must construct a message containing the nonce and their identity to send to P. For this step, we can apply rule **CreateOverture**, which has a rewrite rule of the following form:

$\Gamma; \Delta, A$ has **step1b**$(B, N_a) \longrightarrow \Gamma; \Delta, A$ has **step1c**$(B, N_a, \mathcal{T}_i), A$ has **msg**(**plain**$(N_a \sim A), \mathcal{T}_i)$

Upon applying this rule, we arrive at the state Γ_0, A knows **PublicKey**$(K_P^+, P); S$ has **needkey**(P), A has **step1c**$(P, N_a, \mathcal{T}_l)$, A has **msg**(**plain**$(N_a \sim A), \mathcal{T}_l)$. Now that A has created the overture message, they have to encrypt it with P's public key. We can apply **EncryptOverture**, which

has the following rewrite rule:

Γ, A knows **PublicKey**$(K_B^+, B); \Delta, A$ has **step1c**$(B, N_a, \mathcal{T}_i), A$ has **msg**(**plain**$(M), \mathcal{T}_i)$, !**Crypt**$(H, K_B^+,$ **plain**$(M)) \longrightarrow \Gamma, A$ knows **PublicKey**$(K_B^+, B); \Delta, A$ has **step1d**$(B, N_a, \mathcal{T}_i)$, A has **msg**(**encPub**$(H, B), \mathcal{T}_i)$

With this rule application, using !**Crypt**$(H, K_B^+,$ **plain**$(N_a \sim A))$ in the $-, +, +$ mode, we reach state Γ_0, A knows **PublicKey**$(K_P^+, P); S$ has **needkey**(P), A has **step1d**$(P, N_a, \mathcal{T}_l)$, A has **msg**(**encPub**$(H, P), \mathcal{T}_l)$. Since A has encrypted the overture message, they can now send it to P. The rule **SendOverture**, with the following rewrite rule, would apply here:

$\Gamma; \Delta, A$ has **step1d**$(B, N_a, \mathcal{T}_i), A$ has **msg**$(M, \mathcal{T}_i) \longrightarrow \Gamma; \Delta, A$ has **step1e**$(B, N_a, \mathcal{T}_i), \langle A \rangle$**send**$(M, B)$

We apply the rule and arrive at the state Γ_0, A knows **PublicKey**$(K_P^+, P); S$ has **needkey**(P), A has **step1e**$(P, N_a, \mathcal{T}_l)$, $\langle A \rangle$**send**(**encPub**$(H, P), P)$. For P to receive the message, we apply **ReceiveMessage**, bringing us to the state Γ_0, A knows **PublicKey**$(K_P^+, P); S$ has **needkey**(P), A has **step1e**$(P, N_a, \mathcal{T}_l), P$ has **recv**(**encPub**$(H, P), P)$. Since the message was encrypted with P's public key and P knows their own private key, they can decrypt the message. Here we can apply **UnpackDecryptPublic**, which has the following rewrite rule:

Γ, A knows **PrivateKey**$(K_B^-, B); \Delta, A$ has **recv**(**encPub**$(H, B), C)$, !**Crypt**$(H, K_B^-, M) \longrightarrow \Gamma$, A knows **PrivateKey**$(K_B^-, B); \Delta, A$ has **unpack**$(encPub(B), C, \mathcal{T}_i), A$ has **msg**$(M, \mathcal{T}_i)$, where $\mathcal{T}_i$ is fresh

Applying this rule, using !**Crypt**(H, K_B^-, M) in the $+, +, -$ mode, we arrive at the state Γ_0, A knows **PublicKey**$(K_P^+, P); A$ has **step1e**$(P, N_a, \mathcal{T}_l), P$ has **unpack**$(encPub(P), P, \mathcal{T}_m)$, S has **needkey**$(P), P$ has **msg**(**plain**$(N_a \sim A), \mathcal{T}_m)$. P, upon examining this message, can recognize it as an overture and realize that they are being approached to start a session. At this point, we can apply **RealizeProvokee**, which has the following rewrite rule:

$\Gamma; \Delta, A$ has **unpack**$(encPub(A), A, \mathcal{T}_i), A$ has **msg**(**plain**$(N_b \sim B), \mathcal{T}_i) \longrightarrow \Gamma; \Delta, A$ has **step2a**(B, N_b), A has **msg**(**plain**$(N_b), \mathcal{T}_i), A$ has **needkey**(B)

With this rule application, we arrive at the state Γ_0, A knows **PublicKey**(K_P^+, P); S has **needkey**$(P), A$ has **step1e**$(P, N_a, \mathcal{T}_l), P$ has **step2a**(A, N_a), P has **msg**(**plain**$(N_a), \mathcal{T}_m)$,

P has **needkey**(A). In addition, we have also just shown the derivation Γ_2, A knows **PublicKey**(K_P^+, P); Δ_2, A has **step1a**$(P) \xrightarrow{\epsilon_c} \Gamma_3, A$ knows **PublicKey**$(K_P^+, P); \Delta_3, A$ has **step1e**$(P, N_a, \mathcal{T}_i), P$ has **step2a**(A, N_a). Here $\Gamma_3 = \Gamma_0, A$ knows **PublicKey**(K_P^+, P) and $\Delta_3 = S$ has **needkey**(P), P has **msg**(**plain**$(N_a), \mathcal{T}_m), P$ has **needkey**(A).

We now give the derivation for Γ_3, A knows **PublicKey**$(K_P^+, P); \Delta_3, A$ has **step1e**$(P, N_a, \mathcal{T}_i)$, P has **step2a**$(A, N_a) \xrightarrow{\epsilon_d} \Gamma_4, A$ knows **PublicKey**$(K_P^+, P); \Delta_4, A$ has **step1e**$(P, N_a, \mathcal{T}_i)$, P has **step2a**$(A, N_a), S$ has **step0a**(A, P).

Upon realizing that that they are the provokee in a session being opened by A, P must request A's public key from the server. As before, we apply **RequestKey** to get us to the state Γ_0, A knows **PublicKey**$(K_P^+, P); S$ has **needkey**$(P), A$ has **step1e**$(P, N_a, \mathcal{T}_l), P$ has **step2a**(A, N_a), P has **msg**(**plain**$(N_a), \mathcal{T}_m)$, $\langle P \rangle$**send**(**plain**(**getkey**$(A, P)), S)$. S receives the message, and we apply **ReceiveMessage** to arrive at the state Γ_0, A knows **PublicKey**$(K_P^+, P); S$ has **needkey**(P), A has **step1e**$(P, N_a, \mathcal{T}_l), P$ has **step2a**$(A, N_a), P$ has **msg**(**plain**$(N_a), \mathcal{T}_m)$, S has **recv**(**plain**(**getkey**$(A, P)), S)$. S unpacks the message, then, and we apply **UnpackPlain** to arrive at the state Γ_0, A knows **PublicKey**$(K_P^+, P); S$ has **needkey**$(P), A$ has **step1e**$(P, N_a, \mathcal{T}_l)$, P has **step2a**$(A, N_a), P$ has **msg**(**plain**$(N_a), \mathcal{T}_m), S$ has **unpack**$(plain, P, \mathcal{T}_n)$, S has **msg**(**plain**(**getkey**$(A, P)), \mathcal{T}_n)$. When S examines the messages, they realize that they have received a request from P for A's public key. Applying **GetKeyRequest**, we arrive at the state Γ_0, A knows **PublicKey**$(K_P^+, P); S$ has **needkey**$(P), A$ has **step1e**$(P, N_a, \mathcal{T}_l), P$ has **step2a**(A, N_a), P has **msg**(**plain**$(N_a), \mathcal{T}_m), S$ has **step0a**(A, P), S has **needkey**(A).

Reaching this state, we have shown the derivation for Γ_3, A knows **PublicKey**$(K_P^+, P); \Delta_3$, A has **step1e**$(P, N_a, \mathcal{T}_i), P$ has **step2a**$(A, N_a) \xrightarrow{\epsilon_d} \Gamma_4, A$ knows **PublicKey**$(K_P^+, P); \Delta_4$, A has **step1e**$(P, N_a, \mathcal{T}_i), P$ has **step2a**$(A, N_a), S$ has **step0a**(A, P). In the derivation, we have that $\Gamma_4 = \Gamma_0, A$ knows **PublicKey**(K_P^+, P) and $\Delta_4 = S$ has **needkey**$(P), P$ has **msg**(**plain**$(N_a), \mathcal{T}_m)$, S has **needkey**(A).

Next we show the derivation for Γ_4, A knows **PublicKey**$(K_P^+, P); \Delta_4, A$ has **step1e**$(P, N_a, \mathcal{T}_i)$, P has **step2a**$(A, N_a), S$ has **step0a**$(A, P) \xrightarrow{\epsilon_e} \Gamma_5, A$ knows **PublicKey**(K_P^+, P), P knows **PublicKey**$(K_A^+, A); \Delta_5, A$ has **step1e**$(P, N_a, \mathcal{T}_i), P$ has **step3a**(A, N_a).

At the current state, S must respond to P's key request by encoding A's public key in a message. Applying **ConstructKeyMessage**, we reach the state Γ_0, A knows **PublicKey**(K_P^+, P); S has **needkey**$(P), S$ has **needkey**$(A), A$ has **step1e**$(P, N_a, \mathcal{T}_l), P$ has **step2a**(A, N_a), P has **msg**(**plain**$(N_a), \mathcal{T}_m)$, S has **step0b**$(A, P, \mathcal{T}_o)$, S has **msg**(**plain**$(\kappa(K_A^+, A)), \mathcal{T}_o)$. S then digitally signs the message. In turn, we apply the rule **EncryptKey**, with the **Crypt** predicate in mode $-, +, +$, to arrive at the state Γ_0, A knows **PublicKey**$(K_P^+, P); S$ has **needkey**(P), S has **needkey**$(A), A$ has **step1e**$(P, N_a, \mathcal{T}_l), P$ has **step2a**(A, N_a), P has **msg**(**plain**$(N_a), \mathcal{T}_m)$, S has **step0c**$(A, P, \mathcal{T}_o)$, S has **msg**(**encPri**$(H, S), \mathcal{T}_o)$. After encrypting the message, S then sends it to P. We apply **SendMessage** to arrive at the state Γ_0, A knows **PublicKey**$(K_P^+, P); S$ has **needkey**$(P), S$ has **needkey**$(A), A$ has **step1e**$(P, N_a, \mathcal{T}_l)$, P has **step2a**$(A, N_a), P$ has **msg**(**plain**$(N_a), \mathcal{T}_m)$, $\langle S \rangle$**send**(**encPri**$(H, S), P)$.

