On a Calculus for Security Protocol Development

Gavin Lowe and Michael Auty

January 27, 2005

Abstract

This paper describes a calculus for developing security protocols. Protocol descriptions are annotated with assertions that state properties that will be true when the protocol execution reaches that point. Proof rules are given that allow the assertions to be verified. A novel feature of the calculus is that the initial development of the protocol uses abstract messages that describe the intention of a message, rather than the concrete implementation; rules are given that allow these abstract messages to be refined to concrete implementations. A semantic model of protocol executions is presented. This is used to give a formal meaning to protocol annotations and to abstract messages, and to verify annotation rules and message refinement rules. The calculus is illustrated with the development of a well know protocol; the development helps to cast light on the failure of an earlier version of that protocol.

Contents

1 Introduction

Creating security protocols is a difficult task. Numerous security protocols have been published, only later to be discovered to be flawed; for example, the Needham Schroeder Public Key Protocol was first published in 1978 [NS78], and was the subject of several subsequent analyses (e.g. [BAN89]), only to be found to be flawed in 1995 [Low95].

Various approaches to analysing protocols have been proposed. State exploration techniques (for example [Low96, Low98, MCJ97, MMS97]) build a model of the state space of a small instance of the protocol (with a bounded number of protocol runs), together with a model of the most general attacker who can interact with the protocol, and then use a tool to explore the state space, looking for insecure states. Theorem provers have been used to produce machine-assisted proofs of protocols (for example [Pau98, Coh00]). The NRL Analyzer [Mea96] combines automated theorem proving with state space analysis techniques. Protocols have been verified directly by hand using special-purpose logics such as BAN Logic [BAN89], or GNY Logic [GNY90]. The Strand Spaces approach [THG99] builds a special-purpose model of protocols; the protocols are then either proved by hand, or automatically (for example using Athena [SBP01]).

All these approaches adopt the Dolev-Yao Model [DY83]. It is assumed that the network is under the complete control of a malicious agent or intruder. The intruder can intercept all messages passing on the network; he can create new messages from those he has already seen or knew initially, by encrypting or decrypting with known keys, concatenating or splitting pairs, or hashing; and he can send messages he creates, possibly claiming to come from a different agent. However, perfect cryptography is assumed: for example, the intruder cannot learn anything from a ciphertext if he does not know the appropriate decrypting key.

Proving the security of a protocol, with any of these methods, is nontrivial; further, the proof often gives limited insight into why the protocol is correct, or why it is designed as it is. Designing a security protocol from scratch is harder: we don't have techniques better than using our experience or intuition to produce a protocol we believe to be correct, and then proving it. This is the question we address in this paper. We present a calculus that allows a protocol to be developed systematically from its requirements, and simultaneously proved correct; the development helps to document why the protocol works.

Protocols in our calculus do not take the form of a standard protocol specification, where each message specifies exactly *how* it should be implemented, built from atomic pieces of data, using concatenation, encryption and hashing. Instead protocols in our calculus use *abstract messages* which convey *what* each message is supposed to do. Abstract messages represent requirements on the corresponding concrete messages, and do not specify how these requirements are achieved. The calculus provides various abstract messages, each specifying a requirement on the concrete message; these can then be conjoined to make stronger requirements. Abstract messages help to document the protocol, by showing what each message is intended to achieve.

Our calculus adapts the idea of program annotations [Hoa69] from programs to security protocols. We annotate the protocol description with assertions that state properties that will be true when a protocol reaches that point. More precisely, each protocol annotation will be from the point of view of a single participant: each assertion will state properties that are guaranteed to be true whenever that participant reaches that point in the protocol. We write $\left\{ pre \right\} e \left\{ post \right\}$ to mean that if the sequence of events e is executed, starting from a state where the precondition *pre* holds, then it can be guaranteed that the postcondition *post* will hold in the final state.

We present proof rules that allow assertions to be verified, based on the abstract messages used, assuming the precondition holds. The calculus thus allows protocols to be synthesised and simultaneously proved correct. It also allows partial annotations to be composed, by matching the final assertion of one with the initial assertion of the next. We also present rules to refine abstract messages into concrete messages.

It turns out that such locally-based reasoning works well with certain properties, particularly authentication-like properties; however, it works less well with others, particularly secrecy-like properties, that are more global in their nature, and thus require one to reason about the protocol as a whole. We tend to capture the latter type of properties as an invariant of the protocol, i.e. a property that is true in all states.

In order to explain precisely the meaning of the constructs of the calculus, and to verify the proof rules, we provide a semantic model. In particular, we present semantics for the abstract messages, stating formally what each abstract message means.

To help the reader understand the various elements involved in this calculus, we give a simple worked example in Section 2, explaining briefly each of the elements. In Section 3 we outline the semantic model upon which the calculus is based. We formalise the meaning of annotations in Section 4, verify some structural annotation rules, and define some useful macros for use in annotations. In Section 5 we define a particular property enjoyed by some protocols, namely disjoint encryption [GTF00]: that different encrypted components within the protocol have distinct forms; we prove a theorems, concerning agreement, which is later useful in verifying message refinement rules. We study abstract messages in more detail in Section 6: for each abstract message, we present a semantic definition, rules to allow it to be used in annotations, and rules to refine it to a concrete message. In Section 7 we look at a larger example, namely that of the Adapted Needham

Schroeder Public Key Protocol [Low95]: we derive the protocol using the rules presented earlier; our development gives some insight into the failings of the original version of the protocol. Finally, in Section 8, we sum up and discuss related work and future directions for the research.

2 Example

In order to illustrate the main features of the calculus, we will use it to develop a small protocol. The entire annotation will be from the point of view of an agent a. The protocol will make use of a nonce challenge to provide fresh authentication guarantees for the agent b, and to establish a shared secret. In a more realistic example, we would do an additional annotation from the point of view of b.

We begin by specifying precisely what we require our protocol to do, defining both the assumptions we will make at the start, and the properties we need to hold at the end. Most of these properties are captured by the invariant of the protocol:

- We assume that, a and b are different agents, and that they are both honest (have predictable behaviour):

  $$a \neq b \wedge \mathit{honest}(a, b).$$

- Further, we will ensure that only a and b get to know the nonce na used in the nonce challenge:

  $$\mathit{defined}(na) \Rightarrow \mathit{knows}(na) \subseteq \{a, b\}.$$

  The macro $\mathit{knows}(x)$ represents the set of participants who know x. The macro $\mathit{defined}(x)$ means that x exists, i.e. a value has been generated for this variable. Here we specify that once na has been generated, only a and b may learn it (it doesn't make sense to talk about who knows na before it is generated).

We therefore define the following invariant:

$$I \ \widehat{=}\ a \neq b \wedge \mathit{honest}(a, b) \wedge (\mathit{defined}(na) \Rightarrow \mathit{knows}(na) \subseteq \{a, b\}).$$

We would like to reach a state in which agent a can be certain that agent b has a session running, with the correct value for na:

$$\mathit{session}(b;\ na).$$

The predicate $\mathit{session}(b;\ na)$ states that the agent b is participating in a session of the protocol, and agrees with the local agent a on the value of the variable na; i.e. b's value for na is the same as a's value for na.

The initial specification for the protocol is shown below:

$$\left\{I\right\}$$
$$\ldots$$
$$\left\{I \wedge \mathit{session}(b;\ na)\right\}$$

We annotate protocols in a style similar to Hoare triples [Hoa69]. The annotations specify statements that are guaranteed to hold when the participant involved reaches that point in the protocol. In this example, we assume that initially the invariant must be true; this represents the precondition of the protocol. At the end of the protocol the invariant and $\mathit{session}(b;\ na)$ must hold; this represent the postcondition of the protocol. The ellipses ("...") represent the part of the protocol that we still need to develop.

We now consider how to maintain the invariant. The first two clauses will clearly be maintained. In order to maintain the third clause, each message that is sent must be such that only the agent b may learn na. This is captured by the abstract message $\mathit{canExtract}_b(na)$. We therefore define a *message invariant*, a property satisfied by every message in the protocol:

$$MI \ \widehat{=}\ \mathit{canExtract}_b(na).$$

Later we will give a rule to justify that this indeed maintains the third clause of the invariant.

We now generate the nonce, and show the facts we can deduce about the subsequent state:

$$\left\{I\right\}$$
$$\mathsf{new}\ na$$
$$\left\{I \wedge \mathit{knows}(na) = \{a\}\right\}$$
$$\ldots$$
$$\left\{I \wedge \mathit{session}(b;\ na)\right\}$$

The $\mathsf{new}\ na$ event creates a new nonce and binds it to na in the local state. Afterwards the invariant will still hold, and only a will know na. This is justified by the following proof rule (the "a :" indicates that the annotation relates to role a):

$$a : \left\{\mathit{pre}\right\}\ \mathsf{new}\ x\ \left\{(\exists\, X_0 \bullet \mathit{pre}[X_0/x]) \wedge \mathit{knows}(x) = \{a\}\right\}$$
provided pre refers only to state variables,

noting that $(\exists\, X_0 \bullet I[X_0/x]) \wedge \mathit{knows}(na) = \{a\}$ is logically equivalent to $I \wedge \mathit{knows}(na) = \{a\}$).

We will often concatenate several events and corresponding assertions; for example, in the above annotation, the $\mathsf{new}\ na$ and resulting assertion is concatenated with the part of the protocol still to be developed. The following proof rule justifies this.

$$\frac{\begin{array}{l} a : \{pre\}e_1\{mid\} \\ a : \{mid\}e_2\{post\} \end{array}}{a : \{pre\}e_1\ e_2\{post\}}$$

We write the resulting annotation, corresponding to the consequence of this rule, as

$$a : \Big\{pre\Big\}\ e_1\ \Big\{mid\Big\}\ e_2\ \Big\{post\Big\}$$

We now arrange for the local agent a to send a message:

$\Big\{I\Big\}$
$\textsf{new}\ na$
$\Big\{I \wedge knows(na) = \{a\}\Big\}$
$\textsf{send}\ MI$
$\Big\{I\Big\}$
$\ldots$
$\Big\{I \wedge session(b;\ na)\Big\}$

A message is sent out that must satisfy the message invariant. The intention is that b will actually learn na from this message, in order to make the next step feasible; however, a cannot be sure of this yet, because he cannot be sure that b receives the message.

We now arrange for a to receive a message which shows him that someone knows na; from that and the other conditions that hold, we can deduce that in fact it can only be b that knows na:

$\Big\{I\Big\}$
$\textsf{new}\ na$
$\Big\{I \wedge knows(na) = \{a\}\Big\}$
$\textsf{send}\ MI$
$\Big\{I\Big\}$
$\textsf{receive}\ MI \wedge provesKnowledgeOfNR(na)$
$\Big\{I \wedge \exists\, b' \bullet session(b';\ na) \wedge b' \neq a\Big\}$
$\Big\{I \wedge session(b;\ na)\Big\}$

This message centres around $provesKnowledgeOfNR(na)$, which informs the receiver a that somebody knows the value of na; further, that participant is not a himself: this extra clause ensures that messages are not reflected (the

"NR" stands for "not reflected"). This gives us the $\exists\, b' \bullet \mathit{session}(b';\ na) \wedge b' \neq a$ clause of the assertion. The message also uses the message invariant to ensure the invariant is maintained.

We can now deduce that since someone does exist who has na in their state, and that person is not a, and since the only people who know na (and thus could have na in their state) are a and b (since $\mathit{knows}(na) \subseteq \{a, b\}$), that the person who has na in their state must be b, i.e. $\mathit{session}(b;\ na)$. This establishes the required postcondition.

Formally, we have used the following rule:

$$\frac{\begin{array}{l} a : \{\mathit{pre}\}e\{\mathit{post}\} \\ \mathit{post} \Rightarrow \mathit{post}' \end{array}}{a : \{\mathit{pre}\}e\{\mathit{post}'\}}$$

We will tend to write the resulting annotation as $a : \{\mathit{pre}\}e\{\mathit{post}\}\{\mathit{post}'\}$. For completeness we also present the complimentary rule:

$$\frac{\begin{array}{l} a : \{\mathit{pre}\}e\{\mathit{post}\} \\ \mathit{pre}' \Rightarrow \mathit{pre} \end{array}}{a : \{\mathit{pre}'\}e\{\mathit{post}\}}$$

We will tend to write the resulting annotation as $a : \{\mathit{pre}'\}\{\mathit{pre}\}e\{\mathit{post}\}$.

It should be noted that the abstract messages do not specify how their requirements should be met, merely what properties they must achieve. We now seek to refine the abstract messages to concrete ones. We write $m \sqsubseteq m'$ if message m' meets the requirements of m; typically, m will be an abstract message, and m' a concrete implementation. In some cases, a refinement will hold only in the context of the protocol Π in question, perhaps depending upon some other property that is invariant for the protocol; we sometimes write $m \sqsubseteq_{\Pi} m'$ to stress this dependence.

In order to keep na secret, we will introduce an extra initial assumption, namely that the agents share a secret key k; this will be used to encrypt na in the first message. We strengthen the invariant to specify that k remains secret:

$$\begin{aligned} I \ \widehat{=}\ & a \neq b \wedge \mathit{honest}(a, b) \wedge \mathit{knows}(k) = \{a, b\} \wedge \\ & (\mathit{defined}(na) \Rightarrow \mathit{knows}(na) \subseteq \{a, b\}). \end{aligned}$$

We also strengthen the message invariant accordingly:

$$\mathit{MI} \ \widehat{=}\ \mathit{canExtract}_b(na) \wedge \mathit{keepSecret}(k).$$

The abstract message $\mathit{keepSecret}(k)$ means that nobody may learn k from the message; this ensures the new clause of the invariant is maintained.

We can refine the first abstract message as follows. Firstly, note that

$$\mathit{canExtract}_b(na) \ \sqsubseteq\ \{na\}_k,$$

assuming $knows(k) = \{a, b\}$, for nobody other than a and b will be able to decrypt the message. Further

$$keepSecret(k) \sqsubseteq \{na\}_k.$$

We will give rules later to formally justify these refinements. Hence

$$MI \sqsubseteq \{na\}_k.$$

This last step is justified by the following proof rule concerning the refinement of the conjunction of abstract messages:

$$\frac{\begin{array}{l} m_1 \sqsubseteq m \\ m_2 \sqsubseteq m \end{array}}{m_1 \wedge m_2 \sqsubseteq m}$$

Similarly the second abstract message can be refined by hashing na with the identity of the sender: $hash(na, b)$; the agent a will not accept the message if the identifier b included in the hash is not as expected. It is then reasonably clear that

$$MI \wedge provesKnowledgeOfNR(na) \sqsubseteq hash(na, b).$$

This gives us the concrete protocol below:

new na
send $\{na\}_k$
receive $hash(na, b)$

It turns out that we will have to slightly strengthen the initial assumptions in order to formally justify these refinements: the additional assumptions are necessary, but not obvious, and come out directly from the refinement rules. We will discuss this further when we present those rules, in Section 6.

Of course, the above is not the only possible refinement of the abstract messages. We could have implemented the first message by encrypting na with b's public key ($\{na\}_{PK(b)}$), for example, under suitable assumptions, such as b's secret key being known only to b.

3 Protocol semantics

In this section we build a semantic model of protocol executions; in later sections we build on this to give a semantics to annotations, give a semantics to abstract messages, and prove annotation and refinement rules.

We begin, in Section 3.1 be defining the types of messages and message templates. In Section 3.2 we define abstract messages. We define the local states of agents in Section 3.3, and give an operational semantics. In

Section 3.4 we consider what it means for a protocol to be feasible for a particular agent. We describe the model of the intruder in Section 3.5. We combine these in Section 3.6 to give the model of a global state, lift the operational semantics for individual agents to the global level, and define the traces of a protocol. The notation introduced is summarised in Appendix A.

3.1 Messages

We begin by defining the type of actual messages. It is important to distinguish between message templates and actual messages: the former contain free variables, and are used in the definition of a protocol; the latter have all the variables instantiated with values, and are what are actually sent across the network.

We assume disjoint types *Var* of variables and *Val* of atomic values. We use variables for two purposes within our model: to represent fields within a protocol definition, and as program variables storing values in agents' states. We use the convention of representing variables by small letters and values by capitals. We also assume the existence of a special value $\perp \notin Val$, representing an undefined value.

We assume two inverse functions:

$$\begin{aligned} \_^{-1_{var}} &: Var \nrightarrow Var, \\ \_^{-1_{val}} &: Val \nrightarrow Val. \end{aligned}$$

between variables and values. If M is a value then messages encrypted with M can be decrypted with $M^{-1_{val}}$, and vice versa. If x is a variable then it is intended that $x^{-1_{var}}$ holds the corresponding decrypting key. We assume that $(x^{-1_{var}})^{-1_{var}} = x$, and similarly for values. We will drop the *val* and *var* subscripts where that will not cause confusion.

We define actual messages and message templates by the grammars:

$$\begin{aligned} Msg &::= Val \mid (Msg, Msg) \mid \{Msg\}_{Val} \mid hash(Msg), \\ Template &::= Var \mid Val \mid (Template, Template) \mid \\ &\qquad \{Template\}_{Var} \mid hash(Template). \end{aligned}$$

Messages and templates are built up from atomic values by pairing, encryption and hashing[1]. We omit parentheses where appropriate. Note that an actual message can be obtained from a template by substituting or instantiating all the free variables with values. We use the convention of representing templates by small letters and actual messages by capitals.

We define three submessage relations for later use. We write $M \sqsubset M'$ if M is textually included within M':

$$M \sqsubset M' \Leftarrow M = M',$$

[1]We assume a single hash function, but it is straightforward to extend the model to multiple hash functions.

$$
\begin{aligned}
M \sqsubset (M_1, M_2) &\Leftarrow M \sqsubset M_1 \lor M \sqsubset M_2,\\
M \sqsubset \{M'\}_K &\Leftarrow M \sqsubset M',\\
M \sqsubset \mathit{hash}(M') &\Leftarrow M \sqsubset M'.
\end{aligned}
$$

We define a similar relation over message templates:

$$
\begin{aligned}
m \sqsubset m' &\Leftarrow m = m',\\
m \sqsubset (m_1, m_2) &\Leftarrow m \sqsubset m_1 \lor m \sqsubset m_2,\\
m \sqsubset \{m'\}_k &\Leftarrow m \sqsubset m',\\
m \sqsubset \mathit{hash}(m') &\Leftarrow m \sqsubset m'.
\end{aligned}
$$

We also define submessage relations, over both messages and templates, that include both encryption and decryption keys as submessages:

$$
\begin{aligned}
M \preceq M' &\Leftarrow M = M',\\
M \preceq (M_1, M_2) &\Leftarrow M \preceq M_1 \lor M \preceq M_2,\\
M \preceq \{M'\}_K &\Leftarrow M \preceq M' \lor M \preceq K \lor M \preceq K^{-1},\\
M \preceq \mathit{hash}(M') &\Leftarrow M \preceq M',
\end{aligned}
$$

$$
\begin{aligned}
m \preceq m' &\Leftarrow m = m',\\
m \preceq (m_1, m_2) &\Leftarrow m \preceq m_1 \lor m \preceq m_2,\\
m \preceq \{m'\}_k &\Leftarrow m \preceq m' \lor m \preceq k \lor m \preceq k^{-1},\\
m \preceq \mathit{hash}(m') &\Leftarrow m \preceq m'.
\end{aligned}
$$

Note in particular that the *decrypting* key is a submessage of an encryption. We extend the submessage relation (over messages) to take a set of messages on the right:

$$M \preceq B \Leftrightarrow \exists M' \in B \bullet M \preceq M'.$$

It will also be useful to talk about direct submessages: those submessages that can be obtained without performing any decryption:

$$
\begin{aligned}
m \ll m' &\Leftarrow m = m',\\
m \ll (m_1, m_2) &\Leftarrow m \ll m_1 \lor m \ll m_2.
\end{aligned}
$$

We will make the *strong typing assumption*: i.e. that each honest agent will only accept a value received if it is of the expected type. See [HLS03] for an implementation of this assumption.

We assume a set *TypeName* of names of atomic types (e.g. *Nonce*, *PublicKey*, *AgentIdentity*, ...). We then define a datatype of types of messages by

$$\mathit{Type} ::= \mathit{TypeName} \mid (\mathit{Type}, \mathit{Type}) \mid \{\mathit{Type}\}_{\mathit{Type}} \mid \mathit{hash}(\mathit{Type}).$$

For example, $\{(\mathit{Nonce}, \mathit{AgentIdentity})\}_{\mathit{PublicKey}}$ represents the type of nonces and agent identities encrypted with public keys.

We assume a function

$$type_{var} \ : \ Var \to Type$$

giving the intended types of all variables in the system. Note that this means that if the definitions of two roles in the protocols make use of the same variable name, then they must both give the same type to that variable. We also assume a function

$$type_{val} \ : \ Val \to Type.$$

That gives the types of atomic values. We lift the functions to message templates and messages

$$\begin{aligned} type_{template} &: \ Template \to Type, \\ type_{msg} &: \ Msg \to Type \end{aligned}$$

in the obvious way. We'll drop the subscripts from the $type_*$ functions where that will not cause confusion.

3.2 Abstract messages

In this section we briefly describe the ideas behind abstract messages, and how they are modelled formally. We postpone some of the details to Section 6.

The idea behind abstract messages is that most protocol designers know what they are trying to achieve, but have to write concrete message templates which may have other meanings, or which do not entirely capture the intended meaning. The abstract messages provide the designer with a means to express what the message *should* do, not how to implement it. The concrete implementation of the abstract message can be determined later, and in fact there may be several possible concrete implementations of the same abstract message.

We consider abstract messages defined by the grammar

$$\begin{aligned} AbsMsg \ ::= \ & Template \mid AbsMsg \wedge AbsMsg \mid canExtract_{Var}(Var) \mid \\ & keepSecret(Var) \mid provesKnowledgeOfNR(Var) \mid \ldots . \end{aligned}$$

We have left the grammar open, as we will introduce more abstract messages in Section 6, and we suspect that the study of further example developments will suggest yet more useful abstract messages. Note in particular that a concrete message template is considered to be an abstract message.

We define the semantics of an abstract message to be the set of all the concrete message templates that meet the desired property. The semantics may be dependent upon the protocol: for instance, in one protocol a message may prove knowledge of a value x — and so be an implementation of *provesKnowledgeOfNR*(x) — by revealing a different value y that was previously encrypted with x; however, this won't be the case in all protocols. For this reason we use a semantic function that takes the abstract message and the particular protocol in question, and returns the semantics (set of possible concrete message templates) for that abstract message. We write $[\![m]\!]_\Pi$ for the semantics of abstract message m in protocol Π:

$$[\![\_]\!]_\_ \ : \ AbsMsg \times Protocol \to \mathbb{P}\ Template.$$

In Section 6 we will give the semantics of each form of abstract message, together with rules for using those abstract messages in annotations, and refining them to concrete messages. However, it is worth giving the semantics of a concrete message template here: it is simply the singleton set containing that concrete template:

$$[\![m]\!]_\Pi \ = \ \{m\}, \qquad \text{for } m \in Template.$$

Recall also that we write $am \sqsubseteq am'$ if abstract message am can be implemented by am'; formally, the protocol is an argument of this relation:

$$am \sqsubseteq_\Pi am' \ \Leftrightarrow \ [\![am]\!]_\Pi \supseteq [\![am']\!]_\Pi,$$

We drop the explicit mention of the protocol when it is clear from the context.

Note that there are two degrees of freedom within an abstract message: the choice (made during the design of the protocol) of concrete message template with which to implement it; and the choice (made at run-time) of values to instantiate the free variables.

It is worth considering the implications of the fact that the refinement relation is parameterised by the protocol Π. There are two scenarios to consider:

- The final protocol is known, and a rational construction or verification is being performed. In this case, each refinement step can be verified against the protocol in question.

- The final protocol is not known, but is being developed. Some of the refinement rules we give later will include conditions on the protocol Π (such as the disjoint encryption property: that different encrypted components in the protocol have textually distinct forms). If such a refinement rule is used, the conditions need to be checked against the part of the protocol developed so far, and borne in mind for the remainder of the development or checked at the end.

3.3 Local states

Our global state will comprise a number of honest agents, or *nodes*, together with an intruder, which communicate together. In this section we describe how we model the local states of honest agents. The model includes the program defining how the agent acts, and the binding of variables to values. We give an operational semantics showing how the local state evolves as events are performed.

3.3.1 Protocol templates

Part of the state of an honest agent will be a definition of the sequence of events that it should perform. As with messages, we distinguish between templates for events (using abstract messages), and the actual events themselves (described below in Section 3.3.4).

We consider four types of *event templates* performed by protocol participants:

send The event template $\mathsf{send}\, m$ represents the sending of a message described by the abstract message m;

$\mathsf{receive}$ The event template $\mathsf{receive}\, m$ represents the receipt of a message described by the abstract message m;

new The event template $\mathsf{new}\, x$ represents the fresh generation of a value to be stored in the variable x;

$\mathsf{newpair}$ The event template $\mathsf{new}(x, y)$ represents the fresh generation of a asymmetric key pair to be stored in the variables x and y; we specify that x and y should be inverses in this case: $y = x^{-1_{var}}$.

Note that we generate both members of a key pair together. To enforce this, we will ban the use of the construct $\mathsf{new}\, x$ for x an asymmetric key (i.e. where $x \in \mathrm{dom}\, \_^{-1} \wedge x^{-1} \neq x$).

Formally, event templates are defined by the grammar

$$\begin{aligned} \mathit{EventTemplate} \ ::= \ & \mathsf{send}\, \mathit{AbsMsg} \mid \mathsf{receive}\, \mathit{AbsMsg} \mid \\ & \mathsf{new}\ \mathit{Var} \mid \mathsf{newpair}(\mathit{Var}, \mathit{Var}). \end{aligned}$$

Note that event templates use *abstract* messages, because annotations use abstract messages. However, the *program* followed by an honest agent will use templates containing *concrete* message templates:

$$\begin{aligned} \mathit{Prog} \ \widehat{=} \ & \{ es : \mathit{EventTemplate}^* \mid \\ & \quad \forall\, m \mid \mathsf{send}\, m \text{ in } es \vee \mathsf{receive}\, m \text{ in } es \bullet m \in \mathit{Template} \}. \end{aligned}$$

We write *prog* for a typical element of *Prog*.

We lift the refinement relation from abstract messages to event templates in the obvious way:

$$\begin{aligned} \mathsf{send}\, m \sqsubseteq_\Pi \mathsf{send}\, m' \ & \Leftrightarrow \ m \sqsubseteq_\Pi m', \\ \mathsf{receive}\, m \sqsubseteq_\Pi \mathsf{receive}\, m' \ & \Leftrightarrow \ m \sqsubseteq_\Pi m', \\ \mathsf{new}\, x \sqsubseteq_\Pi \mathsf{new}\, x, & \\ \mathsf{newpair}(x, y) \sqsubseteq \mathsf{newpair}(x, y). & \end{aligned}$$

We lift the notion of refinement from event templates to programs pointwise:[2]

$$\begin{array}{rcl} prog \sqsubseteq_{\Pi} prog' & \Leftrightarrow & length\,prog = length\,prog' \wedge \\ & & \forall\, i \in 1\,..\,length\,prog \bullet prog(i) \sqsubseteq_{\Pi} prog'(i). \end{array}$$

We write $vars(m)$ for the set of variables appearing in message template m:

$$\begin{array}{rcl} vars(x) & = & \{x\}, \quad \text{for } x \in Var, \\ vars(X) & = & \{\}, \quad \text{for } X \in Val, \\ vars(m_1, m_2) & = & vars(m_1) \cup vars(m_2), \\ vars(\{m\}_k) & = & vars(m) \cup vars(k), \\ vars(hash(m)) & = & vars(m). \end{array}$$

We lift this to event templates and to programs in the obvious way.