After S sends the message, P receives it. We apply the **ReceiveMessage** rule to arrive at the state Γ_0, A knows **PublicKey**$(K_P^+, P); S$ has **needkey**$(P), S$ has **needkey**$(A), A$ has **step1e**$(P, N_a, \mathcal{T}_l)$, P has **step2a**$(A, N_a), P$ has **msg**(**plain**$(N_a), \mathcal{T}_m)$, P has **recv**(**encPri**$(H, S), P)$. P then unpacks and decrypts the message. We apply **UnpackDecryptPrivate**, bringing us to the state Γ_0, A knows **PublicKey**$(K_P^+, P); S$ has **needkey**$(P), S$ has **needkey**$(A), A$ has **step1e**$(P, N_a, \mathcal{T}_l)$, P has **step2a**$(A, N_a), P$ has **msg**(**plain**$(N_a), \mathcal{T}_m)$, P has **unpack**$(encPri(S), P, \mathcal{T}_p)$, P has **msg**(**plain**$(\kappa(K_A^+, A)), \mathcal{T}_p)$. Upon examining the decrypted messages, P realizes that this is the response from S, and that they can learn A's public key. Applying the rule **LearnKey**, we arrive at the state Γ_0, A knows **PublicKey**$(K_P^+, P), P$ knows **PublicKey**$(K_A^+, A); S$ has **needkey**(P), S has **needkey**$(A), A$ has **step1e**$(P, N_a, \mathcal{T}_l), P$ has **step2a**$(A, N_a), P$ has **msg**(**plain**$(N_a), \mathcal{T}_m)$.

Upon learning A's public key, P can proceed to the next step of the protocol, preparing to respond to A's overture. We apply **KeyKnown2**, with the following rewrite rule:
ΓA knows **PublicKey**$(K_B^+, B); \Delta, A$ has **step2a**$(B, N_b) \longrightarrow \Gamma A$ knows **PublicKey**$(K_B^+, B); \Delta$,

A has **step3a**(B, N_b)

Applying the rule, we arrive at the state Γ_0, A knows **PublicKey**$(K_P^+, P), P$ knows **PublicKey**(K_A^+, A); S has **needkey**$(P), S$ has **needkey**$(A), A$ has **step1e**$(P, N_a, \mathcal{T}_l), P$ has **msg**(**plain**$(N_a), \mathcal{T}_m)$, P has **step3a**(A, N_a). At this point, we have given the derivation Γ_4, A knows **PublicKey**$(K_P^+, P); \Delta_4$, A has **step1e**$(P, N_a, \mathcal{T}_i), P$ has **step2a**$(A, N_a), S$ has **step0a**$(A, P) \xrightarrow{\epsilon_e} \Gamma_5, A$ knows **PublicKey**(K_P^+, P), P knows **PublicKey**$(K_A^+, A); \Delta_5, A$ has **step1e**$(P, N_a, \mathcal{T}_i), P$ has **step3a**(A, N_a). We see here that $\Gamma_5 = \Gamma_0, A$ knows **PublicKey**$(K_P^+, P), P$ knows **PublicKey**(K_A^+, A) and $\Delta_5 = S$ has **needkey**(P), S has **needkey**$(A), P$ has **msg**(**plain**$(N_a), \mathcal{T}_m)$.

We now give a derivation for Γ_5, A knows **PublicKey**$(K_P^+, P), P$ knows **PublicKey**$(K_A^+, A); \Delta_5$, A has **step1e**$(P, N_a, \mathcal{T}_i), P$ has **step3a**$(A, N_a) \xrightarrow{\epsilon_f} \Gamma_6, A$ knows **PublicKey**(K_P^+, P), P knows **PublicKey**$(K_A^+, A); \Delta_6, P$ has **step3e**$(A, N_a, N_b, \mathcal{T}_j), A$ has **step4a**$(P, N_a, \mathcal{T}_i, N_b)$

Now that P has recognized that they know A's public key, they can generate their own nonce for the session. We apply the rule **GenNonce2**, which has the following rewrite rule:
$\Gamma; \Delta, A$ has **step3a**$(B, N_b) \longrightarrow \Gamma; \Delta, A$ has **step3b**(B, N_b, N_a), where N_a is fresh

Having applied the rule, we now arrive at the state Γ_0, A knows **PublicKey**(K_P^+, P), P knows **PublicKey**$(K_A^+, A); S$ has **needkey**$(P), S$ has **needkey**$(A), A$ has **step1e**$(P, N_a, \mathcal{T}_l)$, P has **msg**(**plain**$(N_a), \mathcal{T}_m)$, P has **step3b**(A, N_a, N_p). Since P has A's nonce and has also generated their own nonce, they can construct an overture response message. We apply the rule **CreateOvertureResponse**, which has the following rewrite rule:
$\Gamma; \Delta, A$ has **step3b**$(B, N_b, N_a), A$ has **msg**(**plain**$(N_b), \mathcal{T}_i) \longrightarrow \Gamma; \Delta, A$ has **step3c**$(B, N_b, N_a, \mathcal{T}_j)$, A has **msg**(**plain**$(N_b \sim N_a \sim A), \mathcal{T}_j)$

Applying the rule brings us to the state Γ_0, A knows **PublicKey**$(K_P^+, P), P$ knows **PublicKey**(K_A^+, A); S has **needkey**$(P), S$ has **needkey**$(A), A$ has **step1e**$(P, N_a, \mathcal{T}_l), P$ has **step3c**$(A, N_a, N_p, \mathcal{T}_q)$, P has **msg**(**plain**$(N_a \sim N_p \sim P), \mathcal{T}_q)$.
Before sending the message, though, P must encrypt it with A's public key, so we apply rule **EncryptOvertureResponse**, which has the following rewrite rule:

Γ, A knows **PublicKey**$(K_B^+, B); \Delta, A$ has **step3c**$(B, N_b, N_a, \mathcal{T}_i), A$ has **msg**$(M, \mathcal{T}_i)$, !**Crypt**(H, K_B^+, M)
$\longrightarrow \Gamma, A$ knows **PublicKey**$(K_B^+, B); \Delta, A$ has **step3d**$(B, N_b, N_a, \mathcal{T}_i), A$ has **msg**(**encPub**$(H, B), \mathcal{T}_i)$

Applying the rule and using the $-, +, +$ mode of **Crypt** brings us to the state Γ_0,
A knows **PublicKey**$(K_P^+, P), P$ knows **PublicKey**$(K_A^+, A); S$ has **needkey**$(P), S$ has **needkey**(A),
A has **step1e**$(P, N_a, \mathcal{T}_l)$, P has **step3d**$(A, N_a, N_p, \mathcal{T}_q)$, P has **msg**(**encPub**$(H, A), \mathcal{T}_q)$. Now that P has encrypted the overture response, they can send it to A. Here we apply **SendOvertureResponse**, which has the following rewrite rule:
$\Gamma; \Delta, A$ has **step3d**$(B, N_b, N_a, \mathcal{T}_i), A$ has **msg**$(M, \mathcal{T}_i) \longrightarrow \Gamma; \Delta, A$ has **step3e**$(B, N_b, N_a, \mathcal{T}_i)$,
$\langle A \rangle$**send**(M, B)

We apply the rule to arrive at the state Γ_0, A knows **PublicKey**$(K_P^+, P), P$ knows **PublicKey**(K_A^+, A);
S has **needkey**$(P), S$ has **needkey**$(A), A$ has **step1e**$(P, N_a, \mathcal{T}_l)$, P has **step3e**$(A, N_a, N_p, \mathcal{T}_q)$,
$\langle P \rangle$**send**(**encPub**$(H, A), A)$. After P sends the message, A receives it, so we apply the **ReceiveMessage** rule and then arrive at the state Γ_0, A knows **PublicKey**(K_P^+, P),
P knows **PublicKey**$(K_A^+, A); S$ has **needkey**$(P), S$ has **needkey**$(A), A$ has **step1e**$(P, N_a, \mathcal{T}_l)$,
P has **step3e**$(A, N_a, N_p, \mathcal{T}_q)$, A has **recv**(**encPub**$(H, A), A)$. A unpacks and decrypts the message. In turn, we apply rule **DecryptUnpackPublic**, bringing us to the state Γ_0,
A knows **PublicKey**$(K_P^+, P), P$ knows **PublicKey**$(K_A^+, A); S$ has **needkey**$(P), S$ has **needkey**(A),
A has **step1e**$(P, N_a, \mathcal{T}_l), P$ has **step3e**$(A, N_a, N_p, \mathcal{T}_q)$, A has **unpack**$(encPub(A), A, \mathcal{T}_r)$,
A has **msg**(**plain**$(N_a \sim N_p \sim P), \mathcal{T}_r)$.

Upon examining the decrypted messages, A should be able to deduce that P has responded correctly to their overture. We can apply rule **RealizeProvokeeResponded**, which has the following rewrite rule:
$\Gamma; \Delta, A$ has **step1e**$(B, N_a, \mathcal{T}_i), A$ has **unpack**$(encPub(A), A, \mathcal{T}_j), A$ has **msg**$(N_a \sim N_b \sim B, \mathcal{T}_j) \longrightarrow$
$\Gamma; \Delta, A$ has **step4a**$(B, N_a, \mathcal{T}_i, N_b), A$ has **msg**(**plain**$(N_b), \mathcal{T}_k)$, where $\mathcal{T}_k$ is fresh

We apply the rule and arrive at the state Γ_0, A knows **PublicKey**$(K_P^+, P), P$ knows **PublicKey**(K_A^+, A);
S has **needkey**$(P), S$ has **needkey**$(A), P$ has **step3e**$(A, N_a, N_p, \mathcal{T}_q)$, A has **step4a**$(P, N_a, \mathcal{T}_l, N_p)$,

A has **msg**(**plain**$(N_p), \mathcal{T}_s)$. At this point, we have given the derivation for Γ_5, A knows **PublicKey**(K_P^+, P), P knows **PublicKey**$(K_A^+, A); \Delta_5, A$ has **step1e**$(P, N_a, \mathcal{T}_i), P$ has **step3a**$(A, N_a) \xrightarrow{\epsilon_f} \Gamma_6$, A knows **PublicKey**$(K_P^+, P), P$ knows **PublicKey**$(K_A^+, A); \Delta_6, P$ has **step3e**$(A, N_a, N_b, \mathcal{T}_j)$, A has **step4a**$(P, N_a, \mathcal{T}_i, N_b)$. Here, we have that $\Gamma_6 = \Gamma_0, A$ knows **PublicKey**(K_P^+, P), P knows **PublicKey**(K_A^+, A) and $\Delta_6 = S$ has **needkey**$(P), S$ has **needkey**$(A), A$ has **msg**(**plain**$(N_p), \mathcal{T}_s)$. Additionally, although the message tag subscripts don't match up completely between the state transition sequence and the actual derivation, the distinction between tags occuring in the same state is what is important.