3.3.2 Bindings

Part of the local state of each honest agent will record the values of variables. We model this by a partial mapping, or *binding*:

$$Binding \;\widehat{=}\; Var \leftrightarrow Val.$$

We will write ρ for a typical binding. Note that $\rho(x)$ need not be an atomic value: it could be a compound value; this will be the case in a protocol where an agent receives an encrypted message that he is expected to simply forward on to another agent (e.g. the Otway-Rees Protocol [OR87], or the Yahalom Protocol [BAN89]).

Note that the binding is a *partial* function. We will occasionally want to deal with a situation where a public key pk is in the domain of the binding ρ but the corresponding secret key sk isn't, and where we want to talk about the value of sk — or more accurately, we want to talk about the inverse value of the value of pk; in such cases, we will abuse notation and just write $\rho(sk)$; more generally, if $k \notin \mathrm{dom}\,\rho$, $k^{-1} \in \mathrm{dom}\,\rho$, then we write $\rho(k)$ as shorthand for $(\rho(k^{-1_{var}}))^{-1_{val}}$.

Further, we will sometimes write $\rho(x) = \bot$ as shorthand for $x \notin \mathrm{dom}\,\rho$ and if x^{-1} is defined that $x^{-1} \notin \mathrm{dom}\,\rho$, i.e. to indicate that x is not bound in ρ.

If as is a set of variables, it is convenient to define $\rho(as)$ as a shorthand for $\{\rho(a) \mid a \in as\}$.

The operational semantics we give, below, will ensure that variables in the bindings are well-typed, in the sense that if $\rho(x) = X$ then $type_{var}(x) = type_{val}(X)$.

[2] $prog(i)$ represents the ith element of the sequence $prog$.

If ρ is a binding and m a message template, then we write $m[\rho]$ for the corresponding actual message, where each variable x is replaced by $\rho(x)$. We adapt some of the above conventions to substitutions: if $x \notin \operatorname{dom}\rho$, $x^{-1} \in \operatorname{dom}\rho$, then we define $x[\rho] = \rho(x^{-1})^{-1}$; if $x, x^{-1} \notin \operatorname{dom}\rho$, then we define $x[\rho] = \perp$.

Similarly, if P is a predicate, we write $P[\rho]$ for the result of the corresponding substitution.

3.3.3 Local states

We will represent the local state of an agent by a triple $(\mathit{prog}, \rho, \mathit{id}) : \mathit{Prog} \times \mathit{Binding} \times \mathit{Var}$, where prog is the remaining sequence of event templates it needs to perform, ρ is a binding, and $\mathit{id} \in \operatorname{dom}\rho$ is a distinguished variable that represents the local agent's identity. Given a local state s, we will write "$s.\mathit{prog}$", "$s.\rho$" and "$s.\mathit{id}$" to refer to the three components. We use the convention that the selection operator "." binds tighter than all other operators, including function application, so for example $s.\rho(x) = (s.\rho)(x)$. Note that $s.\mathit{id}$ is the *variable* that represents the agent's identity, not the value of that identity, which is stored in $s.\rho(s.\mathit{id})$. One can think of the id variables as being the names of the roles in the protocol.

3.3.4 Operational semantics

We now give operational semantics for local states. We consider four types of *events* performed by protocol participants, analogous to event templates:

send The event $\mathsf{send}\, M$ represents the local agent sending actual message M;

receive The event $\mathsf{receive}\, M$ represents the local agent receiving actual message M;

new The event $\mathsf{new}\, X$ represents the local agent freshly generating the value X;

newpair The event $\mathsf{newpair}(X, Y)$ represents the local agent freshly generating the asymmetric key pair (X, Y).

Formally we define events according to the grammar:

$$\mathit{Event} ::= \mathsf{send}\, \mathit{Msg} \mid \mathsf{receive}\, \mathit{Msg} \mid \mathsf{new}\, \mathit{Val} \mid \mathsf{newpair}(\mathit{Val}, \mathit{Val}).$$

We write $s \xrightarrow{E} s'$ to mean that from local state s, the event E can be performed to reach local state s'. The $\longrightarrow$ relation is defined as follows:

- If the next event template in the program is of the form $\mathsf{new}\, x$, then the agent can perform the event $\mathsf{new}\, X$ for a value X of the same type as x; the binding is updated to bind x to X:

$$(\langle \mathsf{new}\, x \rangle \frown prog, \rho, id) \stackrel{\mathsf{new}\, X}{\longrightarrow} (prog, \rho \oplus \{x \mapsto X\}, id),$$
$$\text{provided } type_{val}(X) = type_{var}(x),\ x \notin \mathrm{dom}\,\_^{-1} \vee x^{-1} = x.$$

We will ensure later that the value X generated is fresh.

- The semantics of newpair is very similar, except two values, which must be inverses, are involved:

$$(\langle \mathsf{newpair}(x, y) \rangle \frown prog, \rho, id) \stackrel{\mathsf{newpair}(X,Y)}{\longrightarrow}$$
$$(prog, \rho \oplus \{x \mapsto X, y \mapsto Y\}, id),$$
$$\text{provided } type_{val}(X) = type_{var}(x),\ type_{val}(Y) = type_{var}(y),$$
$$X^{-1} = Y.$$

- If the next event template in the program is of the form send m, then the agent can perform the event send $m[\rho]$, i.e. where variables are instantiated according to the current binding:

$$(\langle \mathsf{send}\, m \rangle \frown prog, \rho, id) \stackrel{\mathsf{send}\, m[\rho]}{\longrightarrow} (prog, \rho, id).$$

- If the next event template in the program is of the form receive m, then the agent can perform the event receive $m[\rho']$, and update its binding to ρ' for a suitable binding ρ'; more precisely, the new binding must: (1) extend ρ by giving new values to the variables received in m; (2) respect the types of variables; (3) respect inverses:

$$(\langle \mathsf{receive}\, m \rangle \frown prog, \rho, id) \stackrel{\mathsf{receive}\, m[\rho']}{\longrightarrow} (prog, \rho', id),$$
$$\text{provided } \rho' \supseteq \rho,\ \mathrm{dom}\,\rho' = \mathrm{dom}\,\rho \cup vars(m),$$
$$\forall\, x \in \mathrm{dom}\,\rho' \bullet type_{var}(x) = type_{val}(\rho(x)),$$
$$\forall\, x, y \in \mathrm{dom}\,\rho' \bullet x^{-1} = y \Rightarrow \rho'(x)^{-1} = \rho'(y).$$

  Note, in particular, that if a variable has had a value bound to it already, and a message using that variable is received, then only the previous value will be accepted: this means that the value received must be checked against the value stored. We will ensure later that we consider only protocols that are feasible, i.e. where the agent really is able to unpack every message he receives to obtain the value for each variable.

We adopt standard shorthands concerning the transition relation; for example, we write $s \longrightarrow s'$ for $\exists\, E \in Event \bullet s \stackrel{E}{\longrightarrow} s'$.

The following lemma captures some properties of the operational semantics.

Lemma 1

1. If $(\langle e\rangle \frown prog, \rho, id) \xrightarrow{E} (prog, \rho', id')$ then $\rho \subseteq \rho' \wedge \operatorname{dom}\rho' = \operatorname{dom}\rho \cup vars(e) \wedge id' = id$.
2. If $(prog \frown prog', \rho, id) \longrightarrow^{*} (prog', \rho', id')$ then $\rho \subseteq \rho' \wedge \operatorname{dom}\rho' = \operatorname{dom}\rho \cup vars(prog) \wedge id' = id$ (where we have lifted *vars* to programs in the obvious way).

Proof: (sketch)

1. This follows from a straightforward case analysis.
2. This follows from the previous case by a straightforward induction.

$\square$

We say that a binding is well-typed if the type of every variable agrees with the type of the value stored in it, and variables that represent inverses of one another store values that are inverses of one another:

$$\begin{aligned} wellTyped(\rho) \;\widehat{=}\; & \forall\, x \in \operatorname{dom}\rho \bullet type_{var}(x) = type_{val}(\rho(x)) \\ & \wedge \\ & \forall\, x, y \in \operatorname{dom}\rho \bullet x^{-1} = y \Rightarrow \rho(x)^{-1} = \rho(y). \end{aligned}$$

The property of being well-typed is preserved by the operational semantics:

Lemma 2

$$wellTyped(\rho) \wedge (prog, \rho, id) \xrightarrow{E} (prog', \rho', id) \;\Rightarrow\; wellTyped(\rho').$$

3.4 Feasible protocols

Recall that a concrete program contains no abstract messages. In this section, we consider the circumstances under which a concrete program is feasible, in the sense that every variable is bound before it is used. We will use the initial binding to store the initial knowledge of the agent in question, i.e. the initial binding will contain those values that it needs to run the protocol, bound to suitable variables. We make this precise below.

We define a predicate *canUnpack* such that $canUnpack(xs, ms)$ means that an agent who has appropriate values for the set of variables xs can unpack the set of templates ms so as to obtain all the variables within it, and also verify that all hashes that are received are as expected. *canUnpack* is defined to be the smallest predicate such that:

$$\begin{aligned} & canUnpack(xs, \{\}), \\ canUnpack(xs, \{v\} \cup ms) \;\Leftarrow\; & canUnpack(xs \cup \{v\}, ms), \\ & \quad\text{for } v \in Var, \end{aligned}$$

$$
\begin{array}{rcl}
\mathit{canUnpack}(xs, \{(m_1, m_2)\} \cup ms) & \Leftarrow & \mathit{canUnpack}(xs, \{m_1, m_2\} \cup ms), \\
\mathit{canUnpack}(xs, \{\{m\}_k\} \cup ms) & \Leftarrow & k^{-1} \in xs \wedge \\
 & & \mathit{canUnpack}(xs, \{m\} \cup ms), \\
\mathit{canUnpack}(xs, \mathit{hash}(m) \cup ms) & \Leftarrow & \mathit{vars}(m) \subseteq xs \wedge \mathit{canUnpack}(xs, ms).
\end{array}
$$

Definition 1 We define *LocalState* to be the set of all triples $(\mathit{prog}, \rho, \mathit{id}) : \mathit{Prog} \times \mathit{Binding} \times \mathit{Var}$ such that:

1. The variable id, representing the agent's identity, is bound:

   $$\mathit{id} \in \operatorname{dom} \rho.$$

2. Whenever the agent is supposed to send a message described by template m, the agent is able to produce the message from his initial knowledge ($\operatorname{dom} \rho$) and the variables bound subsequently ($\mathit{vars}(\mathit{prog}')$ below):

   $$\forall\, \mathit{prog}' \frown \langle \mathsf{send}\ m \rangle \leq \mathit{prog} \bullet \mathit{vars}(m) \subseteq \operatorname{dom} \rho \cup \mathit{vars}(\mathit{prog}').$$

3. Whenever the agent is supposed to receive a message described by template m, the agent is able to unpack the message from his initial knowledge and the variables bound subsequently:

   $$\begin{array}{l}\forall\, \mathit{prog}' \frown \langle \mathsf{receive}\ m \rangle \leq \mathit{prog} \bullet \\ \quad \mathit{canUnpack}(\operatorname{dom} \rho \cup \mathit{vars}(\mathit{prog}'), \{m\}).\end{array}$$

4. Whenever the agent is supposed to generate a new value for a variable, that variable it not already bound:

   $$\begin{array}{l}\forall\, \mathit{prog}' \frown \langle \mathsf{new}\ x \rangle \leq \mathit{prog} \bullet x \notin \operatorname{dom} \rho \cup \mathit{vars}(\mathit{prog}') \wedge \\ \forall\, \mathit{prog}' \frown \langle \mathsf{newpair}(x, y) \rangle \leq \mathit{prog} \bullet x, y \notin \operatorname{dom} \rho \cup \mathit{vars}(\mathit{prog}').\end{array}$$

We say that a protocol is *feasible* if it is a member of *LocalState*. The goal of a protocol development will always be to end up with a feasible protocol, and from now on we will assume that all concrete protocols we deal with are indeed feasible.

Note that one consequence of the above definition is that an honest agent cannot have variables bound as a result of receiving a hash: all variables within the hash must be already known or obtainable from elsewhere in the message.

The following lemma shows that being an element of *LocalState* is preserved by the operational semantics.

Lemma 3 *If* $(\mathit{prog}, \rho, \mathit{id}) \in \mathit{LocalState}$ *and* $(\mathit{prog}, \rho, \mathit{id}) \longrightarrow^* (\mathit{prog}', \rho', \mathit{id})$, *then* $(\mathit{prog}', \rho', \mathit{id}) \in \mathit{LocalState}$.

Proof: (sketch) We need to check each of the conditions from Definition 1 in turn.

Condition 1 is trivial because id does not change, and the binding can only increase (Lemma 1). For condition 2, suppose $prog = prog_1 \frown prog_2 \frown \langle \mathsf{send}\, m \rangle \frown prog_3$ and

$$(prog, \rho, id) \longrightarrow^* (prog_2 \frown \langle \mathsf{send}\, m \rangle \frown prog_3, \rho', id).$$

Then

$$\begin{aligned}
& vars(m) \\
\subseteq\ & \Big\langle \text{the initial state is in } \mathit{LocalState} \Big\rangle \\
& \operatorname{dom} \rho \cup vars(prog_1 \frown prog_2) \\
=\ & (\operatorname{dom} \rho \cup vars(prog_1)) \cup vars(prog_2) \\
=\ & \Big\langle \text{Lemma 1} \Big\rangle \\
& \operatorname{dom} \rho' \cup vars(prog_2),
\end{aligned}$$

as required. Conditions 3 and 4 are very similar. □

3.5 The intruder

We model the intruder by simply recording the set of messages that he knew initially or has seen subsequently. We capture this formally when we discuss global states, below.

We will need to capture the way the intruder can produce new messages from messages he already knows. We write $B \vdash M$ is the message M can be obtained from the set of messages B by the intruder. The relation $\vdash$ is defined by the following six rules.

member $M \in B \Rightarrow B \vdash M$;

pair $B \vdash M_1 \wedge B \vdash M_2 \Rightarrow B \vdash (M_1, M_2)$;

split $B \vdash (M_1, M_2) \Rightarrow B \vdash M_1 \wedge B \vdash M_2$;

encrypt $B \vdash M \wedge B \vdash K \Rightarrow B \vdash \{M\}_K$;

decrypt $B \vdash \{M\}_K \wedge B \vdash K^{-1} \Rightarrow B \vdash M$;

hash $B \vdash M \Rightarrow B \vdash hash(M)$.