Finally, we show the derivation for Γ_6, A knows **PublicKey**$(K_P^+, P), P$ knows **PublicKey**(K_A^+, A); Δ_6, P has **step3e**$(A, N_a, N_b, \mathcal{T}_j), A$ has **step4a**$(P, N_a, \mathcal{T}_i, N_b) \xrightarrow{\epsilon_g} \Gamma_0, A$ knows **PublicKey**(K_P^+, P), P knows **PublicKey**$(K_A^+, A); S$ has **needkey**$(P), S$ has **needkey**$(A), A$ has **sessionApp**(P, N_a, N_b), P has **sessionProv**(A, N_a, N_b)

Having processed P's response, A is now ready to encrypt P's nonce and send it back. We apply the rule **EncryptProvokeeNonce**, which has the following rewrite rule:
Γ, A knows **PublicKey**$(K_B^+, B); \Delta, A$ has **step4a**$(B, N_a, \mathcal{T}_i, N_b), A$ has **msg**(**plain**$(N_b), \mathcal{T}_j)$, !**Crypt**$(H, K_B^+,$ **plain**$(N_b)) \longrightarrow \Gamma; \Delta, A$ has **step4b**$(B, N_a, \mathcal{T}_i, N_b, \mathcal{T}_j), A$ has **msg**(**encPub**$(H, B), \mathcal{T}_j)$

Applying the rule and using the $-, +, +$ mode of **Crypt**, we arrive at the state Γ_0, A knows **PublicKey**$(K_P^+, P), P$ knows **PublicKey**$(K_A^+, A); S$ has **needkey**$(P), S$ has **needkey**(A), P has **step3e**$(A, N_a, N_p, \mathcal{T}_q), A$ has **step4b**$(P, N_a, \mathcal{T}_l, N_p, \mathcal{T}_s), A$ has **msg**(**encPub**$(H, P), \mathcal{T}_s)$. After encrypting the message, A is ready to send it off to P and conclude that they have established a secure session. We apply **SecureSessionApproacher** to that end, which has the following rewrite rule:
$\Gamma; \Delta A$ has **step4b**$(B, N_a, \mathcal{T}_i, N_b, \mathcal{T}_j), A$ has **msg**$(M, \mathcal{T}_j) \longrightarrow \Gamma; \Delta, A$ has **sessionApp**(B, N_a, N_b), $\langle A \rangle$**send**(M, B)

We apply the rule and arrive at the state Γ_0, A knows **PublicKey**$(K_P^+, P), P$ knows **PublicKey**(K_A^+, A);

S has **needkey**(P), S has **needkey**(A), P has **step3e**$(A, N_a, N_p, \mathcal{T}_q)$, A has **sessionApp**(P, N_a, N_p), $\langle A \rangle$**send**(**encPub**$(H, P), P)$. Then P receives the message and so we apply **ReceiveMessage**. This rule application brings us to the state Γ_0, A knows **PublicKey**(K_P^+, P), P knows **PublicKey**(K_A^+, A); S has **needkey**(P), S has **needkey**(A), P has **step3e**$(A, N_a, N_p, \mathcal{T}_q)$, A has **sessionApp**(P, N_a, N_p), P has **recv**(**encPub**$(H, P), P)$. After receiving the message, P then unpacks and decrypts it, We then apply the **UnpackDecryptPublic** rule, bringing us to the state Γ_0, A knows **PublicKey**(K_P^+, P), P knows **PublicKey**(K_A^+, A); S has **needkey**(P), S has **needkey**(A), P has **step3e**$(A, N_a, N_p, \mathcal{T}_q)$, A has **sessionApp**(P, N_a, N_p), P has **unpack**$(encPub(P), P, \mathcal{T}_t)$, P has **msg**(**plain**$(N_p), \mathcal{T}_t)$.

Upon examining the messages they've processed, P realizes that the nonce that A sent matches with their own nonce and so then can conclude that they have a secure session with A. We apply the **SecureSessionProvokee** rule, which has the following rewrite rule:
$\Gamma; \Delta, A$ has **step3e**$(B, N_b, N_a, \mathcal{T}_i), A$ has **unpack**$(encPub(A), A, \mathcal{T}_j), A$ has **msg**(**plain**$(N_a), \mathcal{T}_j)$
$\longrightarrow \Gamma; \Delta, A$ has **sessionProv**(B, N_b, N_a)

Once we apply the rule, we arrive at the state Γ_0, A knows **PublicKey**(K_P^+, P), P knows **PublicKey**(K_A^+, A); S has **needkey**(P), S has **needkey**(A), A has **sessionApp**(P, N_a, N_p), P has **sessionProv**(A, N_a, N_p). This application concludes the final state transition sequence we needed to show, Γ_6, A knows **PublicKey**(K_P^+, P), P knows **PublicKey**(K_A^+, A); Δ_6, P has **step3e**$(A, N_a, N_b, \mathcal{T}_j)$, A has **step4a**$(P, N_a, \mathcal{T}_i, N_b) \xrightarrow{\epsilon_g} \Gamma_0$, A knows **PublicKey**(K_P^+, P), P knows **PublicKey**(K_A^+, A); S has **needkey**(P), S has **needkey**(A), A has **sessionApp**(P, N_a, N_b), P has **sessionProv**(A, N_a, N_b).

We have shown the existence of a state transition sequence derivation which corresponds to a correct execution of the protocol[1]. Thus, our formalism is sound with respect to the protocol. □

[1] Additionally, in Appendix B, we define a proof language, as well as a trace of the derivation in that language

3.5 Completeness

Theorem 3.3. *For all state transition sequences* $\Gamma_0; \Delta_0 \xrightarrow{\epsilon'} \Gamma_0, A$ knows ***PublicKey***(K_P^+, P), P knows ***PublicKey***$(K_A^+, A); S$ has ***needkey***$(P), S$ has ***needkey***$(A), A$ has ***sessionApp***(P, N_a, N_p), P has ***sessionProv***(A, N_a, N_p) *,* ϵ' *has the form of* ϵ*.*

Proof: We consider the state Γ_0, A knows **PublicKey**$(K_P^+, P), P$ knows **PublicKey**(K_A^+, A); S has **needkey**$(P), S$ has **needkey**$(A), A$ has **sessionApp**$(P, N_a, N_p), P$ has **sessionProv**(A, N_a, N_p), which we claim represents a terminal state following an execution of the protocol. We invert the rewrite rules and work backwards to arrive at $\Gamma_0; \Delta_0$. We then see that the state transition sequence must have followed the derivation form given by ϵ. As in the the proof of soundness, we highlight newly introduced propositions for clarity. Additionally, for the sake of brevity, we do not explicitly present the rewrite rule forms in this proof. A comprehensive list of the rewrite rules can also be found in Appendix A.

We begin with Γ_0, A knows **PublicKey**$(K_P^+, P), P$ knows **PublicKey**$(K_A^+, A); S$ has **needkey**(P), S has **needkey**$(A), A$ has **sessionApp**$(P, N_a, N_p), P$ has **sessionProv**(A, N_a, N_p). It must be the case that **SecureSessionProvokee** was the most recently applied rule, as all of the propositions except for P has **sessionProv**(A, N_a, N_p) are generated on the right hand side of the rewrite rules along with other propositions that are not in the current context. So they must have been consumed before P has **sessionProv**(A, N_a, N_p) was generated.

By inversion on the rule, then, we now consider the state Γ_0, A knows **PublicKey**(K_P^+, P), P knows **PublicKey**$(K_A^+, A); S$ has **needkey**$(P), S$ has **needkey**$(A), A$ has **sessionApp**(P, N_a, N_p), P has **step3e**$(A, N_a, N_p, \mathcal{T}_i)$, P has **unpack**$(encPub(P), P, \mathcal{T}_j)$, P has **msg**(**plain**$(N_p), \mathcal{T}_j)$. To have arrived at this state, we must have just applied the **UnpackDecryptPublic** rule. Again, all the propositions except for P has **unpack**$(encPub(P), P, \mathcal{T}_j)$ and P has **msg**(**plain**$(N_p), \mathcal{T}_j)$ are generated along with other propositions which do not appear in the current context. The matching $\mathcal{T}_j$ in these two exceptions, as well as the $encPub(P)$ term indicates that they are the result

of the same **UnpackDecryptPublic** rule application.

We invert on the rule and now consider the state Γ_0, A knows **PublicKey**(K_P^+, P), P knows **PublicKey**(K_A^+, A); S has **needkey**(P), S has **needkey**(A), A has **sessionApp**(P, N_a, N_p), P has **step3e**$(A, N_a, N_p, \mathcal{T}_i)$, P has **recv**(**encPub**$(H, P), P)$. As mentioned in the soundness proof in Section 3.4, we consider the **Crypt** predicate as defined in an external signature. So while we do invert on it, we do not include it in our context after the inversion. By the same line of reasoning as before, P has **recv**(**encPub**$(H, P), P)$ is the only proposition produced on the right hand side of a rule such that all right hand propositions are present in the current state. So the rule which must have generated it is **ReceiveMessage**.

We want to invert on the rule, but first we must determine whether the inversion gives us the premise $\langle A \rangle$**send**(**encPub**$(H, P), P)$, $\langle P \rangle$**send**(**encPub**$(H, P), P)$, or $\langle S \rangle$**send**(**encPub**$(H, P), P)$. If we choose either of the latter two, our derivation ends up at a dead end, so it must be the first. Then upon applying inversion, our current inverted state must be Γ_0, A knows **PublicKey**(K_P^+, P), P knows **PublicKey**(K_A^+, A); S has **needkey**(P), S has **needkey**(A), A has **sessionApp**(P, N_a, N_p), P has **step3e**$(A, N_a, N_p, \mathcal{T}_i)$, $\langle A \rangle$**send**(**encPub**$(H, P), P)$. From here, we see that the next rule to invert on must be **SecureSessionApproacher**.

Taking $\langle A \rangle$**send**(**encPub**$(H, P), P)$ and A has **sessionApp**(P, N_a, N_p) to be the right hand propositions, we invert the rule and arrive at the state Γ_0, A knows **PublicKey**(K_P^+, P), P knows **PublicKey**(K_A^+, A); S has **needkey**(P), S has **needkey**(A), P has **step3e**$(A, N_a, N_p, \mathcal{T}_i)$, A has **step4b**$(P, N_a, \mathcal{T}_j, N_p, \mathcal{T}_k)$, A has **msg**$(M, \mathcal{T}_k)$. We don't have explicit information as to what the contents of M are, but the matching tags in A has **step4b**$(P, N_a, \mathcal{T}_j, N_p, \mathcal{T}_k)$ and A has **msg**$(M, \mathcal{T}_k)$ indicate that the rule which generated this state must have been **EncryptProvokeeNonce**, and that M must then have the form **encPub**(H, P).

Taking A has **step4b**$(P, N_a, \mathcal{T}_j, N_p, \mathcal{T}_k)$ and A has **msg**(**encPub**$(H, P), \mathcal{T}_k)$ to be the right hand propositions, we invert on the rewrite rule and arrive at the state Γ_0, A knows **PublicKey**(K_P^+, P), P knows **PublicKey**(K_A^+, A); S has **needkey**(P), S has **needkey**(A), P has **step3e**$(A, N_a, N_p, \mathcal{T}_i)$,

A has **step4a**$(P, N_a, \mathcal{T}_j, N_p)$, A has **msg**(**plain**$(N_p), \mathcal{T}_k)$.

At this point, we have shown that all state transition sequence derivations $\Gamma_0; \Delta_0 \xrightarrow{\epsilon'} \Gamma_0$, A knows **PublicKey**(K_P^+, P), P knows **PublicKey**(K_A^+, A); S has **needkey**(P), S has **needkey**(A), A has **sessionApp**(P, N_a, N_p), P has **sessionProv**(A, N_a, N_p) must end with the form given by Γ_6, A knows **PublicKey**(K_P^+, P), P knows **PublicKey**(K_A^+, A); Δ_6, P has **step3e**$(A, N_a, N_b, \mathcal{T}_j)$, A has **step4a**$(P, N_a, \mathcal{T}_i, N_b) \xrightarrow{\epsilon_g} \Gamma_0, A$ knows **PublicKey**(K_P^+, P), P knows **PublicKey**(K_A^+, A); S has **needkey**(P), S has **needkey**(A), A has **sessionApp**(P, N_a, N_b), P has **sessionProv**(A, N_a, N_b).