Below we write "*intruder*" for the identity of the intruder[3].

[3]It is straightforward to extend the model so as to give the intruder multiple identities, or equivalently to allow several intruders with different identities to work together.

3.6 Global states

A *global state* is a collection of local states of honest agents, together with the state of the intruder. We model this by a function σ with domain $0 \mathinner{..} n$ for some n: $\sigma(0)$ will represent the state of the intruder; $\sigma(1), \ldots, \sigma(n)$ will represent the states of the honest agents. Formally:

$$\begin{aligned} \mathit{GlobalState} \;\widehat{=}\; \{\sigma : \mathbb{N} \nrightarrow (\mathit{LocalState} \cup \mathbb{P}\,\mathit{Message}) \mid \\ \exists\, n : \mathbb{N} \bullet \operatorname{dom}\sigma = 0 \mathinner{..} n \wedge \sigma(0) \in \mathbb{P}\,\mathit{Message} \wedge \\ \forall\, i \in 1 \mathinner{..} n \bullet \sigma(i) \in \mathit{LocalState}\}. \end{aligned}$$

Note that several different nodes may have the same identity variables, representing that several nodes are running the same role in the protocol. Further, several different nodes may have the same value (in the binding) for the identity variables, representing that a particular honest agent may be running several different programs (or roles) simultaneously.

In our examples and informal discussions, we will tend to assume that all of the roles in the global state belong to the same protocol. However, this is not necessary: our model includes the possibility of roles from several different protocols, modelling the case of several protocols operating in the same environment. Recall that some of our message refinement rules will be dependent upon the protocols in question; typically, the rules will place restrictions, such as disjoint encryption, upon the protocols; when we are considering an enviroment containing several protocols, these restrictions apply to *all* of those protocols. We briefly return to this point in the conclusion.

Below we will write σ_0 for the initial global state, and n for the number of honest nodes. We take a system running a protocol to be defined by σ_0 together with the typing environment provided by type_{var}, type_{val}, $\_^{-1_{var}}$ and $\_^{-1_{val}}$.

We assume that the intruder's identity *intruder* is distinct from the identities of all the other nodes:

$$\forall\, i \in 1 \mathinner{..} n \bullet \sigma_0(i).\rho(\sigma_0(i).id) \neq \mathit{intruder}.$$

In most annotations, we will assume that the programs of different nodes are consistent in the sense that they use the same variable name for variables that are intended to be equal. For example, if an agent has a send event $\mathsf{send}\; m$ that is intended to be received in the event $\mathsf{receive}\; m'$, then m and m' will be defined using the same variables, so will in fact be syntactically equal. Further, if two nodes have the same identity variables, they will be running the same program: $\sigma_0(i).id = \sigma_0(j).id \Rightarrow \sigma_0(i).prog = \sigma_0(j).prog$.

3.6.1 Operational semantics

We now give operational semantics for global states. We write $\sigma \xrightarrow{i:E} \sigma'$ to represent that from global state σ, node i can perform the event E causing the global state to evolve to σ'. The operational semantics is defined by the four rules below. We arrange for all communications to go via the intruder, rather than having honest agents synchronise directly; so a send event by an honest agent simply causes the corresponding message to be added to the intruder's knowledge; and a $\mathsf{receive}$ event can happen provided the intruder can produce the corresponding message.

We consider first $\mathsf{new}\, X$ events. We need to specify that the value X that results from this event really is a new value; this is captured by the following predicate:

$$isNew(X)(\sigma) \ \widehat{=}\ X \not\preceq \sigma(0) \wedge \forall\, i > 0;\ y \in \operatorname{dom} \sigma(i).\rho \bullet X \not\preceq \sigma(i).\rho(y).$$

The event $i : \mathsf{new}\, X$ can occur if: (1) the node i can do the corresponding $\mathsf{new}\, X$ event; (2) no other node changes its state; and (3) the value X is new:

$$\frac{\begin{array}{l}\sigma(i) \xrightarrow{\mathsf{new}\, X} \sigma'(i) \\ \forall\, j \in 0 \,..\, n \mid j \neq i \bullet \sigma(j) = \sigma'(j) \\ isNew(X)(\sigma)\end{array}}{\sigma \xrightarrow{i:\mathsf{new}\, X} \sigma'} \quad [\, i > 0\,]$$

The semantics of $\mathsf{newpair}$ events is very similar:

$$\frac{\begin{array}{l}\sigma(i) \xrightarrow{\mathsf{newpair}(X,Y)} \sigma'(i) \\ \forall\, j \in 0 \,..\, n \mid j \neq i \bullet \sigma(j) = \sigma'(j) \\ isNew(X)(\sigma) \wedge isNew(Y)(\sigma)\end{array}}{\sigma \xrightarrow{i:\mathsf{newpair}(X,Y)} \sigma'} \quad [\, i > 0\,]$$

The event $i : \mathsf{send}\, M$ can occur if: (1) the node i can do the corresponding $\mathsf{send}\, M$ event; (2) M is added to the intruder's knowledge; and (3) no other node changes its state:

$$\frac{\begin{array}{l}\sigma(i) \xrightarrow{\mathsf{send}\, M} \sigma'(i) \\ \sigma'(0) = \sigma(0) \cup \{M\} \\ \forall\, j \in 1 \,..\, n \mid j \neq i \bullet \sigma(j) = \sigma'(j)\end{array}}{\sigma \xrightarrow{i:\mathsf{send}\, M} \sigma'} \quad [\, i > 0\,]$$

The event $i : \mathsf{receive}\, M$ can occur if: (1) the node i can do the corresponding $\mathsf{receive}\, M$ event; (2) the intruder is able to produce the message M

to send it (possibly faked) to i; and (3) no other node changes its state:

$$\frac{\begin{array}{l}\sigma(i) \stackrel{\mathsf{receive}\,M}{\longrightarrow} \sigma'(i) \\ \sigma(0) \vdash M \\ \forall j \in 0\,..\,n \mid j \neq i \bullet \sigma(j) = \sigma'(j)\end{array}}{\sigma \stackrel{i:\mathsf{receive}\,M}{\longrightarrow} \sigma'} \quad [\, i > 0 \,]$$

3.6.2 Protocol traces

A *system trace* is an alternating sequence of the form

$$\langle \sigma_0, i_1{:}E_1, \sigma_1, i_2{:}E_2, \sigma_2, \ldots, \sigma_n \rangle,$$

where each σ_j is a global state, each i_j is a node index, and each E_j is an event, such that

$$\sigma_0 \stackrel{i_1:E_1}{\longrightarrow} \sigma_1 \stackrel{i_2:E_2}{\longrightarrow} \sigma_2 \ldots \sigma_n.$$

This trace represents a protocol run in which the initial state is σ_0, then event $i_1 : E_1$ occurs and the state evolves into σ_1, and so on. We write $traces(\Pi)$ for the set of all traces that can be observed of Π. We define $States(\Pi)$ to be all the reachable states, i.e. states appearing in some trace of Π.

We write $\sigma \stackrel{\langle i_1:E_1,\ldots,i_n:E_n\rangle}{\Longrightarrow} \sigma'$ to indicate $\sigma \stackrel{i_1:E_1}{\longrightarrow} \ldots \stackrel{i_n:E_n}{\longrightarrow} \sigma'$. We write $\sigma \Longrightarrow \sigma'$ for $\exists\, tr \bullet \sigma \stackrel{tr}{\Longrightarrow} \sigma'$.

If tr is a sequence of events, then we write $tr \upharpoonright i$ for the restriction of tr to the events performed by node i:

$$\begin{array}{rcll} \langle\rangle \upharpoonright i & = & \langle\rangle, & \\ (\langle j : E\rangle \frown tr) \upharpoonright i & = & \langle E\rangle \frown (tr \upharpoonright i), & \text{if } j = i, \\ & & tr \upharpoonright i, & \text{otherwise.} \end{array}$$

4 Annotations

In this section we consider annotations in more detail. In Section 4.1 we formally define the meaning of an annotation. In Section 4.2 we give some structural annotation rules. In Section 4.3 we give formal definitions of the annotation macros we have used, together with a few annotation rules using them. Finally in Section 4.4 we give an annotation rule for the $\mathsf{new}\,x$ construct.

4.1 Correctness of annotations

Consider an assertion P that is intended to hold for a node $i > 0$ in some state σ. The free variables within P refer to the values within i's binding ($\sigma(i).\rho$) and so need to be substituted with those values; the resulting predicate is then interpreted with respect to σ: $P[\sigma(i).\rho](\sigma)$. We abbreviate this to $P(\sigma)[i]$, pronounced "P in σ for i":

$$P(\sigma)[i] \;\widehat{=}\; P[\sigma(i).\rho](\sigma).$$

For example

$$(\mathit{knows}(x) = \{a, b\})(\sigma)[i] \;\;\equiv\;\; \mathit{knows}(X)(\sigma) = \{A, B\}$$
$$\text{where } X = \sigma(i).\rho(x),\ A = \sigma(i).\rho(a),\ B = \sigma(i).\rho(b).$$

We specify that if $x \notin \operatorname{dom} \sigma(i).\rho$ but $x^{-1} \in \operatorname{dom} \sigma(i).\rho$, then the effect of the substitution on x is to produce $(\sigma(i).\rho(x^{-1}))^{-1}$; and if $x, x^{-1} \notin \operatorname{dom} \sigma(i).\rho$, then the effect of the substitution on x is to produce $\perp$.

Note that we need to be careful with the substitution, for not every free occurrence of a variable x within P refers to i's value for x: some may refer to a different node's value for x. In such cases, we define the substitution to "do the right thing"; we make this more precise when we discuss relevant macros, below, specifically the *session* macro.

We can now define invariants of protocols:

Definition 2 Predicate P is an *invariant* of protocol Π for node i if

$$\forall \sigma \in \mathit{States}(\Pi) \bullet P(\sigma)[i].$$

Predicate P is an *invariant* of protocol Π for role a if P is invariant for every node with identity a:

$$\forall \sigma \in \mathit{States}(\Pi);\ i \in 1 \mathrel{..} n \mid \sigma(i).\mathit{id} = a \bullet P(\sigma)[i].$$

Suppose $\sigma_0(i).\mathit{prog} = \mathit{es}_0 \frown \mathit{es}_1$; then to say that i can be sure that predicate P holds after es_0 means that for every state σ where i has remaining program es_1, it must be the case that P holds in σ for i:

$$\forall \sigma \in \mathit{States}(\Pi) \mid \sigma(i).\mathit{prog} = \mathit{es}_1 \bullet P(\sigma)[i].$$

We now formally define the annotation $a : \Big\{\mathit{pre}\Big\}\ \mathit{es}\ \Big\{\mathit{post}\Big\}$, where es is a sequence of *abstract* message templates. Roughly speaking, we want to say that the annotation is correct if *post* holds just after *es* is performed, assuming *pre* always holds just before *es*. Recall, however, that the annotation may use abstract messages within *es*, whereas the actual system will use concrete messages; we therefore consider all executions resulting from event templates that are refinements of *es*. More precisely, if $\sigma(i).\mathit{id} = a$ and $\sigma_0(i).\mathit{prog} = \mathit{es}_0 \frown \mathit{es}' \frown \mathit{es}_1$ where $\mathit{es}' \sqsupseteq \mathit{es}$, then the annotation is correct if *post* always holds after $\mathit{es}_0 \frown \mathit{es}'$, assuming *pre* always holds after es_0.

Definition 3

$$\begin{array}{l} a : \Big\{ pre \Big\}\ es\ \Big\{ post \Big\} \mathrel{\widehat{=}} \\ \quad \forall\, i \in 1 \mathinner{..} n \mid \sigma_0(i).id = a \bullet \\ \quad \forall\, es_0, es_1, es' \mid \sigma_0(i).prog = es_0 \frown es' \frown es_1 \land es' \sqsupseteq es \bullet \\ \qquad (\forall\, \sigma \in States(\Pi) \mid \sigma(i).prog = es' \frown es_1 \bullet pre(\sigma)[i]) \\ \qquad \Rightarrow \\ \qquad (\forall\, \sigma' \in States(\Pi) \mid \sigma'(i).prog = es_1 \bullet post(\sigma')[i]). \end{array}$$

The following lemma relates annotations to invariants.

Lemma 4 If

$$\begin{array}{l} \forall\, i \in 1 \mathinner{..} n \mid \sigma_0(i).id = a \bullet \\ \quad P(\sigma_0)[i] \land \forall\, e \text{ in } \sigma(i).prog \bullet a : \Big\{ P \Big\}\ e\ \Big\{ P \Big\} \end{array}$$

then P is an invariant of the protocol for a.

Proof: (sketch) By induction on the length of the trace leading to each state. The first conjunct of the assumption proves the base case; the second conjunct proves the inductive case. □

4.2 Structural annotation rules

We now prove some of the structural annotation rules that we used earlier. Within these rules, we blur the distinction between single events and sequences of events.

Annotation Rule 1 (Strengthen precondition)

$$\frac{\begin{array}{l} a : \{pre\} e \{post\} \\ pre' \Rightarrow pre \end{array}}{a : \{pre'\} e \{post\}}$$

Proof: Suppose

$$\begin{array}{l} \sigma_0(i).id = a \land \sigma_0(i).prog = es_0 \frown e' \frown es_1 \land e' \sqsupseteq e \land \\ \forall\, \sigma \in States(\Pi) \mid \sigma(i).prog = e' \frown es_1 \bullet pre'(\sigma)[i]. \end{array}$$

Then by the second hypothesis,

$$\forall\, \sigma \in States(\Pi) \mid \sigma(i).prog = e' \frown es_1 \bullet pre(\sigma)[i].$$

Then by the first hypothesis,

$$\forall\, \sigma' \in States(\Pi) \mid \sigma'(i).prog = es_1 \bullet post(\sigma')[i],$$

and so $a : \{pre'\} e \{post\}$ as required. □

Annotation Rule 2 (Weaken postcondition)

$$\frac{\begin{array}{l} a : \{pre\}e\{post\} \\ post \Rightarrow post' \end{array}}{a : \{pre\}e\{post'\}}$$

Proof: Suppose

$$\sigma_0(i).id = a \wedge \sigma_0(i).prog = es_0 \frown e' \frown es_1 \wedge e' \sqsupseteq e \wedge$$
$$\forall\, \sigma \in States(\Pi) \mid \sigma(i).prog = e' \frown es_1 \bullet pre(\sigma)[i].$$

Then by the first hypothesis,

$$\forall\, \sigma' \in States(\Pi) \mid \sigma'(i).prog = es_1 \bullet post(\sigma')[i].$$

Hence, by the second hypothesis,

$$\forall\, \sigma' \in States(\Pi) \mid \sigma'(i).prog = es_1 \bullet post'(\sigma')[i].$$

And so $a : \{pre\}e\{post'\}$, as required. □

Annotation Rule 3 (Sequential composition)

$$\frac{\begin{array}{l} a : \{pre\}e_1\{mid\} \\ a : \{mid\}e_2\{post\} \end{array}}{a : \{pre\}e_1 e_2\{post\}}$$

Proof: Suppose

$$\sigma_0(i).id = a \wedge \sigma_0(i).prog = es_0 \frown e_1' \frown e_2' \frown es_1 \wedge e_1' \sqsupseteq e_1 \wedge e_2' \sqsupseteq e_2 \wedge$$
$$\forall\, \sigma \in States(\Pi) \mid \sigma(i).prog = e_1' \frown e_2' \frown es_1 \bullet pre(\sigma)[i].$$

Then by the first hypothesis,

$$\forall\, \sigma' \in States(\Pi) \mid \sigma'(i).prog = e_2' \frown es_1 \bullet mid(\sigma')[i].$$

Hence by the second hypothesis,

$$\forall\, \sigma'' \in States(\Pi) \mid \sigma''(i).prog = es_1 \bullet post(\sigma'')[i].$$

Hence $a : \{pre\}e_1 e_2\{post\}$ as required. □

We also give a rule concerning conjunctions of postconditions; this rule allows us to verify conjuncts of a postcondition separately.

Annotation Rule 4 (Conjunction of postconditions)

$$\frac{\begin{array}{l} a : \{pre\}e\{post_1\} \\ a : \{pre\}e\{post_2\} \end{array}}{a : \{pre\}e\{post_1 \wedge post_2\}}$$

Proof: Suppose

$$\begin{array}{l}\sigma_0(i).id = a \wedge \sigma_0(i).prog = es_0 \frown e' \frown es_1 \wedge e' \sqsupseteq e \wedge \\ \forall\, \sigma \in States(\Pi) \mid \sigma(i).prog = e' \frown es_1 \bullet pre(\sigma)[i].\end{array}$$

Then by the first hypothesis

$$\forall\, \sigma' \in States(\Pi) \mid \sigma'(i).prog = es_1 \bullet post_1(\sigma')[i],$$

and by the second hypothesis,

$$\forall\, \sigma' \in States(\Pi) \mid \sigma'(i).prog = es_1 \bullet post_2(\sigma')[i].$$

Hence

$$\forall\, \sigma' \in States(\Pi) \mid \sigma'(i).prog = es_1 \bullet (post_1 \wedge post_2)(\sigma')[i],$$

and so $a : \{pre\}e\{post_1 \wedge post_2\}$. □

4.3 Annotation macros

In this section we give semantics to the annotation macros that we introduced informally earlier.

4.3.1 *knows*

The macro $knows(x)$ returns the set of participants who know the value of x. Recall that assertions are interpreted with respect to a particular state, say state σ, and a particular node, say node i; therefore the value of x in question is $\sigma(i).\rho(x)$. This value is obtained via the substitution:

$$knows(x)(\sigma)[i] = knows(x)[\sigma(i).\rho](\sigma) = knows(\sigma(i).\rho(x))(\sigma).$$

We therefore define the meaning of *knows* with respect to a *value* X (as opposed to a variable).

Recall that if $x \notin \operatorname{dom} \sigma(i).\rho$, but $x^{-1} \in \operatorname{dom} \sigma(i).\rho$ then $\sigma(i).\rho(x)$ is shorthand for $(\sigma(i).\rho(x^{-1}))^{-1}$, so in this case $knows(x)$ represents the agents who know the decrypting key corresponding to encryptions using the value of x^{-1}.

For later convenience, we start by defining a function $knows_{id}(X)$ that gives the set of node identifiers where X is known. X is known by honest node i if $\sigma(i).\rho(y) = X$ for some y; X is known by the intruder if $\sigma(0) \vdash X$.

$$\begin{array}{rcl} knows_{id}(X)(\sigma) & \mathrel{\widehat=} & \{i \mid i \in 1 \mathinner{..} n \wedge \exists\, y \bullet \sigma(i).\rho(y) = X\} \\ & & \cup \\ & & (\text{if } \sigma(0) \vdash X \text{ then } \{0\} \text{ else } \{\}).\end{array}$$

We now define the function $knows(X)$ which returns the corresponding set of agent identities:

$$knows(X)(\sigma) \mathrel{\widehat{=}} \\ \{\text{if } i > 0 \text{ then } \sigma(i).\rho(\sigma(i).id) \text{ else } intruder \mid i \in knows_{id}(X)(\sigma)\}.$$

Equivalently, $knows(X)$ could have been defined by

$$\begin{aligned} knows(X)(\sigma) \;\mathrel{\widehat{=}}\; & \{\sigma(i).\rho(\sigma(i).id) \mid i \in 1 \mathinner{..} n \wedge \exists\, y \bullet \sigma(i).\rho(y) = X\} \\ & \cup \\ & (\text{if } \sigma(0) \vdash X \text{ then } \{intruder\} \text{ else } \{\}). \end{aligned}$$

Note that the value of $knows(X)$ cannot, in general, be relied upon to stay the same from one state to another, even if the agent currently being considered does not perform any events: messages sent elsewhere may cause new agents to learn X. However, the value of $knows(X)$ cannot decrease as an execution progresses.

The following annotation rule shows that $knows(x)$ does not change as a result of a receive event, provided the recipient already knows the value.

Annotation Rule 5 For every message m, and for every set as of variables representing identities such that $a \in as$:

$$a : \Big\{knows(x) = as\Big\}\ \mathsf{receive}\ m\ \Big\{knows(x) = as\Big\}\,.$$

Proof: Suppose

$$\sigma_0(i).id = a \wedge \sigma_0(i).prog = es_0 \frown e' \frown es_1 \wedge e' \sqsupseteq \mathsf{receive}\ m \wedge \\ \forall\, \sigma \in States(\Pi) \mid \sigma(i).prog = e' \frown es_1 \bullet (knows(x) = as)(\sigma)[i].$$

Let σ' be such that $\sigma'(i).prog = es_1$, and let σ be the state immediately before the event corresponding to e'. Let

$$X = \sigma(i).\rho(x), \qquad A = \sigma(i).\rho(a), \qquad As = \sigma(i).\rho(as).$$

Let $\hat{\sigma} = \sigma' \oplus \{i \mapsto \sigma(i)\}$. Then $\hat{\sigma}$ is reachable by the same trace that led to σ' but without the event corresponding to e', so $\hat{\sigma} \in States(\Pi)$. Also, $\hat{\sigma}(i).prog = \sigma(i).prog = e' \frown es_1$, so by the assumption corresponding to the precondition, $(knows(x) = as)(\hat{\sigma})[i]$, i.e.:

$$knows(X)(\hat{\sigma}) = As.$$

For every $j \neq i$, $j \in knows_{id}(X)(\sigma') \Leftrightarrow j \in knows_{id}(X)(\hat{\sigma})$, because $\sigma'(j) = \hat{\sigma}(j)$; and so for every $B \neq A$, $B \in knows(X)(\sigma') \Leftrightarrow B \in knows(X)(\hat{\sigma})$. And $A \in knows(X)(\sigma')$ and $A \in knows(X)(\hat{\sigma})$ by the precondition and the assumption that $a \in as$. Hence $knows(X)(\sigma') = knows(X)(\hat{\sigma}) = As$, i.e. $(knows(x) = as)(\sigma')$, as required. $\square$

4.3.2 *holds*

It is useful to define a macro $holds(X)$ that gives the identities of those agents who have the atomic value X as a submessage of one of the messages they know:

$$\begin{aligned} holds(X)(\sigma) \;\widehat{=}\; & \{\sigma(i).\rho(\sigma(i).id) \mid \exists\, y \bullet X \sqsubset \sigma(i).\rho(y)\} \\ & \cup \\ & (\text{if } \exists\, M \in \sigma(0) \bullet X \sqsubset M \text{ then } \{intruder\} \text{ else } \{\}). \end{aligned}$$

Note that $hold(X)$ includes those agents who hold X, even as a submessage by contrast with $knows(X)$, where X must equal all of a message stored by the agent. We have $knows(X)(\sigma) \subseteq holds(X)(\sigma)$.

The following lemma shows how i can acquire X:

Lemma 5 If

$$\sigma \xrightarrow{j:E} \sigma' \wedge A \notin holds(X)(\sigma) \wedge A \in holds(X)(\sigma') \wedge \sigma(j).\rho(\sigma(j).id) = B$$

then

$$\begin{aligned} & (E = \mathsf{new}\, X \vee \exists\, Y \bullet E = \mathsf{newpair}(X, Y) \vee E = \mathsf{newpair}(Y, X)) \wedge \\ & \quad A = B \\ & \vee \\ & \exists\, M \bullet E = \mathsf{send}\, M \wedge A = intruder \wedge X \sqsubset M \\ & \vee \\ & \exists\, M \bullet E = \mathsf{receive}\, M \wedge A = B \wedge X \sqsubset M. \end{aligned}$$

Proof: Direct from the operational semantics. □

4.3.3 *session*

If B is an honest agent then the notation

$$session(b \rightsquigarrow B;\; x_1 \rightsquigarrow X_1, \ldots, x_k \rightsquigarrow X_k)(\sigma)$$

means that for some node i, the variable representing the agent's identity is b, that b is bound to B, and each x_j is bound to X_j. If B is dishonest then the notation means that B knows each of the X_j: a dishonest agent is not forced to bind values to variables in any predictable way.

$$\begin{aligned} & session(b \rightsquigarrow B;\; x_1 \rightsquigarrow X_1, \ldots, x_k \rightsquigarrow X_k)(\sigma) \;\widehat{=} \\ & \quad \exists\, i > 0 \bullet \sigma(i).id = b \wedge \sigma(i).\rho(b) = B \wedge \forall\, j \in 1 \mathinner{..} k \bullet \sigma(i).\rho(x_j) = X_k \\ & \quad \vee \\ & \quad B = intruder \wedge \forall\, j \in 1 \mathinner{..} k \bullet \sigma(0) \vdash X_j. \end{aligned}$$

Recall that an assertion P is interpreted with respect to a particular node, say node i, via the substitution $P(\sigma)[i] = P[\sigma(i).\rho](\sigma)$. In the case of the *session* macro, we define this substitution to be performed only on variables on the right hand side of $\rightsquigarrow$ symbols, not those on the left hand side. For example,

$$
\begin{aligned}
&\mathit{session}(b \rightsquigarrow c;\ x \rightsquigarrow y)(\sigma)[i] = \\
&\quad \mathit{session}(b \rightsquigarrow \sigma(i).\rho(c);\ x \rightsquigarrow \sigma(i).\rho(y))(\sigma),
\end{aligned}
$$

i.e. the other node's b variable is bound to the value of node i's c variable, and the other node's x variable is bound to the value of node i's y variable. We will extend this convention — that substitution does not apply on the left of $\rightsquigarrow$ symbols — to other annotation macros later.

Often the value of a variable, x say, in one agent's state, say B's state, will match the value of the variable of the same name in the current scope; if the current annotation is from the point of view of agent A, then this means that A's value of x is the same as B's value of x. In such cases we simplify the binding "$x \rightsquigarrow x$" to just "x", representing that from A's point of view, B has x bound to the correct variable. We adopt the same convention with the identity variable. For example,

$$
\begin{aligned}
\mathit{session}(b;\ x)(\sigma)[i] &\equiv \mathit{session}(b \rightsquigarrow b;\ x \rightsquigarrow x)(\sigma)[i] \\
&\equiv \mathit{session}(b \rightsquigarrow \sigma(i).\rho(b);\ x \rightsquigarrow \sigma(i).\rho(x))(\sigma).
\end{aligned}
$$

The following lemma relates the *session* and *knows* macros.

Lemma 6

$$
\mathit{session}(a \rightsquigarrow b;\ x \rightsquigarrow y)(\sigma)[i] \ \Rightarrow\ \sigma(i).\rho(b) \in \mathit{knows}(\sigma(i).\rho(y))(\sigma).
$$

4.3.4 *honest*

The predicate $\mathit{honest}(X)$ asserts that the set of participants in X are honest in the sense that they do not deviate from the protocol definition:

$$
\mathit{honest}(X) \ \widehat{=}\ \mathit{intruder} \notin X.
$$

Note that if $\mathit{honest}(X)$ holds, then it will hold throughout an execution as an invariant. We simplify notation and write, for example, $\mathit{honest}(a, b)$ as a shorthand for $\mathit{honest}(\{a, b\})$.