Returning our attention to the derivation at hand, we must have arrived at our current state from the **RealizeProvokeeResponded** rule. We then take A has **step4a**$(P, N_a, \mathcal{T}_j, N_p)$ and A has **msg**(**plain**$(N_p), \mathcal{T}_k)$ as the right hand propositions for the rule and invert it to arrive at the state Γ_0, A knows **PublicKey**(K_P^+, P), P knows **PublicKey**(K_A^+, A); S has **needkey**(P), S has **needkey**(A), P has **step3e**$(A, N_a, N_p, \mathcal{T}_i)$, A has **step1e**$(P, N_a, \mathcal{T}_j)$, A has **unpack**$(encPub(A), A, \mathcal{T}_l)$, A has **msg**(**plain**$(N_a \sim N_p \sim P), \mathcal{T}_l)$. We see that we must have arrived at this state by the rule **UnpackDecryptPublic**.

We take A has **unpack**$(encPub(A), A, \mathcal{T}_l)$ and A has **msg**(**plain**$(N_a \sim N_p \sim P), \mathcal{T}_l)$ as the right hand propositions and invert the rule, arriving at the state Γ_0, A knows **PublicKey**(K_P^+, P), P knows **PublicKey**(K_A^+, A); S has **needkey**(P), S has **needkey**(A), P has **step3e**$(A, N_a, N_p, \mathcal{T}_i)$, A has **step1e**$(P, N_a, \mathcal{T}_j)$, A has **recv**(**encPub**$(H, A), A)$. This state must have immediately followed an application of **ReceiveMessage**.

Taking A has **recv**(**encPub**$(H, A), A)$ to be the right hand proposition, we again face the question of whether the rule premise was $\langle A \rangle$**send**(**encPub**$(H, A), A)$, $\langle P \rangle$**send**(**encPub**$(H, A), A)$, or $\langle S \rangle$**send**(**encPub**$(H, A), A)$. Ths time, picking either the first or third last leads us to a dead end, so we must take the second when inverting the rule. Doing so leads us to the state Γ_0, A knows **PublicKey**(K_P^+, P), P knows **PublicKey**(K_A^+, A); S has **needkey**(P), S has **needkey**(A), P has **step3e**$(A, N_a, N_p, \mathcal{T}_i)$, A has **step1e**$(P, N_a, \mathcal{T}_j)$, $\langle P \rangle$**send**(**encPub**$(H, A), A)$. The preced-

ing rule in the derivation must have been **SendOvertureResponse**.

For the rule inversion, we take $\langle P \rangle$**send**(**encPub**$(H, A), A)$ and P has **step3e**$(A, N_a, N_p, \mathcal{T}_i)$ to be the right hand propositions. Inverting takes us to the state Γ_0, A knows **PublicKey**(K_P^+, P), P knows **PublicKey**$(K_A^+, A); S$ has **needkey**$(P), S$ has **needkey**$(A), A$ has **step1e**$(P, N_a, \mathcal{T}_j)$, P has **step3d**$(A, N_a, N_p, \mathcal{T}_i)$, P has **msg**(**encPub**$(H, A), \mathcal{T}_i)$. It must have been that **EncryptOvertureResponse** was applied to derive this state.

Taking P has **step3d**$(A, N_a, N_p, \mathcal{T}_i)$ and P has **msg**(**encPub**$(H, A), \mathcal{T}_i)$ to be the right hand propositions, we invert the rule to arrive at the state Γ_0, A knows **PublicKey**(K_P^+, P), P knows **PublicKey**$(K_A^+, A); S$ has **needkey**$(P), S$ has **needkey**$(A), A$ has **step1e**$(P, N_a, \mathcal{T}_j)$, P has **step3c**$(A, N_a, N_p, \mathcal{T}_i)$, P has **msg**$(M, \mathcal{T}_i)$. Again, while the rule doesn't explicitly indicate how to interpret M in P has **msg**$(M, \mathcal{T}_i)$, we can see by the matching tags, $\mathcal{T}_i$, in that proposition and in P's state marker that M must take the form **plain**$(N_a \sim N_p \sim P)$ and that the rule used to derive this state must have been **CreateOvertureResponse**.

We can take P has **msg**$(M, \mathcal{T}_i)$ and P has **step3c**$(A, N_a, N_p, \mathcal{T}_i)$ to be the right hand propositions of **CreateOvertureResponse** and invert the rule. Doing so leads us to the state Γ_0, A knows **PublicKey**$(K_P^+, P), P$ knows **PublicKey**$(K_A^+, A); S$ has **needkey**$(P), S$ has **needkey**(A), A has **step1e**$(P, N_a, \mathcal{T}_j)$, P has **step3b**(A, N_a, N_p), P has **msg**(**plain**$(N_a), \mathcal{T}_m)$. This state must have been derived from an application of **GenNonce2**.

We invert the rule, taking P has **step3b**(A, N_a, N_p) to be the right hand proposition of **GenNonce2** and arrive at the state Γ_0, A knows **PublicKey**$(K_P^+, P), P$ knows **PublicKey**(K_A^+, A); S has **needkey**$(P), S$ has **needkey**$(A), A$ has **step1e**$(P, N_a, \mathcal{T}_j)$, P has **step3a**(A, N_a), P has **msg**(**plain**$(N_a), \mathcal{T}_m)$.

So far, we have shown that all state transition sequences $\Gamma_0; \Delta_0 \xrightarrow{\epsilon'} \Gamma_0, A$ knows **PublicKey**(K_P^+, P), P knows **PublicKey**$(K_A^+, A); S$ has **needkey**$(P), S$ has **needkey**$(A), A$ has **sessionApp**(P, N_a, N_p), P has **sessionProv**(A, N_a, N_p) must end according to the perviously shown form, preceded by a state transition sequence of the form Γ_5, A knows **PublicKey**$(K_P^+, P), P$ knows **PublicKey**$(K_A^+, A); \Delta_5$,

A has **step1e**$(P, N_a, \mathcal{T}_i)$, P has **step3a**$(A, N_a) \xrightarrow{\epsilon_f} \Gamma_6$, A knows **PublicKey**(K_P^+, P), P knows **PublicKey**(K_A^+, A); Δ_6, P has **step3e**$(A, N_a, N_b, \mathcal{T}_j)$, A has **step4a**$(P, N_a, \mathcal{T}_i, N_b)$

Returning to the derivation, we see at this point that the state must have been derived from applying **KeyKnown2**. Taking P has **step3a**(A, N_a) to be the right hand proposition, we invert the rule to arrive at the state Γ_0, A knows **PublicKey**(K_P^+, P), P knows **PublicKey**(K_A^+, A); S has **needkey**(P), S has **needkey**(A), A has **step1e**$(P, N_a, \mathcal{T}_j)$, P has **step2a**(A, N_a), P has **msg**(**plain**$(N_a), \mathcal{T}_m)$. Note how it may seem at this point that we are stuck. It might seem tempting to guess that **RealizeProvokee** would be the most recently applied rule, but we are missing the critical P has **needkey**(A).

Notice that our terminal persistent context contains two pieces of knowledge that aren't in Γ_0, A knows **PublicKey**(K_P^+, P) and P knows **PublicKey**(K_A^+, A). It must be the case, then, that this knowledge was derived somewhere. Since no rules with purely linear consequents can apply here, it must be the case that a rule with a persistent consequent was applied. The only rule with a persistent consequent is **LearnKey**. So if this derivation is to exist, it must be the case that **LearnKey** was applied. It is now a question of whether A knows **PublicKey**(K_P^+, P) or P knows **PublicKey**(K_A^+, A) is to be taken as the right hand proposition. Taking the former leads us to another dead end, so we continue the derivation with the latter as the right hand proposition.

Doing so, we invert **LearnKey** to arrive at the state Γ_0, A knows **PublicKey**(K_P^+, P); S has **needkey**(P), S has **needkey**(A), A has **step1e**$(P, N_a, \mathcal{T}_j)$, P has **step2a**(A, N_a), P has **msg**(**plain**$(N_a), \mathcal{T}_m)$, P has **unpack**$(encPri(S), P, \mathcal{T}_n)$, P has **msg**(**plain**$(\kappa(K_A^+, A)), \mathcal{T}_n)$. Here, we see that the rule applied to derive this state must have been **UnpackDecryptPrivate**.

We invert the rule, taking P has **unpack**$(encPri(S), P, \mathcal{T}_n)$ and P has **msg**(**plain**$(\kappa(K_A^+, A)), \mathcal{T}_n)$ to be the right hand propositions, and we are led to the state Doing so, we invert **LearnKey** to arrive at the state Γ_0, A knows **PublicKey**(K_P^+, P); S has **needkey**(P), S has **needkey**(A), A has **step1e**$(P, N_a, \mathcal{T}_j)$, P has **step2a**(A, N_a), P has **msg**(**plain**$(N_a), \mathcal{T}_m)$, P has **recv**(**encPri**$(H, S), P)$. We see here that this state have been derived by applying **ReceiveMessage**.

We can take P has **recv**(**encPri**$(H, S), P)$ to be the right hand proposition of the rule, but we are again faced with the choice of whether the left hand proposition is $\langle A \rangle$**send**(**encPri**$(H, S), P)$, $\langle P \rangle$**send**(**encPri**$(H, S), P)$, or $\langle S \rangle$**send**(**encPri**$(H, S), P)$. Selecting either of the former two leads us to dead ends, so we continue with the lattermost as the left hand derivation. Doing so, we arrive at the state Γ_0, A knows **PublicKey**$(K_P^+, P); S$ has **needkey**$(P), S$ has **needkey**(A), A has **step1e**$(P, N_a, \mathcal{T}_j), P$ has **step2a**$(A, N_a), P$ has **msg**(**plain**$(N_a), \mathcal{T}_m)$, $\langle S \rangle$**send**(**encPri**$(H, S), P)$. The only rule which could have derived this state is **SendKey**.

Taking $\langle S \rangle$**send**(**encPri**$(H, S), P)$ as the right hand proposition, we invert the rule to arrive at the state Γ_0, A knows **PublicKey**$(K_P^+, P); S$ has **needkey**$(P), S$ has **needkey**(A), A has **step1e**$(P, N_a, \mathcal{T}_j), P$ has **step2a**$(A, N_a), P$ has **msg**(**plain**$(N_a), \mathcal{T}_m)$, S has **msg**(**encPri**$(H, S), \mathcal{T}_o)$, S has **step0c**$(A, P, \mathcal{T}_o)$. It must have been an application of **EncryptKey** which derived this state.

We take S has **step0c**$(A, P, \mathcal{T}_o)$ and S has **msg**(**encPri**$(H, S), \mathcal{T}_o)$ as our right hand propositions and invert the **EncryptKey** rule to arrive at the state Γ_0, A knows **PublicKey**(K_P^+, P); S has **needkey**$(P), S$ has **needkey**$(A), A$ has **step1e**$(P, N_a, \mathcal{T}_j), P$ has **step2a**(A, N_a), P has **msg**(**plain**$(N_a), \mathcal{T}_m)$, S has **msg**$(M, \mathcal{T}_o)$, S has **step0b**$(A, P, \mathcal{T}_o)$. Again, we do not have explicit information as to what form M takes, but because of the matching tags between S's message and S's state marker, we can see that **ConstructKeyMessage** would be the only possible rule to invert on, forcing M to be **plain**$(\kappa(K_A^+, A))$.

We invert the rule, taking S has **step0b**$(A, P, \mathcal{T}_o)$ and S has **msg**(**plain**$(\kappa(K_A^+, A)), \mathcal{T}_o)$ to be the right hand propositions of **ConstructKeymessage**. Doing so leads us to the state Γ_0, A knows **PublicKey**$(K_P^+, P); S$ has **needkey**$(P), S$ has **needkey**$(A), A$ has **step1e**$(P, N_a, \mathcal{T}_j)$, P has **step2a**$(A, N_a), P$ has **msg**(**plain**$(N_a), \mathcal{T}_m)$, S has **step0a**(A, P).

So far, we have shown that all state transition sequences $\Gamma_0; \Delta_0 \xrightarrow{\epsilon'} \Gamma_0, A$ knows **PublicKey**(K_P^+, P), P knows **PublicKey**$(K_A^+, A); S$ has **needkey**$(P), S$ has **needkey**$(A), A$ has **sessionApp**(P, N_a, N_p), P has **sessionProv**(A, N_a, N_p) must end according to the previously shown form, preceded by a

state transition sequence of the form Γ_4, A knows **PublicKey**(K_P^+, P); Δ_4, A has **step1e**$(P, N_a, \mathcal{T}_i)$, P has **step2a**(A, N_a), S has **step0a**$(A, P) \xrightarrow{\epsilon_e} \Gamma_5, A$ knows **PublicKey**(K_P^+, P), P knows **PublicKey**(K_A^+, A); Δ_5, A has **step1e**$(P, N_a, \mathcal{T}_i)$, P has **step3a**(A, N_a)

Resuming with the derivation at hand, we see at this point that the rule that must have been applied to derive this state is **GetKeyRequest**. We take S has **step0a**(A, P) and S has **needkey**(A) to be our right hand propositions and invert the rule to arrive at the state Γ_0, A knows **PublicKey**(K_P^+, P); S has **needkey**(P), A has **step1e**$(P, N_a, \mathcal{T}_j)$, P has **step2a**(A, N_a), P has **msg**(**plain**$(N_a), \mathcal{T}_m)$, S has **unpack**$(plain, S, \mathcal{T}_p)$, S has **msg**(**plain**(**getkey**$(A, P)), \mathcal{T}_p)$. It must be the case that we last applied **UnpackPlain** to arrive at this state.

Taking S has **unpack**$(plain, P, \mathcal{T}_p)$ and S has **msg**(**plain**(**getkey**$(P, A)), \mathcal{T}_p)$ to be the right hand propositions, we invert the rule, leading us to the state Γ_0, A knows **PublicKey**(K_P^+, P); S has **needkey**(P), A has **step1e**$(P, N_a, \mathcal{T}_j)$, P has **step2a**(A, N_a), P has **msg**(**plain**$(N_a), \mathcal{T}_m)$, S has **recv**(**plain**(**getkey**$(A, P)), S)$. This state must have been derived immediately following an application of **ReceiveMessage**.

We want to invert the rule, so we take S has **recv**(**plain**(**getkey**$(P, A)), P)$ to be the right hand proposition. Again, though, we find ourselves having to determine whether the left hand proposition is $\langle S\rangle$**send**(**plain**(**getkey**$(A, P)), S)$, $\langle A\rangle$**send**(**plain**(**getkey**$(A, P)), S)$, or $\langle P\rangle$**send**(**plain**(**getkey**$(A, P)), S)$. Selecting either of the first two leads us to a dead end, so we take $\langle P\rangle$**send**(**plain**(**getkey**$(A, P)), S)$ to be our left hand proposition and arrive at the state Γ_0, A knows **PublicKey**(K_P^+, P); S has **needkey**(P), A has **step1e**$(P, N_a, \mathcal{T}_j)$, P has **step2a**(A, N_a), P has **msg**(**plain**$(N_a), \mathcal{T}_m)$, $\langle P\rangle$**send**(**plain**(**getkey**$(A, P)), S)$. We see here that the only rule which could have been applied to derive this state is **RequestKey**.

We take $\langle P\rangle$**send**(**plain**(**getkey**$(A, P)), S)$ as our right hand proposition and invert the **RequestKey** rule to arrive at the state Γ_0, A knows **PublicKey**(K_P^+, P); S has **needkey**(P), A has **step1e**$(P, N_a, \mathcal{T}_j)$, P has **step2a**(A, N_a), P has **msg**(**plain**$(N_a), \mathcal{T}_m)$, P has **needkey**(A).

At this point, we have shown that all state transition sequences $\Gamma_0; \Delta_0 \xrightarrow{\epsilon'} \Gamma_0$, A knows **PublicKey**(K_P^+, P), P knows **PublicKey**(K_A^+, A); S has **needkey**(P), S has **needkey**(A), A has **sessionApp**(P, N_a, N_p), P has **sessionProv**(A, N_a, N_p) must end according to the previously shown form, preceded by a state transition sequence of the form Γ_3, A knows **PublicKey**(K_P^+, P); Δ_3, A has **step1e**$(P, N_a, \mathcal{T}_i)$, P has **step2a**$(A, N_a) \xrightarrow{\epsilon_d} \Gamma_4$, A knows **PublicKey**(K_P^+, P); Δ_4, A has **step1e**$(P, N_a, \mathcal{T}_i)$, P has **step2a**(A, N_a), S has **step0a**(A, P)

Returning our attention to the derivation, we see that, at this point, the only rule which could have derived this state is **RealizeProvokee**. We invert the rule, taking P has **needkey**(A), P has **msg**(**plain**$(N_a), \mathcal{T}_m)$, and P has **step2a**(A, N_a) as our right hand propositions, and arrive at the state Γ_0, A knows **PublicKey**(K_P^+, P); S has **needkey**(P), A has **step1e**$(P, N_a, \mathcal{T}_j)$, P has **unpack**$(encPub(P), P, \mathcal{T}_m)$, P has **msg**(**plain**$(N_a \sim A), \mathcal{T}_m)$. Here, the only rule application which could have derived the current state is **UnpackDecryptPublic**.

Taking P has **unpack**$(encPub(P), P, \mathcal{T}_m)$ and P has **msg**(**plain**$(N_a \sim A), \mathcal{T}_m)$ as our right hand propositions, we invert the rule to arrive at the state Γ_0, A knows **PublicKey**(K_P^+, P); S has **needkey**(P), A has **step1e**$(P, N_a, \mathcal{T}_j)$, P has **recv**(**encPub**$(H, P), P)$. This state can only have been derived from an application of the rule **ReceiveMessage**.

We want to invert the rule, taking P has **recv**(**encPub**$(H, P), P)$ to be the right hand proposition. As with every time we invert on **ReceiveMessage**, we must determine whether the left hand proposition is $\langle A \rangle$**send**(**encPub**$(H, P), P)$, $\langle S \rangle$**send**(**encPub**$(H, P), P)$, or $\langle P \rangle$**send**(**encPub**$(H, P), P)$. Taking either of the latter two results in a dead end, so we take the first. Upon inverting the rule, we arrive at the state Γ_0, A knows **PublicKey**(K_P^+, P); S has **needkey**(P), A has **step1e**$(P, N_a, \mathcal{T}_j)$, $\langle A \rangle$**send**(**encPub**$(H, P), P)$. To derive this state, it must be the case that **SendOverture** was the rule applied.

We take $\langle A \rangle$**send**(**encPub**$(H, P), P)$ and A has **step1e**$(P, N_a, \mathcal{T}_j)$ to be the right hand propositions and invert the rule, arrivig at the state Γ_0, A knows **PublicKey**(K_P^+, P); S has **needkey**(P),

A has **step1d**$(P, N_a, \mathcal{T}_j)$, A has **msg**(**encPub**$(H, P), \mathcal{T}_j)$. Now to derive at this state, it must have been **EncryptOverture** which was most recently applied.

We invert the rule, taking A has **step1d**$(P, N_a, \mathcal{T}_j)$ and A has **msg**(**encPub**$(H, P), \mathcal{T}_j)$ as right hand propositions, and arrive at the state Γ_0, A knows **PublicKey**$(K_P^+, P); S$ has **needkey**(P), A has **step1c**$(P, N_a, \mathcal{T}_j)$, A has **msg**(**plain**$(M), \mathcal{T}_j)$. Although this rule gives us no information as to what form M takes, we can see that, given the matching tags of the two new propositions in this state, the rule which must have been applied to derive this state must have been **CreateOverture**. If we take these two propositions as the right hand propositions, we force M to be $N_a \sim A$.

Taking A has **step1c**$(P, N_a, \mathcal{T}_j)$ and A has **msg**(**plain**$(N_a \sim A), \mathcal{T}_j)$, we invert **CreateOverture** to arrive at the state Γ_0, A knows **PublicKey**$(K_P^+, P); S$ has **needkey**(P), A has **step1b**(P, N_a). To derive this state, it must be the case that **GenNonce1** was the most recently applied rule.

We take A has **step1b**(P, N_a) to be the right hand proposition and invert the rule to arrive at the state Γ_0, A knows **PublicKey**$(K_P^+, P); S$ has **needkey**(P), A has **step1a**(P).

So far, we have shown that all state transition sequences $\Gamma_0; \Delta_0 \xrightarrow{\epsilon'} \Gamma_0, A$ knows **PublicKey**(K_P^+, P), P knows **PublicKey**$(K_A^+, A); S$ has **needkey**$(P), S$ has **needkey**$(A), A$ has **sessionApp**(P, N_a, N_p), P has **sessionProv**(A, N_a, N_p) must end according to the previously shown form, preceded by a state transition sequence of the form Γ_2, A knows **PublicKey**$(K_P^+, P); \Delta_2, A$ has **step1a**$(P) \xrightarrow{\epsilon_c}$ Γ_3, A knows **PublicKey**$(K_P^+, P); \Delta_3, A$ has **step1e**$(P, N_a, \mathcal{T}_i), P$ has **step2a**(A, N_a)

Resuming with the derivation at hand, we see that it must have been **KeyKnown1** which was just applied to derive this state. We invert the rule, taking A has **step1a**(P) as our right hand consequent, and arrive at the state Γ_0, A knows **PublicKey**$(K_P^+, P); S$ has **needkey**(P), A has **init**(P). At this point there are no more rules with linear consequents which can apply. Therefore, it must have been the one rule which has persistent consequents, **LearnKey**, which was applied to derive at this state.

We invert **LearnKey**, taking A knows **PublicKey**(K_P^+, P) to be the right hand proposition, and arrive at the state $\Gamma_0; S$ has **needkey**$(P), A$ has **init**(P), A has **unpack**$(encPri(S), A, \mathcal{T}_q)$, A has **msg**(**plain**$(\kappa(K_P^+, P))\mathcal{T}_q$,). To derive this state, it must be the case that **UnpackDecrypt-Private** was most recently applied.

Taking A has **unpack**$(encPri(S), A, \mathcal{T}_q)$ and A has **msg**(**plain**$(\kappa(K_P^+, P)), \mathcal{T}_q)$ to be our right hand propositions, we invert the rule and arrive at the state $\Gamma_0; S$ has **needkey**$(P), A$ has **init**(P), A has **recv**(**encPri**$(H, S), A)$. This state must have been derived from most recently applying the rule **ReceiveMessage**.