4.3.5 *associatedWith*

We will sometimes want to say that particular values are associated with one another, so that if an agent receives one, then he must also receive the others (one could say that the values are bound together; we avoid that term because we are using the word "binding" in a different sense). We write $\mathit{associatedWith}_{x \rightsquigarrow X}(y_1 \rightsquigarrow Y_1, \ldots, y_n \rightsquigarrow Y_n)(b)$ to indicate that if agent b has X stored in variable x, then he has $Y_1, \ldots, Y_n$ stored in variables $y_1, \ldots, y_n$:

$$\begin{array}{l} \mathit{associatedWith}_{x \rightsquigarrow X}(y_1 \rightsquigarrow Y_1, \ldots, y_n \rightsquigarrow Y_n)(b)(\sigma) \mathrel{\widehat{=}} \\ \quad \forall\, j > 0 \mid \sigma(j).id = b \bullet \\ \qquad \sigma(j).\rho(x) = X \Rightarrow \sigma(j).\rho(y_1) = Y_1 \wedge \ldots \wedge \sigma(j).\rho(y_n) = Y_n. \end{array}$$

We drop the "b" to indicate that the association holds for all roles:

$$\begin{array}{l} \mathit{associatedWith}_{x \rightsquigarrow X}(y_1 \rightsquigarrow Y_1, \ldots, y_n \rightsquigarrow Y_n)(\sigma) \mathrel{\widehat{=}} \\ \quad \forall\, b \bullet \mathit{associatedWith}_{x \rightsquigarrow X}(y_1 \rightsquigarrow Y_1, \ldots, y_n \rightsquigarrow Y_n)(b)(\sigma). \end{array}$$

Within annotations, we will use the shorthand

$$\begin{array}{l} \mathit{associatedWith}_{x}(y_1, \ldots, y_n) \mathrel{\widehat{=}} \\ \qquad \mathit{associatedWith}_{x \rightsquigarrow x}(y_1 \rightsquigarrow y_1, \ldots, y_n \rightsquigarrow y_n). \end{array}$$

Recall the convention that substitution does not apply on the left right of the $\rightsquigarrow$ symbol; hence $\mathit{associatedWith}_x(y_1, \ldots, y_n)(\sigma)[i]$ means that if any other node j has x bound to the same value as i does, then j also has $y_1, \ldots, y_n$ bound to the same values as i does; in other words, i's value for x is associated inseparably with its values for $y_1, \ldots, y_n$.

The following lemma relates *associatedWith* to the *session* macro:

Lemma 7

$$\begin{array}{l} \left(\begin{array}{l} \mathit{session}(b \rightsquigarrow B;\ x \rightsquigarrow X, y_1 \rightsquigarrow Y_1, \ldots, y_m \rightsquigarrow Y_m) \wedge \\ \mathit{honest}(B) \wedge \mathit{associatedWith}_{x \rightsquigarrow X}(z_1 \rightsquigarrow Z_1, \ldots, z_n \rightsquigarrow Z_n) \end{array}\right) \Rightarrow \\ \quad \mathit{session}(b \rightsquigarrow B;\ x \rightsquigarrow X, y_1 \rightsquigarrow Y_1, \ldots, y_m \rightsquigarrow Y_m, \\ \qquad\qquad\qquad z_1 \rightsquigarrow Z_1, \ldots, z_n \rightsquigarrow Z_n). \end{array}$$

4.3.6 *uniquelyBound*

The annotation macro $\mathit{uniquelyBound}(x \rightsquigarrow X)$ means that the value X is bound only to the variable x:

$$\begin{array}{l} \mathit{uniquelyBound}(x \rightsquigarrow X)(\sigma) \mathrel{\widehat{=}} \\ \quad \forall\, j > 0;\ y \in \mathit{Var} \bullet \sigma(j).\rho(y) = X \Rightarrow y = x. \end{array}$$

We define the standard shorthand:

$$\mathit{uniquelyBound}(x) \mathrel{\widehat{=}} \mathit{uniquelyBound}(x \rightsquigarrow x).$$

Recall the convention that substitution does not apply on the left of the $\rightsquigarrow$ symbol; hence $\mathit{uniquelyBound}(x)(\sigma)[i]$ means that i's value for x is bound only to the variable x in other nodes.

4.3.7 *defined*

We say that a value X is *defined* if it is not the special value $\perp$:

$$\mathit{defined}(X) \;\widehat{=}\; X \neq \perp.$$

Note that

$$\mathit{defined}(x)(\sigma)[i] \;\equiv\; x \in \mathrm{dom}(\sigma(i).\rho).$$

4.3.8 Always

If P is a predicate, then $\Box\, P$ (pronounced "always P") represents that P holds in this and all subsequent states:

$$\Box\, P(\sigma) \;\widehat{=}\; \forall\, \sigma', \mathit{tr} \mid \sigma \overset{\mathit{tr}}{\Longrightarrow} \sigma' \bullet P(\sigma').$$

The following lemmas consider properties of this definition.

Lemma 8 $\Box\, P \Rightarrow P$.

Proof: This follows immediately from the definition, because $\sigma \overset{\langle\rangle}{\Longrightarrow} \sigma$.

$\Box$

Lemma 9 If $\mathit{vars}(P) \subseteq \mathrm{dom}\,\sigma(i).\rho$ then then

$$(\Box\, P)(\sigma)[i] \;\equiv\; \forall\, \sigma' \mid \sigma \Longrightarrow \sigma' \bullet P(\sigma')[i].$$

Proof:

$$\begin{array}{ll}
 & (\Box\, P)(\sigma)[i] \\
\equiv & (\Box(P[\sigma(i).\rho]))(\sigma) \\
\equiv & \forall\, \sigma' \mid \sigma \Longrightarrow \sigma' \bullet P[\sigma(i).\rho](\sigma') \\
\equiv & \left\langle \begin{array}{l} \mathit{vars}(P) \subseteq \mathrm{dom}\,\sigma(i).\rho, \text{ so } \sigma(i).\rho \\ \text{and } \sigma'(i).\rho \text{ agrees on those variables (Lemma 1)} \end{array} \right\rangle \\
 & \forall\, \sigma' \mid \sigma \Longrightarrow \sigma' \bullet P[\sigma'(i).\rho](\sigma') \\
\equiv & \forall\, \sigma' \mid \sigma \Longrightarrow \sigma' \bullet P(\sigma')[i].
\end{array}$$

$\Box$

Note that the condition of the lemma is necessary: if $P \;\widehat{=}\; x = \perp$ and $x \notin \mathrm{dom}\,\sigma(i).\rho$ then $(\Box\, P)(\sigma)[i] \equiv \Box(\perp = \perp)(\sigma)[i] \equiv \mathit{true}$; but $P(\sigma')[i]$ will not hold in some subsequent state σ' if x is bound in the meantime.

The following rule allows assertions of the form $\Box\, P$ to be carried forward through annotations.

Annotation Rule 6 For all events e and predicates P

$$a : \left\{ \mathit{defined}(\mathit{vars}(P)) \wedge \Box\, P \right\} e \left\{ \Box\, P \right\}$$

Proof: Suppose

$$\sigma_0(i).id = a \wedge \sigma_0(i).prog = es_0 \frown e' \frown es_1 \wedge e' \sqsupseteq e \wedge \\ \forall \sigma \in States(\Pi) \mid \sigma(i).prog = e' \frown es_1 \bullet (defined(vars(P)) \wedge \Box P)(\sigma)[i].$$

Suppose σ' is such that $\sigma'(i).prog = es_1$ and suppose $\sigma' \Longrightarrow \sigma''$; we need to show $P(\sigma'')[i]$. Let σ be the state immediately before the event corresponding to e'; then $\sigma(i).prog = e' \frown es_1$ and $\sigma \Longrightarrow \sigma''$. Then from the assumption corresponding to the precondition, $(defined(vars(P)) \wedge \Box P)(\sigma)[i]$. The first conjunct is equivalent to $vars(P) \subseteq \operatorname{dom} \sigma(i)$. Hence from Lemma 9, $P(\sigma'')[i]$. Hence $(\Box P)(\sigma')[i]$, as required. $\Box$

4.4 new x

In this section we verify the proof rule for new x given earlier.

Annotation Rule 7 (New) If *pre* refers only to state variables then

$$a : \left\{pre\right\} \ \mathsf{new}\, x \ \left\{knows(x) = \{a\} \wedge (\exists X_0 \bullet pre[X_0/x])\right\}$$

where X_0 is a fresh identifier.

Note that the restriction on *pre* is necessary to prevent preconditions such as $\#\rho = 3$, which would not be preserved by the creation of a new variable within ρ. Note also that if x is not free in *pre* then the second conjunct of the postcondition simplifies to *pre*.
Proof: Suppose

$$\sigma_0(i).id = a \wedge \sigma_0(i).prog = es_0 \frown e' \frown es_1 \wedge e' \sqsupseteq \mathsf{new}\, x \wedge \\ \forall \sigma \in States(\Pi) \mid \sigma(i).prog = e' \frown es_1 \bullet pre(\sigma)[i].$$

Then $e' = \mathsf{new}\, x$. Suppose σ' is such that $\sigma'(i).prog = es_1$, and let σ be the state immediately before the new x event. Then $pre(\sigma)[i]$ and $\sigma'(i).\rho = \sigma'(i) \oplus \{x \mapsto X\}$ for some fresh X. Let σ'' be the global state immediately after the transition, which might not be the same as σ' at nodes other than i; then:

$$\sigma \xrightarrow{i:\mathsf{new}\, X} \sigma'' \longrightarrow^* \sigma' \wedge \sigma''(i) = \sigma'(i) \wedge \\ isNew(X)(\sigma) \wedge \forall j \neq i \bullet \sigma(j) = \sigma''(j).$$

We consider the two conjuncts of the postcondition separately. For the first conjunct, we need to show

$$(knows(x) = \{a\})(\sigma')[i] \ \equiv \ knows(X)(\sigma') = \{A\},$$

where $A = \sigma(i).\rho(a)$. Clearly $(knows(X) = \{A\})(\sigma'')$ because X is fresh:

$$\begin{aligned} isNew(X)(\sigma) &\Rightarrow \forall B \bullet B \notin holds(X)(\sigma) \\ &\Rightarrow \forall B \neq A \bullet B \notin holds(X)(\sigma''). \end{aligned}$$

But node i sends no messages between σ'' and σ', and so

$$\forall B \neq A \bullet B \notin holds(X)(\sigma')$$

from Lemma 5 and a simple case analysis. Hence $knows(X)(\sigma') = \{A\}$.

For the second conjunct, we need to show $(\exists X_0 \bullet pre[X_0/x])(\sigma')[i]$. Let $\hat{\sigma} = \sigma' \oplus \{i \mapsto \sigma(i)\}$. Then it is clear that $\hat{\sigma} \in States(\Pi)$, reachable via the same trace that reached σ' except excluding the $i : \mathsf{new}\, X$ event. Further, $\hat{\sigma}(i).prog = \mathsf{new}\, x \frown es_1$. Hence by the hypothesis of the rule, $pre(\hat{\sigma})[i]$. But

$$
\begin{array}{ll}
 & pre(\hat{\sigma})[i] \\
\equiv & \left\langle \text{definition} \right\rangle \\
 & pre[\hat{\sigma}(i).\rho](\hat{\sigma}) \\
\Rightarrow & \left\langle \text{predicate calculus} \right\rangle \\
 & (\exists X_0 \bullet pre[X_0/x])[\hat{\sigma}(i).\rho](\hat{\sigma}) \\
\equiv & \left\langle x \text{ not free in } \exists X_0 \bullet pre[X_0/x] \right\rangle \\
 & (\exists X_0 \bullet pre[X_0/x])[\sigma'(i).\rho](\hat{\sigma}) \\
\equiv & \left\langle x \text{ not free in } \exists X_0 \bullet pre[X_0/x];\ pre \text{ refers only to state variables} \right\rangle \\
 & (\exists X_0 \bullet pre[X_0/x])[\sigma'(i).\rho](\sigma') \\
\equiv & \left\langle \text{definition} \right\rangle \\
 & (\exists X_0 \bullet pre[X_0/x])(\sigma')[i].
\end{array}
$$

□

We believe a similar rule holds for newpair; verifying it is left as future work.

5 Disjoint encryption

In this section we define the disjoint encryption property [GTF00]: that different encrypted components within the protocol have distinct forms. We then prove a theorem that follows from it.

We start by extending the submessage relation to *Template* $\leftrightarrow$ *Event-Template* in the obvious way:

$$
\begin{aligned}
m \sqsubset \mathsf{send}\, m' &\Leftrightarrow m \sqsubset m', \\
m \sqsubset \mathsf{receive}\, m' &\Leftrightarrow m \sqsubset m.
\end{aligned}
$$

We now capture the disjoint encryption assumption.

Definition 4 (Disjoint encryption) Suppose in the initial state σ_0, the j_1th message of the program at node i_1 and the j_2th message of the program at node i_2 both contain encrypted submessages that have the same type:

$$\{m_1\}_{k_1} \sqsubset \sigma_0(i_1).prog(j_1) \wedge \{m_2\}_{k_2} \sqsubset \sigma_0(i_2).prog(j_2) \wedge$$
$$type(\{m_1\}_{k_1}) = type(\{m_2\}_{k_2}).$$

Then these two encrypted components are, in fact, syntactically equal components:

$$\{m_1\}_{k_1} = \{m_2\}_{k_2}.$$

Guttman and Thayer [GTF00] consider the idea of disjoint encryption in the context of two protocols operating in the same environment: their property specified that the protocols should not have encrypted components of the same form (i.e. type); they prove that in this case the two protocols are independent, i.e. there are no interactions between them. Our definition is slightly weaker (and in a slightly different context): two encrypted components may have the same form, but if they do, they should use the same variables.

We now prove a result that shows that, under certain circumstances, all occurrences of a value X in honest agents' states are bound to the same variable x.

We will need the following lemma which says that if the intruder can deduce a message containing $\{M\}_K$, then either he knows both M and K (so can perform the encryption), or he knows a message containing $\{M\}_K$:

Lemma 10 If $B \vdash M' \wedge \{M\}_K \sqsubset M'$ then $B \vdash M \wedge B \vdash K$ or $\{M\}_K \sqsubset B$.

Proof: Straightforward rule induction. $\square$

We now prove the result alluded to above.