We want to invert the rule, taking A has **recv**(**encPri**$(H, S), A)$ as our right hand proposition. Once more, we are faced with determining whether the left hand proposition will be $\langle A\rangle$**send**(**encPri**$(H, S), A)$, $\langle S\rangle$**send**(**encPri**$(H, S), A)$, or $\langle P\rangle$**send**(**encPri**$(H, S), A)$. If we choose either the first or the third, we end up at a dead end, so we take $\langle S\rangle$**send**(**encPri**$(H, S), A)$ to be the left hand proposition and arrive at the state $\Gamma_0; S$ has **needkey**$(P), A$ has **init**(P), $\langle S\rangle$**send**(**encPri**$(H, S), A)$. Here, we see that the only rule which could have derived this state is **SendKey**.

We take $\langle S\rangle$**send**(**encPri**$(H, S), A)$ to be our right hand proposition and invert the rule, arriving at the state $\Gamma_0; S$ has **needkey**$(P), A$ has **init**(P), S has **step0c**$(P, A, \mathcal{T}_r)$, S has **msg**$(M, \mathcal{T}_r)$. Although we do not have explicit information from this inversion as to what M is, we see by the overall structure of the propositions and the matching tags that the only rule which could have been applied to derive this state is **EncryptKey**. Because of this rule application, M is forced to be **encPri**(H, S).

We then invert on **EncryptKey**, taking S has **step0c**$(P, A, \mathcal{T}_r)$ and S has **msg**(**encPri**$(H, S), \mathcal{T}_r)$ to be our right hand propositions, and arrive at the state $\Gamma_0; S$ has **needkey**$(P), A$ has **init**(P), S has **step0b**$(P, A, \mathcal{T}_r)$, S has **msg**$(M, \mathcal{T}_r)$. Again, we have no information from this rule inversion as to what the form of M is, but given the proposition forms and the matching tags, we see that **ConstructKeyMessage** must have been the most recently applied rule to derive this state. As such, M must take the form **plain**$(\kappa(K_P^+, P))$.

Taking S has **step0b**$(P, A, \mathcal{T}_r)$ and S has **msg**(**plain**$(\kappa(K_P^+, P)), \mathcal{T}_r)$ to be the right hand propositions, we invert **ConstructKeyMessage** to arrive at the state Γ_0; S has **needkey**(P), A has **init**(P), S has **step0a**(P, A).

At this point, we have shown that all state transition sequences $\Gamma_0; \Delta_0 \xrightarrow{\epsilon'} \Gamma_0$, A knows **PublicKey**(K_P^+, P), P knows **PublicKey**(K_A^+, A); S has **needkey**(P), S has **needkey**(A), A has **sessionApp**(P, N_a, N_p), P has **sessionProv**(A, N_a, N_p) must end according to the previously shown form, preceded by a state transition sequence of the form $\Gamma_1; \Delta_1, S$ has **step0a**(P, A) $\xrightarrow{\epsilon_b} \Gamma_2, A$ knows **PublicKey**(K_P^+, P); Δ_2, A has **step1a**(P)

Returning to the derivation, we see that, from here, the only rule which could be applied to derive the current state is **GetKeyRequest**. We take S has **step0a**(P, A) and S has **needkey**(P) to be the right hand propositions, we invert the **GetKeyRequest** rule and arrive at the state $\Gamma_0; A$ has **init**(P), S has **unpack**$(plain, A, \mathcal{T}_s)$, S has **msg**(**plain**(**getkey**$(P, A)), \mathcal{T}_s)$. We see that this state can ony have been derived from the rule **UnpackPlain**.

Taking S has **unpack**$(plain, A, \mathcal{T}_s)$ and S has **msg**(**plain**(**getkey**$(P, A)), \mathcal{T}_s)$ to be the right hand propositions, we invert the **UnpackPlain** rule and arrive at the state $\Gamma_0; A$ has **init**(P), S has **recv**(**plain**(**getkey**$(P, A)), S)$. The only rule which could have been applied to derive this state, we see, is **ReceiveMessage**.

We want to invert the rule, taking S has **recv**(**plain**(**getkey**$(P, A)), S)$ to be the right hand proposition, but it is ambiguous whether the left hand proposition is $\langle A \rangle$**send**(**plain**(**getkey**$(P, A)), S)$, $\langle S \rangle$**send**(**plain**(**getkey**$(P, A)), S)$, or $\langle P \rangle$**send**(**plain**(**getkey**$(P, A)), S)$. But the only rule which generates a proposition of the form $\langle A \rangle$**send**(**plain**(**getkey**$(B, C)), D)$ is **RequestKey**, and in that rule, A and C always match. So it must be the case that $\langle A \rangle$**send**(**plain**(**getkey**$(P, A)), S)$ is the correct left hand proposition. Inverting the rule, we arrive at the state $\Gamma_0; A$ has **init**(P), $\langle A \rangle$**send**(**plain**(**getkey**$(P, A)), S)$. The only rule which could have been applied to derive this state, we see, is **RequestKey**.

We take $\langle A \rangle \textbf{send}(\textbf{plain}(\textbf{getkey}(P, A)), S)$ to be the right hand proposition and invert the rule to arrive at the state $\Gamma_0; A \text{ has } \textbf{init}(P), A \text{ has } \textbf{needkey}(B)$, which we recognize as $\Gamma_0; \Delta_0$.

We have shown that for all state transition sequence derivations $\Gamma_0; \Delta_0 \xrightarrow{\epsilon'} \Gamma_0$, $A \text{ knows } \textbf{PublicKey}(K_P^+, P), P \text{ knows } \textbf{PublicKey}(K_A^+, A); S \text{ has } \textbf{needkey}(P), S \text{ has } \textbf{needkey}(A)$, $A \text{ has } \textbf{sessionApp}(P, N_a, N_p), P \text{ has } \textbf{sessionProv}(A, N_a, N_p)$, it must be the case that ϵ' follows the form of ϵ. $\square$

3.6 Summary

Having shown both the soundness and completeness of the formalism with respect to the protocol, we conclude that our formalism is an adequate representation of the Needham-Schröder-Lowe public key authentication protocol.

Chapter 4

Conclusions

4.1 Comparison With Prior Work

We have given a formalism for the Needham-Schröder-Lowe protocol in linear epistemic logic which we have proven to adequately capture the semantics of the protocol. Our formalism is interesting in that it provides an explicit account of the participants' knowledge states as they go through the protocol.

Previously presented formalisms, such as that of Durgin et al.[3], have different approaches which are computationally interesting, but they focus solely on the actions of the participants. As the MSR formalism can be interpreted as a variation of linear logic, it might be possible to extend that formalism epistemically to more closely resemble our approach. Unfortunately, as there has been no adequacy theorem provided for the MSR formalism, we cannot know exactly how our approach compares in quality.

4.2 Future Work

The Needham-Schröder example in [3] comes with a formalism of the intruder attack discovered by Lowe in [5]. We do not provide such a model, but it would be interesting to see how the

intruder attack would be interpreted in terms of the participants' knowledge states. In particular, the attack in question is impossible when Lowe's fix is applied to the Needham-Schröder protocol.

A formalism for the Needham-Schröder protocol in linear epistemic logic would involve modifying only two of the rules in the Needham-Schröder-Lowe formalism we provided: **CreateOvertureResponse** and **RealizeProvokeeResponded**. Running an intruder attack in both versions of the formalism would show more explicitly how an individual principal may or may not be fooled by the intruder's fradulent messages.

Beyond the scope of this thesis, though, we believe that the expressiveness that linear epistemic logic brings to a protocol specification could be beneficial for precisely specifying subtle action sequences which could be misinterpreted or left vulnerable to attacks. We intend to develop on this idea and seek out venues that would be receptive to the work, such as the IEEE Computer Security Foundations Symposium or the European Association for Computer Science Logic Conference.

4.3 Summary of Contributions

In presenting this work, we have made the following contributions:

- The Needham-Schröder-Lowe public key authentication protocol can be adequately formalized using linear epistemic logic.
- Using linear epistemic logic as the protocol language allows us to present the formalism at the same level of abstraction as the protocol itself.
- Expressing a protocol in linear epistemic logic makes the semantics precise in terms of what triggers individual participants' action sequences.

Appendix A

Rewrite Rules

We present the rewrite rules for each of our formalism's operations defined in Section 2.2.3.

RequestKey:

Γ, A knows **Server**$(B); \Delta, A$ has **needkey**(C), !**AgentNeq**$(B, C) \longrightarrow \Gamma, A$ knows **Server**$(B); \Delta,$ $\langle A \rangle$**send**(**plain**(**getkey**$(B, A)), C)$

GetKeyRequest:

Γ, A knows **Server**$(A); \Delta, A$ has **unpack**$(plain, A, T), A$ has **msg**(**plain**(**getkey**$(B, C)), T) \longrightarrow$ Γ, A knows **Server**$(A); \Delta, A$ has **step0a**$(B, C), A$ has **needkey**(B)

ConstructKeyMessage:

Γ, A knows **PublicKey**$(K_B^+, B); \Delta, A$ has **step0a**$(B, C) \longrightarrow \Gamma, A$ knows **PublicKey**$(K_B^+, B); \Delta,$ A has **step0b**$(B, C, \mathcal{T}_i), A$ has **msg**(**plain**$(\kappa(K_B^+, B)), \mathcal{T}_i)$, with $\mathcal{T}_i$ being fresh

EncryptKey:

Γ, A knows **PrivateKey**$(K_A^-, A); \Delta, A$ has **step0b**$(B, C, \mathcal{T}_i), M$ has $\mathcal{T}_i$, !**Crypt**$(H, K_A^-, M) \longrightarrow$ Γ, A knows **PrivateKey**$(K_A^-, A); \Delta, A$ has **step0c**$(B, C, \mathcal{T}_i), A$ has **msg**(**encPri**$(H, A), \mathcal{T}_i)$

SendKey:

$\Gamma; \Delta, A \text{ has } \mathbf{step0c}(B, C, \mathcal{T}_i), A \text{ has } \mathbf{msg}(M, \mathcal{T}_i) \longrightarrow \Gamma; \Delta, \langle A \rangle \mathbf{send}(M, C)$

LearnKey:

$\Gamma, A \text{ knows } \mathbf{Server}(B); \Delta, A \text{ has } \mathbf{unpack}(encPri(B), A, \mathcal{T}_i), A \text{ has } \mathbf{msg}(\mathbf{plain}(\kappa(K_B^+, B)), \mathcal{T}_i) \longrightarrow$
$\Gamma, A \text{ knows } \mathbf{Server}(B), A \text{ knows } \mathbf{PublicKey}(K_B^+, B); \Delta$

KeyKnown1:

$\Gamma, A \text{ knows } \mathbf{PublicKey}(K_B^+, B); \Delta, A \text{ has } \mathbf{init}(B) \longrightarrow \Gamma, A \text{ knows } \mathbf{PublicKey}(K_B^+, B); \Delta,$
$A \text{ has } \mathbf{step1a}(B)$

GenNonce1:

$\Gamma; \Delta, A \text{ has } \mathbf{step1a}(B) \longrightarrow \Gamma; \Delta, A \text{ has } \mathbf{step1b}(B, N_a)$, where N_a is fresh

CreateOverture:

$\Gamma; \Delta, A \text{ has } \mathbf{step1b}(B, N_a) \longrightarrow \Gamma; \Delta, A \text{ has } \mathbf{step1c}(B, N_a, \mathcal{T}_i), A \text{ has } \mathbf{msg}(\mathbf{plain}(N_a \sim A), \mathcal{T}_i)$