Theorem 1 Suppose:

1. The protocol satisfies the disjoint encryption property.
2. The intruder did not initially hold any message containing X:

   $$X \not\sqsubset \sigma_0(0).$$

3. Any honest agent who held X initially had it bound to x:

   $$\mathit{uniquelyBound}(x \rightsquigarrow X)(\sigma_0).$$

   And further X is not held as a proper submessage of any variable:

   $$\forall\, i > 0;\ y \in \operatorname{dom} \sigma_0(i).\rho \bullet X \sqsubset \sigma_0(i).\rho(y) \Rightarrow X = \sigma_0(i).\rho(y)$$

   (y will necessarily equal x if both sides of the implication hold).
4. Trace tr ends in state σ_1, where the intruder does not know X:

   $$\text{last}\, tr = \sigma_1 \wedge \sigma_1(0) \nvdash X.$$

5. If X is generated in a new or $\mathsf{newpair}$ event, then it is generated to instantiate x:

$$\begin{array}{l}
\forall\, tr' \frown \langle \sigma, i : \mathsf{new}\, X, \sigma' \rangle \leq tr \bullet \sigma(i).prog = \langle \mathsf{new}\, x \rangle \frown \sigma'(i).prog \\
\land \\
\forall\, tr' \frown \langle \sigma, i : \mathsf{newpair}(X, Y), \sigma' \rangle \leq tr \bullet \\
\quad \exists\, y \bullet \sigma(i).prog = \langle \mathsf{newpair}(x, y) \rangle \frown \sigma'(i).prog \\
\land \\
\forall\, tr' \frown \langle \sigma, i : \mathsf{newpair}(Y, X), \sigma' \rangle \leq tr \bullet \\
\quad \exists\, y \bullet \sigma(i).prog = \langle \mathsf{newpair}(y, x) \rangle \frown \sigma'(i).prog.
\end{array}$$

Then any honest agent who holds X does so with it bound to the variable x:

$$uniquelyBound(x \rightsquigarrow X)(\sigma_1). \tag{1}$$

Note that this theorem really concerns two quite different scenarios:

- If an honest agent does hold X initially (necessarily bound to x), then no new or $\mathsf{newpair}$ events for X can occur (because of the freshness condition on such events), and so assumption 5 holds vacuously.
- If no honest agent holds X initially, then it must be introduced (if at all) by a $\mathsf{new}\, x$, $\mathsf{newpair}(x, y)$ or $\mathsf{newpair}(y, x)$ event. The theorem then gives a result about *all* values X that could be introduced for x. Note that in this case, assumptions 2, 3 and 5 are automatically satisfied.

Proof: Suppose, for a contradiction, that the result does not hold. Consider the shortest counter-example trace tr. By assumption 3, tr is not the trivial trace $\langle \sigma_0 \rangle$. So consider the last event of tr, and perform a case analysis:

- Case $i : \mathsf{new}\, Y$. new events change bindings only for the node and variable in question. Hence the only way that equation (1) can be falsified by this event is if it is a $\mathsf{new}\, X$ event for a variable $y \neq x$. But this contradicts assumption 5.
- Case $i : \mathsf{newpair}(Y, Z)$. This is very similar to the previous case.
- Case $i : \mathsf{send}\, M$. send events do not change any bindings, so cannot falsify equation (1).
- Case $i : \mathsf{receive}\, M$. The intruder does not know X in the final state, so it cannot appear as plaintext in M; a variable is bound to X as a result of the event, and no variables are bound as the result of hashes, so X cannot occur only within a hash; hence X must appear encrypted in M, instantiating a variable other than x.

  Consider the smallest encrypted component of M containing the occurrence of X that gets mis-bound, say $\{M_1\}_K$, with $X \ll M_1$, instantiating template $\{m_1\}_k$. Now $\sigma_1(0) \not\vdash M_1$ because $\sigma_1(0) \not\vdash X$. Hence by Lemma 10, $\{M_1\}_K \preceq \sigma_1(0)$.

Now consider the earliest point in the trace at which $\{M_1\}_K \preceq \sigma(0)$. This was not true in the initial state by assumption 2. Hence it must have come about as the result of an event $j : \mathsf{send}\ M'$ with $\{M_1\}_K \preceq M'$. Now, j cannot have had $\{M_1\}_K$ stored within a variable in the initial state by the second part of assumption 3; and cannot have stored $\{M_1\}_K$ within a variable as the result of a receive event, for no earlier event has included $\{M_1\}_K$; hence j must have constructed this encrypted component, say to instantiate template $\{m'_1\}_{k'}$. By the presumed minimality of the counterexample tr, node j has X bound only to x in this state, so X instantiates only x in $\{m'_1\}_{k'}$.

Then $\mathit{type}(\{m_1\}_k) = \mathit{type}(\{m'_1\}_{k'})$, so by the disjoint encryption assumption, $\{m\}_k = \{m'\}_{k'}$. Hence X must instantiate the same variables of $\{m\}_k$ in the $\mathsf{receive}\ M$ event as it does of $\{m'\}_{k'}$ in the $\mathsf{send}\ M'$ event, namely just x. This gives a contradiction.

□

6 Abstract messages

In this section we consider abstract messages in more detail. We consider each of the atomic abstract messages from Section 2, as well as some additional useful ones; we also consider concrete messages as abstract messages, and the conjunction of abstract messages. For each of the constructs, we give a formal semantics, proof rules governing how it can be used in annotations, and rules showing how it can be refined to a concrete message.

6.1 Refinement

Recall that the semantics of an abstract message am in protocol Π is the set of message templates that could be used to implement am, written $[\![am]\!]_\Pi$. Recall also that we write $am \sqsubseteq_\Pi am'$ if abstract message am can be implemented by am' in the context of protocol Π:

$$am \sqsubseteq_\Pi am' \;\Leftrightarrow\; [\![am]\!]_\Pi \supseteq [\![am']\!]_\Pi.$$

We drop the subscript Π when it is clear from the context.

The following lemma follows directly from the definition.

Lemma 11 Refinement is a preorder.

The following rules show how refinement can be used within annotations: refining a sent or received message preserves the correctness of an annotation.

Annotation Rule 8 (Refine sent message)

$$\frac{\begin{array}{l} a : \{pre\}\,\mathsf{send}\; m\{post\} \\ m \sqsubseteq_{\Pi} m' \end{array}}{a : \{pre\}\,\mathsf{send}\; m'\{post\}}$$

Proof: Suppose

$$\begin{array}{l} \sigma_0(i).id = a \wedge \sigma_0(i).prog = es_0 \frown e' \frown es_1 \wedge e' \sqsupseteq \mathsf{send}\; m' \wedge \\ \forall\, \sigma \in States(\Pi) \mid \sigma(i).prog = e' \frown es_1 \bullet pre(\sigma)[i]. \end{array}$$

Then $e' \sqsupseteq \mathsf{send}\; m$ by the second hypothesis. So by the first hypothesis,

$$\forall\, \sigma' \in States(\Pi) \mid \sigma'(i).prog = es_1 \bullet post(\sigma')[i],$$

Hence $\{pre\} a : \mathsf{send}\; m'\{post\}$. □

Annotation Rule 9 (Refine received message)

$$\frac{\begin{array}{l} b : \{pre\}\,\mathsf{receive}\; m\{post\} \\ m \sqsubseteq_{\Pi} m' \end{array}}{b : \{pre\}\,\mathsf{receive}\; m'\{post\}}$$

Proof: Similar to the previous proof. □

It is sometimes useful to talk about message refinements that hold under certain assumptions about the state from which the message is sent or received. We write $e \sqsubseteq \left\{pre\right\} e'$ to mean that under the assumption that e' is performed from a state where *pre* holds, it refines e. More precisely, the semantics of e' restricted to those messages sent of received from states where *pre* must hold (the set *ms*, below) is included in the semantics of e. For send events, the definition is as follows:

$$\begin{array}{l} a : \mathsf{send}\; am \sqsubseteq_{\Pi} \left\{pre\right\} \mathsf{send}\; am' \mathrel{\widehat{=}} \\ \quad \forall\, i > 0 \mid \sigma_0(i).id = a \bullet [\![am]\!]_{\Pi} \supseteq [\![am']\!]_{\Pi} \cap ms \\ \quad \text{where } ms = \{m \mid \forall\, tr \frown \langle \sigma,\, i : \mathsf{send}\; m\rangle \in traces(\Pi) \bullet pre(\sigma)[i]\}. \end{array}$$

The definition for receive events is essentially identical.

Lemma 12 If $a : \left\{pre\right\} e \left\{post\right\}$ is a valid annotation, and $e \sqsubseteq \left\{pre'\right\} e'$, then $a : \left\{pre \wedge pre'\right\} e' \left\{post\right\}$ is a valid annotation.

Proof: We consider just the case where $e = \mathsf{send}\, m$ and $e' = \mathsf{send}\, m'$; the case for receives is similar.

Following the definition of $a : \left\{pre \wedge pre'\right\} \mathsf{send}\, m' \left\{post\right\}$, suppose:

$$\sigma_0(i).id = a \wedge \sigma_0(i).prog = es_0 \frown \mathsf{send}\, m'' \frown es_1 \wedge m'' \sqsupseteq m' \wedge$$
$$\forall\, \sigma \in States(\Pi) \mid \sigma(i).prog = \mathsf{send}\, m'' \frown es_1 \bullet (pre \wedge pre')(\sigma)[i].$$

Then $m'' \in ms \mathrel{\widehat{=}} \{m_1 \mid \forall\, tr \frown \langle \sigma, i : \mathsf{send}\, m_1 \rangle \in traces(\Pi) \bullet pre'(\sigma)[i]\}$. Also, $m'' \in [\![m']\!]_\Pi$, by assumption. Hence $m'' \in [\![m]\!]_\Pi$, by the second assumption. Let σ' be such that $\sigma'(i).prog = es_1$; then by the first assumption, $post(\sigma')[i]$, and so $a : \left\{pre \wedge pre'\right\} \mathsf{send}\, m' \left\{post\right\}$, as required. □

6.2 Concrete messages

We consider concrete messages to be a particular type of abstract messages. The semantics of a concrete message is simply the singleton set containing the concrete message:

$$[\![x]\!]_\Pi \mathrel{\widehat{=}} \{x\}.$$

6.3 Conjunction

Abstract messages can be combined by conjunction: the conjoined abstract message represents the conjunction of the requirements of the components.

The semantics of a conjunction is the intersection of the semantics of the two components:

$$[\![m_1 \wedge m_2]\!]_\Pi \mathrel{\widehat{=}} [\![m_1]\!]_\Pi \cap [\![m_2]\!]_\Pi.$$

It is worth considering the case where $[\![m_1 \wedge m_2]\!]_\Pi = \{\}$, which is the case when m_1 and m_2 represent incompatible requirements. Such a specification is infeasible: it suggests that the protocol designer has made an error, leaving too many requirements in one abstract message.

The following lemma follows directly from the definition.

Lemma 13 Conjunction represents the least upper bound relation with respect to refinement.

The following two rules relate conjunction to refinement.

Refinement Rule 1 (Refinement by conjunction)

$$m \sqsubseteq m \wedge m'$$

Refinement Rule 2 (Conjunction of requirements)

$$\frac{\begin{array}{l} m_1 \sqsubseteq m \\ m_2 \sqsubseteq m \end{array}}{m_1 \wedge m_2 \sqsubseteq m}$$

From these and earlier rules, we can deduce the following corollaries.

Annotation Rule 10 (Conjunction of sent messages)

$$\frac{\begin{array}{l}\{pre\}\ \mathsf{send}\ m_1\{post_1\}\\ \{pre\}\ \mathsf{send}\ m_2\{post_2\}\end{array}}{\{pre\}\ \mathsf{send}\ m_1 \wedge m_2\{post_1 \wedge post_2\}}$$

Proof: From Refinement Rule 1, $m_1 \sqsubseteq m_1 \wedge m_2$. Hence from Annotation Rule 8 and the first hypothesis, $\{pre\}\ \mathsf{send}\ m_1 \wedge m_2\{post_1\}$. Similarly, $\{pre\}\ \mathsf{send}\ m_1 \wedge m_2\{post_2\}$. Hence by Annotation Rule 4,

$$\{pre\}\mathsf{send}\ m_1 \wedge m_2\{post_1 \wedge post_2\}.$$

□

Annotation Rule 11 (Conjunction of received messages)

$$\frac{\begin{array}{l}\{pre\}\ \mathsf{receive}\ m_1\{post_1\}\\ \{pre\}\ \mathsf{receive}\ m_2\{post_2\}\end{array}}{\{pre\}\ \mathsf{receive}\ m_1 \wedge m_2\{post_1 \wedge post_2\}}$$

Proof: Similar to the previous rule. □

6.4 *provesKnowledgeOf*

The abstract message *provesKnowledgeOf*(x) proves to the recipient of the message that some agent knows the recipient's value of x, and, if that agent is not the intruder, he has that value bound to his own variable x. This allows the receiver to verify state information about the sender concerning the variable x. *provesKnowledgeOf* specifies nothing about who may learn data from this message.

6.4.1 Semantics

The semantics of *provesKnowledgeOf*(x) is the set of messages m that if an instantiation is received by some node i (the instantiation in state σ' will be $m[\sigma'(i).\rho]$), then in the previous state σ, either the intruder knew i's value X for x, or some other honest node j had its x variable bound to X:

$$\begin{array}{l}[\![provesKnowledgeOf(x)]\!]_\Pi \mathrel{\widehat{=}}\\ \quad \{m \mid \forall\, tr \frown \langle \sigma, i : \mathsf{receive}\ m[\sigma'(i).\rho], \sigma' \rangle \in traces(\Pi) \bullet\\ \qquad\qquad \sigma(0) \vdash X \vee\\ \qquad\qquad \exists\, j > 0 \bullet j \neq i \wedge \sigma(j).\rho(x) = X\\ \qquad\qquad \text{where } X = \sigma'(i).\rho(x)\}.\end{array}$$

The following is an obvious extension:

$$
\begin{array}{l}
[\![\mathit{provesKnowledgeOf}(x_1, \ldots, x_k)]\!]_\Pi \mathrel{\widehat{=}} \\
\quad \{m \mid \forall\, tr \frown \langle \sigma, i : \mathsf{receive}\, m[\sigma'(i).\rho], \sigma' \rangle \in \mathit{traces}(\Pi) \bullet \\
\qquad\qquad (\forall\, l \in 1 \mathinner{..} k \bullet \sigma(0) \vdash X_l) \vee \\
\qquad\qquad \exists\, j > 0 \bullet j \neq i \wedge \forall\, l \in 1 \mathinner{..} k \bullet \sigma(j).\rho(x_l) = X_l \\
\qquad\qquad \text{where } X_l = \sigma'(i).\rho(x_l) \text{ for } l \in 1 \mathinner{..} k\}.
\end{array}
$$

Note that the abstract message $\mathit{provesKnowledgeOf}(x_1, \ldots, x_k)$ is not the same as $\mathit{provesKnowledgeOf}(x_1) \wedge \ldots \wedge \mathit{provesKnowledgeOf}(x_k)$: in the latter abstract message, it might be different agents who know the different x_l.