EncryptOverture:

$\Gamma, A \text{ knows } \mathbf{PublicKey}(K_B^+, B); \Delta, A \text{ has } \mathbf{step1c}(B, N_a, \mathcal{T}_i), A \text{ has } \mathbf{msg}(\mathbf{plain}(M), \mathcal{T}_i),$
$!\mathbf{Crypt}(H, K_B^+, \mathbf{plain}(M)) \longrightarrow \Gamma, A \text{ knows } \mathbf{PublicKey}(K_B^+, B); \Delta, A \text{ has } \mathbf{step1d}(B, N_a, \mathcal{T}_i),$
$A \text{ has } \mathbf{msg}(\mathbf{encPub}(H, B), \mathcal{T}_i)$

SendOverture:

$\Gamma; \Delta, A \text{ has } \mathbf{step1d}(B, N_a, \mathcal{T}_i), A \text{ has } \mathbf{msg}(M, \mathcal{T}_i) \longrightarrow \Gamma; \Delta, A \text{ has } \mathbf{step1e}(B, N_a, \mathcal{T}_i), \langle A \rangle \mathbf{send}(M, B)$

ReceiveMessage:

$\Gamma; \Delta, \langle A \rangle$**send**$(M, B) \longrightarrow \Gamma; \Delta, C$ has **recv**(M, B)

UnpackPlain:

$\Gamma; \Delta, A$ has **recv**(**plain**$(M), B) \longrightarrow \Gamma; \Delta, A$ has **unpack**$(plain, B, \mathcal{T}_i), A$ has **msg**$(M, \mathcal{T}_i)$, with $\mathcal{T}_i$ fresh

UnpackDecryptPublic:

Γ, A knows **PrivateKey**$(K_B^-, B); \Delta, A$ has **recv**(**encPub**$(H, B), C)$, !**Crypt**$(H, K_B^-, M) \longrightarrow \Gamma$, A knows **PrivateKey**$(K_B^-, B); \Delta, A$ has **unpack**$(encPub(B), C, \mathcal{T}_i), A$ has **msg**$(M, \mathcal{T}_i)$, where $\mathcal{T}_i$ is fresh

UnpackDecryptPrivate:

Γ, A knows **PublicKey**$(K_B^+, B); \Delta, A$ has **recv**(**encPri**$(H, B), C)$, !**Crypt**$(H, K_B^+, M) \longrightarrow \Gamma$, A knows **PublicKey**$(K_B^+, B); \Delta, A$ has **unpack**$(encPri(B), C, \mathcal{T}_i), A$ has **msg**$(M, \mathcal{T}_i)$, where $\mathcal{T}_i$ is fresh

RealizeProvokee:

$\Gamma; \Delta, A$ has **unpack**$(encPub(A), A, \mathcal{T}_i), A$ has **msg**(**plain**$(N_b \sim B), \mathcal{T}_i) \longrightarrow \Gamma; \Delta, A$ has **step2a**(B, N_b), A has **msg**(**plain**$(N_b), \mathcal{T}_i), A$ has **needkey**(B)

KeyKnown2:

ΓA knows **PublicKey**$(K_B^+, B); \Delta, A$ has **step2a**$(B, N_b) \longrightarrow \Gamma A$ knows **PublicKey**$(K_B^+, B); \Delta$, A has **step3a**(B, N_b)

GenNonce2:

$\Gamma; \Delta, A$ has **step3a**$(B, N_b) \longrightarrow \Gamma; \Delta, A$ has **step3b**(B, N_b, N_a), where N_a is fresh

CreateOvertureResponse:

$\Gamma; \Delta, A$ has **step3b**$(B, N_b, N_a), A$ has **msg**(**plain**$(N_b), \mathcal{T}_i) \longrightarrow \Gamma; \Delta, A$ has **step3c**$(B, N_b, N_a, \mathcal{T}_j)$, A has **msg**(**plain**$(N_b \sim N_a \sim A), \mathcal{T}_j)$

EncryptOvertureResponse:

Γ, A knows **PublicKey**$(K_B^+, B); \Delta, A$ has **step3c**$(B, N_b, N_a, \mathcal{T}_i), A$ has **msg**$(M, \mathcal{T}_i)$, !**Crypt**(H, K_B^+, M) $\longrightarrow \Gamma, A$ knows **PublicKey**$(K_B^+, B); \Delta, A$ has **step3d**$(B, N_b, N_a, \mathcal{T}_i), A$ has **msg**(**encPub**$(H, B), \mathcal{T}_i)$

SendOvertureResponse:

$\Gamma; \Delta, A$ has **step3d**$(B, N_b, N_a, \mathcal{T}_i), A$ has **msg**$(M, \mathcal{T}_i) \longrightarrow \Gamma; \Delta, A$ has **step3e**$(B, N_b, N_a, \mathcal{T}_i)$, $\langle A \rangle$**send**(M, B)

RealizeProvokeeResponded:

$\Gamma; \Delta, A$ has **step1e**$(B, N_a, \mathcal{T}_i), A$ has **unpack**$(encPub(A), A, \mathcal{T}_j), A$ has **msg**$(N_a \sim N_b \sim B, \mathcal{T}_j) \longrightarrow$ $\Gamma; \Delta, A$ has **step4a**$(B, N_a, \mathcal{T}_i, N_b), A$ has **msg**(**plain**$(N_b), \mathcal{T}_k)$, where $\mathcal{T}_k$ is fresh

EncryptProvokeeNonce:

Γ, A knows **PublicKey**$(K_B^+, B); \Delta, A$ has **step4a**$(B, N_a, \mathcal{T}_i, N_b), A$ has **msg**(**plain**$(N_b), \mathcal{T}_j)$, !**Crypt**$(H, K_B^+,$ **plain**$(N_b)) \longrightarrow \Gamma; \Delta, A$ has **step4b**$(B, N_a, \mathcal{T}_i, N_b, \mathcal{T}_j), A$ has **msg**(**encPub**$(H, B), \mathcal{T}_j)$

SecureSessionApproacher:

$\Gamma; \Delta A$ has **step4b**$(B, N_a, \mathcal{T}_i, N_b, \mathcal{T}_j), A$ has **msg**$(M, \mathcal{T}_j) \longrightarrow \Gamma; \Delta, A$ has **sessionApp**(B, N_a, N_b), $\langle A \rangle$**send**(M, B)

SecureSessionProvokee:

$\Gamma; \Delta, A$ has **step3e**$(B, N_b, N_a, \mathcal{T}_i), A$ has **unpack**$(encPub(A), A, \mathcal{T}_j), A$ has **msg**(**plain**$(N_a), \mathcal{T}_j)$ $\longrightarrow \Gamma; \Delta, A$ has **sessionProv**(B, N_b, N_a)

Appendix B

Proof Languge

We give a proof language to specify rewrite step derivations and state transition sequences.

B.1 Notation

A protocol state is represented as a bracketed list of proof terms with proposition types. We give our initial state $\Gamma; \Delta$ first, and then we construct a state transition $\Gamma; \Delta \longrightarrow \Gamma'; \Delta'$ under rule R which evaluates to a terminal protocol state $\Gamma_T; \Delta_T$. The terminal state contains all facts remaining in the linear context which have not been consumed in the derivation. The proof is constructed in the form of a let-binding in which we introduce only the new proof terms in $\Gamma'; \Delta'$ resulting from applying R to only the relevant proof terms in $\Gamma; \Delta$.

As an example, suppose $\Gamma; \Delta$ is represented as `[x1:A1, ..., xn:An]`, where x_i is a proof term of proposition type A_i. Suppose also that some rule R has a rewrite step of the form $\Gamma_R; \Delta_R, A_2, A_3, A_4 \longrightarrow \Gamma_R; \Delta_R, B_1, B_2$. Then the derivation $\Gamma; \Delta \longrightarrow \Gamma'; \Delta'$ under R would be expressed in our proof language as it is in Figure B.1. Notice that the terminal state contains all proof terms which have not been consumed in a sequence of state transitions.

Extending beyond single rewrite step derivations, a state transition sequence derivation is essentially represented as nested rewrite step proofs. We consider an initial state $\Gamma; \Delta$ represented

```
[ x1:A1, ..., xn:An ]
let
  [y1:B1, y2:B2] = R [x2:A2, x3:A3, x4:A4]
in
  [y1:B1, y2:B2, x1:A1, x5:A5, ..., xn:An]
end
```

Figure B.1: A derivation of a state transition under rule R.

as `[w1:A1, w2:A2]`. Let ϵ be a sequence of rewrite steps R_1, R_2, R_3. Suppose the rewrite step for R_1 takes the form $\Gamma; \Delta, A_1 \longrightarrow \Gamma; \Delta, B_1, B_2$, the step for R_2 takes the form $\Gamma; \Delta, B_1 \longrightarrow \Gamma; \Delta, C_1, C_2$, and the step for R_3 takes the form $\Gamma; \Delta, A_2, C_1 \longrightarrow \Gamma; \Delta, D_1$. Then the derivation for $\Gamma; \Delta \xrightarrow{\epsilon} \Gamma'; \Delta'$ would be expressed as it is in Figure B.2.

```
[ w1:A1, w2:A2 ]
let
  [ x1:B1, x2:B2 ] = R1 [ w1:A1 ]
  [ y1:C1, y2:C2 ] = R2 [ x1:B1 ]
  [ z1:D1 ] = R3 [ w2:A2, y1:C1 ]
in
  [ x2:B2, y2:C2, z1:D1 ]
end
```

Figure B.2: A derivation of a state transition sequence under ϵ.

B.2 Soundness Trace

Here we demonstrate a more involved use of the proof language to generate a trace through an execution of the Needham-Schröder-Lowe protocol that corresponds to the Soundness proof derivation in Section 3.4.

```
[ a1:[[A]]PrivateKey(KA-, A), a2:[[A]]Server(S),
 a3:[[A]]PublicKey(KS+, S), a4:[[S]]Server(S),
 a5:[[S]]PublicKey(KA+, A), a6:[[S]]PublicKey(KP+, P),
 a7:[[S]]PrivateKey(KS-, S), a8:[[P]]PrivateKey(KP-, P),
```

```
 a9:[[P]]Server(S), a10:[[P]]PublicKey(KS+, S),
 a11:[A]needkey(P), a12:[A]init(P) ]
let
  // A sends S a request for P's public key
  [ b1:<A>send(plain(getkey(P, A)), S) ] =
    RequestKey [ a11:[A]needkey(P), a2:[[A]]Server(S),
    b2:!AgentNeq(P, S) ]

  // S receives and unpacks the message
  [ c1:[S]recv(plain(getkey(P, A)), S) ] =
    ReceiveMessage [ b1:<A>send(plain(getkey(P, A)), S) ]
  [ d1:[S]unpack(plain, S, Ti), d2:[S]msg(plain(getkey(P, A)), Ti) ] =
    UnpackPlain [ c1:[S]recv(plain(getkey(P, A)), S) ]

  // S processes A's request for P's key
  [ e1:[S]step0a(P, A), e2:[S]needkey(P) ] =
    GetKeyRequest [ d1:[S]unpack(plain, S, Ti),
    d2:[S]msg(plain(getkey(P, A)), Ti), a4:[[S]]Server(S) ]
  [ f1:[S]step0b(P, A, Tj), f2:[S]msg(plain(k(KP+, P)), Tj) ] =
    ConstructKeyMessage [ e1:[S]step0a(P, A),
    a10:[[S]]PublicKey(KP+, P) ]
  [ g1:[S]step0c(P, A, Tj), g2:[S]msg(encPri(H, S), Tj) ] =
    EncryptKey [ f1:[S]step0b(P, A, Tj),
    f2:[S]msg(plain(k(KP+, P)), Tj), a7:[[S]]PrivateKey(KS-, S),
    f3:!Crypt(H, KS-, plain(k(KP+, P))) ]
  [ h1:<S>send(encPri(H, S), A) ] =
    SendKey [ g1:[S]step0c(P, A, Tj), g2:[S]msg(encPri(H, S), Tj)
```

```
// A receives S's message, decrypts it, and learns P's public key
[ i1:[A]recv(encPri(H, S), A) ] =
  ReceiveMessage [ h1:<S>send(encPri(H, S), A) ]
[ j1:[A]unpack(encPri(S), A, Tk), j2:[A]msg(plain(k(KP+, P), Tk) ] =
  UnpackDecryptPrivate [ i1:[A]recv(encPri(H, S), A),
  a3:[[A]]PublicKey(KS+, S), j3:!Crypt(H, KS-, plain(k(KP+, P)) ]
[ k1:[[A]]PublicKey(KP+, P) ] =
  LearnKey [ j1:[A]unpack(encPri(S), A, Tk), a2':[[A]]Server(S),
  j2:[A]msg(plain(k(KP+, P), Tk) ]

// A initializes the protocol, extending an overture to P,
[ l1:[A]step1a(P) ] =
  KeyKnown1 [ a12:[A]init(P), k1:[[A]]PublicKey(KP+, P) ]
[ m1:[A]step1b(P, Na) ] = GenNonce1 [ l1:[A]step1a(P) ]
[ n1:[A]step1c(P, Na, Tl), n2:[A]msg(plain(Na ~ A), Tl) ] =
  CreateOverture [ m1:[A]step1b(P, Na) ]
[ o1:[A]step1d(P, Na, Tl), o2:[A]msg(encPub(H', P), Tl) ] =
  EncryptOverture [ n1:[A]step1c(P, Na, Tl),
  n2:[A]msg(plain(Na ~ A), Tl), k1':[[A]]PublicKey(KP+, P),
  o3:!crypt(H', KP+, plain(Na ~ A)) ]
[ p1:[A]step1e(P, Na, Tl), p2:<A>send(encPub(H', P), P) ] =
  SendOverture [ o1:[A]step1d(P, Na, Tl),
  o2:[A]msg(encPub(H', P), Tl) ]

// P receives A's overture and recognizes it as such
[ q1:[P]recv(encPub(H', P), P) ] =
```

```
  ReceiveMessage [ p2:<A>send(encPub(H', P), P) ]
[ r1:[P]unpack(endPub(P), P, Tm), r2:[P]msg(plain(Na ~ A), Tm) ] =
  UnpackDecryptPublic [ q1:[P]recv(encPub(H', P), P),
  a8:[[P]]PrivateKey(KP-, P), r3:!Crypt(H', KP+, plain(Na ~ A)) ]
[ s1:[P]step2a(A, Na), s2:[P]msg(plain(Na), Tm), s3:[P]needkey(A) ] =
  RealizeProvokee [ r1:[P]unpack(endPub(P), P, Tm),
  r2:[P]msg(plain(Na ~ A), Tm) ]

// P requests A's public key from S
[ t1:<P>send(plain(getkey(A, P)), S) ] =
  RequestKey [ s3:[P]needkey(A), a9:[[P]]Server(S),
  t2:!AgentNeq(A, S) ]

// S receives and unpacks the message
[ u1:[S]recv(plain(getkey(A, P)), S) ] =
  ReceiveMessage [ t1:<P>send(plain(getkey(A, P)), S) ]
[ v1:[S]unpack(plain, S, Tn), v2:[S]msg(plain(getkey(A, P)), Tn) ] =
  UnpackPlain [ u1:[S]recv(plain(getkey(A, P)), S) ]

// S processes P's request for A's key
[ w1:[S]step0a(A, P), w2:[S]needkey(A) ] =
  GetKeyRequest [ v1:[S]unpack(plain, S, Tn),
  v2:[S]msg(plain(getkey(A, P)), Tn), a4':[[S]]Server(S) ]
[ x1:[S]step0b(A, P, To), x2:[S]msg(plain(k(KA+, A)), To) ] =
  ConstructKeyMessage [ w1:[S]step0a(A, P),
  a5:[[S]]PublicKey(KA+, A) ]
[ y1:[S]step0c(A, P, To), y2:[S]msg(encPri(H'', S), To) ] =
```

```
  EncryptKey [ x1:[S]step0b(A, P, To),
  x2:[S]msg(plain(k(KA+, A)), To), a7':[[S]]PrivateKey(KS-, S),
  y3:!Crypt(H, KS-, plain(k(KA+, A))) ]
[ z1:<S>send(encPri(H'', S), P) ] =
  SendKey [ y1:[S]step0c(A, P, To), y2:[S]msg(encPri(H'', S), To)

// P receives S's message, decrypts it, and learns A's public key
[ aa1:[P]recv(encPri(H'', S), P) ] =
  ReceiveMessage [ z1:<S>send(encPri(H'', S), P) ]
[ ab1:[P]unpack(encPri(S), P, Tp), ab2:[P]msg(plain(k(KA+, A), Tp) ] =
  UnpackDecryptPrivate [ aa1:[P]recv(encPri(H'', S), P),
  a10:[[P]]PublicKey(KS+, S), ab3:!Crypt(H, KS-, plain(k(KA+, A)) ]
[ ac1:[[P]]PublicKey(KA+, A) ] =
  LearnKey [ ab1:[P]unpack(encPri(S), P, Tp), a9':[[P]]Server(S),
  ab2:[P]msg(plain(k(KA+, A), Tp) ]

// P responds to A's overture
[ ad1:[P]step3a(A, Na) ] =
  KeyKnown2 [ s1:[P]step2a(A, Na), ac1:[[P]]PublicKey(KA+, A) ]
[ ae1:[P]step3b(A, Na, Np) ] = GenNonce2 [ ad1:[P]step3a(A, Na) ]
[ af1:[P]step3c(A, Na, Np, Tq), af2:[P]msg(plain(Na ~ Np ~ P), Tq) ] =
  CreateOvertureResponse [ ae1:[P]step3b(A, Na, Np),
  s2:[P]msg(plain(Na), Tm) ]
[ ag1:[P]step3d(A, Na, Np, Tq), ag2:[P]msg(encPub(H''', A), Tq) ] =
  EncryptOvertureResponse [ af1:[P]step3c(A, Na, Np, Tq),
  af2:[P]msg(plain(Na ~ Np ~ P), Tq), ac1':[[P]]PublicKey(KA+, A),
  ag3:!Crypt(H''', KA+, plain(Na ~ Np ~ P)) ]
```

```
[ ah1:[P]step3e(A, Na, Np, Tq), ah2:<P>send(encPub(H’’’, A), A) ] =
  SendOvertureResponse [ ag1:[P]step3d(A, Na, Np, Tq),
  ag2:[P]msg(encPub(H’’’, A), Tq) ]

// A receives P’s response and recognizes it as such
[ ai1:[A]recv(encPub(H’’’, A), A) ] =
  ReceiveMessage [ ah2:<P>send(encPub(H’’’, A), A) ]
[ aj1:[A]unpack(encPub(A), A, Tr),
aj2:[A]msg(plain(Na ~ Np ~ P), Tr) ] =
  UnpackDecryptPublic [ ai1:[A]recv(encPub(H’’’, A), A),
  a1:[[A]]PrivateKey(KA-, A),
  aj3:!Crypt(H’’’, KA-, plain(Na ~ Np ~ P)) ]
[ ak1:[A]step4a(P, Na, Tl, Np), ak2:[A]msg(plain(Nb), Ts) ] =
  RealizeProvokeeResponded [ p1:[A]step1e(P, Na, Tl),
  aj1:[A]unpack(encPub(A), A, Tr),
  aj2:[A]msg(plain(Na ~ Np ~ P), Tr) ]

// A responds to P’s response and concludes that
// they have established a secure session with P
[ al1:[A]step4b(P, Na, Tl, Np, Ts), al2:[A]msg(encPub(H’’’’, P), Ts) ] =
  EncryptProvokeeNonce [ ak1:[A]step4a(P, Na, Tl, Np),
  ak2:[A]msg(plain(Np), Ts), k1’:[[A]]PublicKey(KP+, P),
  al3:!Crypt(H’’’’, KP+, plain(Np)) ]
[ am1:[A]sessionApp(P, Na, Np),
  am2:<A>send(encPub(H’’’’, P), P) ] =
  SecureSessionApproacher [ al1:[A]step4b(P, Na, Tl, Np, Ts),
  al2:[A]msg(encPub(H’’’’, P), Ts) ]
```

```
  // P receives and decrypts A's response and concludes
  // that they have established a secure session with A
  [ an1:[P]recv(encPub(H''', P), P) ] =
   ReceiveMessage [ am2:<A>send(encPub(H'', P), P) ]
  [ ao1:[P]unpack(encPub(P), P, Tt), ao2:[P]msg(plain(Np), Tt) ] =
    UnpackDecryptPublic [ an1:[P]recv(encPub(H''', P), P),
    a8':[[P]]PrivateKey(KP-, P), ao3:!Crypt(H''', KP-, plain(Np)) ]
  [ ap1: [P]sessionProv(A, Na, Np) ] =
    SecureSessionProvokee [ ah1:[P]step3e(A, Na, Np, Tq),
    ao1:[P]unpack(encPub(P), P, Tt), ao2:[P]msg(plain(Np), Tt) ]
in
  // The linear context
  [ e2:[S]needkey(P), w2:[S]needkey(A),
  am1:[A]sessionApp(P, Na, Np),
  ap1:[P]sessionProv(A, Na, Np) ]
end
```

Bibliography

[1] Henry DeYoung and Frank Pfenning. Reasoning about the consequences of authorization policies in a linear epistemic logic. In *Workshop on Foundations of Computer Security (FCS'09)*, Los Angeles, California, August 2009. 1, 1.2.2, 3.2

[2] D. Dolev and Andrew C. Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 29(2):198–208, Mar 1983. ISSN 0018-9448. doi: 10.1109/TIT.1983.1056650. 1.1

[3] Nancy Durgin, Patrick Lincoln, John Mitchell, and Andre Scedrov. Multiset rewriting and the complexity of bounded security protocols. *Journal of Computer Security*, 12:247–311, 2004. 1.1, 4.1, 4.2

[4] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1 – 101, 1987. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/0304-3975(87)90045-4. URL `http://www.sciencedirect.com/science/article/pii/0304397587900454`. 1.2.1

[5] Gavin Lowe. An attack on the Needham-Schroeder public key authentication protocol. *Information Processing Letters*, 56(3):131–136, November 1995. 1, 2.1, 2.1, 2, 4.2

[6] Jean marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2:297–347, 1992. 1.2.2

[7] Roger Needham and Michael Schroeder. Using encryption for authentification in large networks of computers. *Communications of the ACM*, 21(12), December 1978. 1.1

[8] Frank Pfenning. 15-816 Linear Logic course notes, 2001. 1.2.1