It is useful to define an extension of *provesKnowledgeOf* where the recipient receives evidence of the role played by the other agent; the abstract message $\mathit{provesKnowledgeOf}(x_1, \ldots, x_k, id = b)$ tells the recipient that the other agent was following a role with identity variable b:

$$
\begin{array}{l}
[\![\mathit{provesKnowledgeOf}(x_1, \ldots, x_k, id = b)]\!]_\Pi \mathrel{\widehat{=}} \\
\quad \{m \mid \forall\, tr \frown \langle \sigma, i : \mathsf{receive}\, m[\sigma'(i).\rho], \sigma' \rangle \in \mathit{traces}(\Pi) \bullet \\
\qquad\qquad (\forall\, l \in 1 \mathinner{..} k \bullet \sigma(0) \vdash X_l) \vee \\
\qquad\qquad \exists\, j > 0 \bullet j \neq i \wedge \sigma(j).id = b \wedge \forall\, l \in 1 \mathinner{..} k \bullet \sigma(j).\rho(x_l) = X_l \\
\qquad\qquad \text{where } X_l = \sigma'(i).\rho(x_l) \text{ for } l \in 1 \mathinner{..} k\}.
\end{array}
$$

The *provesKnowledgeOfNR* abstract messages are slightly stronger, as they give the recipient the additional guarantee that the message was not replayed from himself: the other node j has an identity different from the receiving node i:

$$
\begin{array}{l}
[\![\mathit{provesKnowledgeOfNR}(x_1, \ldots, x_k)]\!]_\Pi \mathrel{\widehat{=}} \\
\quad \{m \mid \forall\, tr \frown \langle \sigma, i : \mathsf{receive}\, m[\sigma'(i).\rho], \sigma' \rangle \in \mathit{traces}(\Pi) \bullet \\
\qquad\qquad (\forall\, l \in 1 \mathinner{..} k \bullet \sigma(0) \vdash X_l) \vee \\
\qquad\qquad \exists\, j > 0 \bullet \sigma(j).\rho(\sigma(j).id) \neq \sigma(i).\rho(\sigma(i).id) \wedge \\
\qquad\qquad\qquad\quad \forall\, l \in 1 \mathinner{..} k \bullet \sigma(j).\rho(x_l) = X_l \\
\qquad\qquad \text{where } X_l = \sigma'(i).\rho(x_l) \text{ for } l \in 1 \mathinner{..} m\}, \\
[\![\mathit{provesKnowledgeOfNR}(x_1, \ldots, x_k, id = b)]\!]_\Pi \mathrel{\widehat{=}} \\
\quad \{m \mid \forall\, tr \frown \langle \sigma, i : \mathsf{receive}\, m, \sigma' \rangle \in \mathit{traces}(\Pi) \bullet \\
\qquad\qquad (\forall\, l \in 1 \mathinner{..} k \bullet \sigma(0) \vdash X_l) \vee \\
\qquad\qquad \exists\, j > 0 \bullet \sigma(j).id = b \wedge \sigma(j).\rho(b) \neq \sigma(i).\rho(\sigma(i).id) \wedge \\
\qquad\qquad\qquad\quad \forall\, l \in 1 \mathinner{..} k \bullet \sigma(j).\rho(x_l) = X_l \\
\qquad\qquad \text{where } X_l = \sigma'(i).\rho(x_l) \text{ for } l \in 1 \mathinner{..} k\}.
\end{array}
$$

6.4.2 Annotation rules

The following proof rule shows how *provesKnowledgeOf* can be used in annotations.

Annotation Rule 12 (provesKnowledgeOf.1)

$$\begin{array}{l} a : \Big\{ \mathit{true} \Big\} \\ \quad \mathsf{receive}\ \mathit{provesKnowledgeOf}(x_1, \ldots, x_k) \\ \quad \Big\{ \exists\, b \in \mathit{Var};\ B \in \mathit{Val} \bullet \mathit{session}(b \rightsquigarrow B;\ x_1, \ldots, x_k) \Big\} \end{array}$$

Proof: Suppose

$$\begin{array}{l} \sigma_0(i).\mathit{id} = a \wedge \sigma_0(i).\mathit{prog} = \mathit{es}_0 \frown e' \frown \mathit{es}_1 \wedge \\ e' \sqsupseteq \mathsf{receive}\ \mathit{provesKnowledgeOf}(x_1, \ldots, x_k) \wedge \\ \forall\, \sigma \in \mathit{States}(\Pi) \mid \sigma(i).\mathit{prog} = e' \frown \mathit{es}_1 \bullet \mathit{true}(\sigma)[i]. \end{array}$$

Let σ' be such that $\sigma'(i).\mathit{prog} = \mathit{es}_1$, and let σ be the state immediately before the event corresponding to e'. Let $X_l = \sigma'(i).\rho(x_l)$ for $l \in 1..k$. Then, from the semantics of $\mathit{provesKnowledgeOf}(x)$, there are two possibilities:

- Case $\forall\, l \in 1 \ldots k \bullet \sigma(0) \vdash X_l$, so $\forall\, l \in 1 \ldots k \bullet \sigma'(0) \vdash X_l$. Then, for arbitrary $b \in \mathit{Var}$,

  $$\mathit{session}(b \rightsquigarrow \mathit{intruder};\ x_1 \rightsquigarrow X_1, \ldots, x_k \rightsquigarrow X_k)(\sigma')$$

  from the definition of *session*. So

  $$\exists\, b, B \bullet \mathit{session}(b \rightsquigarrow B;\ x_1 \rightsquigarrow X_1, \ldots, x_k \rightsquigarrow X_k)(\sigma'),$$

  and so

  $$(\exists\, b, B \bullet \mathit{session}(b \rightsquigarrow B;\ x_1 \rightsquigarrow x_1, \ldots, x_k \rightsquigarrow x_k))(\sigma')[i],$$

  as required.
- Case for some $j > 0,\ j \neq i \wedge \forall\, l \in 1 \ldots k \bullet \sigma(j).\rho(x_l) = X_l$. Let $b = \sigma(j).\mathit{id}$ and $B = \sigma(j).\rho(b)$. Then $\mathit{session}(b \rightsquigarrow B;\ x_1 \rightsquigarrow X_1, \ldots, x_k \rightsquigarrow X_k)(\sigma)$ and so $\mathit{session}(b \rightsquigarrow B;\ x_1 \rightsquigarrow X_1, \ldots, x_k \rightsquigarrow X_k)(\sigma')$. The proof then proceeds as in the previous case.

□

The following three rules show how the variants of *provesKnowledgeOf* give the recipient extra information about the other agent.

Annotation Rule 13 (provesKnowledgeOf.2)

$$\begin{array}{l} a : \Big\{ \mathit{true} \Big\} \\ \quad \mathsf{receive}\ \mathit{provesKnowledgeOf}(x_1, \ldots, x_k, \mathit{id} = b) \\ \quad \Big\{ \exists\, B \in \mathit{Val} \bullet \mathit{session}(b \rightsquigarrow B;\ x_1, \ldots, x_k) \Big\} \end{array}$$

Proof: This is a straightforward adaptation of the proof of the previous rule. □

Annotation Rule 14 (provesKnowledgeOfNR.1)

$$
\begin{array}{l}
a : \Big\{ \mathit{true} \Big\} \\
\quad \textsf{receive}\ \mathit{provesKnowledgeOfNR}(x_1, \ldots, x_k) \\
\quad \Big\{ \exists\, b \in \mathit{Var};\ B \in \mathit{Val} \bullet \mathit{session}(b \rightsquigarrow B;\ x_1, \ldots, x_k) \wedge B \neq a \Big\}
\end{array}
$$

Proof: This is a straightforward adaptation of the proof of Rule 12. In the first case, $B = \mathit{intruder} \neq \sigma'(i).\rho(a)$; but this is equivalent to $(B \neq a)(\sigma')[i]$, as required. In the second case, we have the additional condition that $\sigma(j).\rho(\sigma(j).id) \neq \sigma(i).\rho(\sigma(i).id)$, which is equivalent to $B \neq \sigma'(i).\rho(a)$, i.e. $(B \neq a)(\sigma')[i]$, as required. □

Annotation Rule 15 (provesKnowledgeOfNR.2)

$$
\begin{array}{l}
a : \Big\{ \mathit{true} \Big\} \\
\quad \textsf{receive}\ \mathit{provesKnowledgeOfNR}(x_1, \ldots, x_k, id = b) \\
\quad \Big\{ \exists\, B \in \mathit{Val} \bullet \mathit{session}(b \rightsquigarrow B;\ x_1, \ldots, x_k) \wedge B \neq a \Big\}
\end{array}
$$

Proof: Straightforward adaptation of the previous two proofs. □

6.4.3 Refinement rules

We now state and prove some refinement rules for *provesKnowledgeOf*. We begin by showing that, subject to some provisos, if an encrypted message contains all the elements of xs, either as direct sub-messages or as the encrypting key, then that message refines $\mathit{provesKnowledgeOf}(xs)$. For example, subject to the provisos

$$\mathit{provesKnowledgeOf}(x, y, z) \sqsubseteq \{x, y\}_z.$$

Further, any message containing such an encrypted component, possibly with additional fields, will refine the same abstract message. We will need the following lemma.

Lemma 14 If $B \vdash M \wedge M' \ll M$ then $B \vdash M'$.

Refinement Rule 3 Suppose message template m is such that for some encrypted message template $m' = \{m''\}_y \sqsubset m$,

$$\forall\, x \in xs \bullet x \ll m'' \vee x = y.$$

Then

$$\mathit{provesKnowledgeOf}(xs) \sqsubseteq_{\Pi} m,$$

provided:

1. the protocol satisfies the disjoint encryption property;
2. no node sends and then receives an instantiation of m'; and
3. either (a) at least one field of m' is freshly generated by the recipient; or (b) the intruder does not initially know any instantiation of m' unless he also knows the direct submessages and the encrypting key.

Proof: Let $xs = \{x_1, \ldots, x_k\}$. Following the definition of *provesKnowledgeOf*, consider a trace

$$tr \frown \langle \sigma, i : \mathsf{receive}\, m[\sigma'(i).\rho], \sigma' \rangle.$$

Let $M' = m'[\sigma'(i).\rho]$, and let $X_l = \sigma'(i).\rho(x_l)$ for $l \in 1..k$. Consider the first message of the trace that contains M' (which might be the above message):

- Case $j : \mathsf{send}\, M$. Suppose this event occurs from state σ''. Then by the disjoint encryption property, the component of M that equals M' must itself instantiate m', so

  $$M' = m'[\sigma'(i).\rho] = m'[\sigma''(j).\rho].$$

  Then by the assumptions concerning m', $\sigma'(i).\rho$ and $\sigma''(j).\rho$ must agree on each of the x_l and on y:

  $$\forall\, l \in 1 \mathinner{..} k \bullet X_l = \sigma'(i).\rho(x_l) = \sigma''(j).\rho(x_l) = \sigma(j).\rho(x_l) \wedge$$
  $$\sigma'(i).\rho(y) = \sigma''(j).\rho(y) = \sigma(j).\rho(y).$$

  Finally, $j \neq i$ by assumption 2 of the rule. Hence the second disjunct in the definition of *provesKnowledgeOf*(xs) is satisfied.

- Case $j : \mathsf{receive}\, M$. Suppose this event occurs from state σ''. We consider two cases.

  Case $M' \in \sigma_0(0)$. This cannot hold if part (a) of assumption 3 holds. If part (b) holds, then all the direct submessages and the encrypting key are also known initially; i.e.

  $$\forall\, l \in \{1 \mathinner{..} k\} \bullet \sigma_0(0) \vdash X_l$$

  because of the assumption about the form of the message. Hence

  $$\forall\, l \in \{1 \mathinner{..} k\} \bullet \sigma(0) \vdash X_l.$$

  Case $M' \notin \sigma_0(0)$. Then $M' \notin \sigma''(0)$, since M' cannot have been added to that knowledge by any subsequent event. Hence by Lemmas 10 and 14, the intruder knows all the direct subcomponents of M' and the encrypting key. So by the assumption about the form of m', the intruder knows the values instantiating each of the x_l:

  $$\forall\, l \in \{1 \mathinner{..} k\} \bullet \sigma''(0) \vdash X_l.$$

Hence

$$\forall\, l \in \{1 \,..\, k\} \bullet \sigma(0) \vdash X_l.$$

In both cases, the first disjunct in the definition of *provesKnowledgeOf*(xs) is satisfied.

Hence we have shown $m \in [\![provesKnowledgeOf(xs)]\!]_\Pi$, and so

$$provesKnowledgeOf(xs) \;\sqsubseteq_\Pi\; m.$$

□

Note that it is important that the elements of xs are *direct* subcomponents (or the encrypting key). For example $\{x, \{y\}_w\}_z$ does not refine *provesKnowledgeOf*(x, y): the intruder could form this message by taking a message of the form $\{y\}_w$ for which he does not know y, and then building the message using his own value for x; then no single agent knows both x and y.

The following rule extends Refinement Rule 3 to deal with the *provesKnowledgeOf*$(xs, id = b)$ abstract message.

Refinement Rule 4 Suppose the conditions of Refinement Rule 3 are satisfied, and in addition only role b ever sends messages containing m', i.e.:

$$\begin{array}{l} \forall\, j \in 1 \,..\, n \bullet \\ \quad (\exists\, m \bullet \mathsf{send}\ m \text{ in } \sigma_0(j).prog \wedge m' \sqsubset m) \Rightarrow \sigma_0(j).id = b. \end{array}$$

Then

$$provesKnowledgeOf(xs, id = b) \;\sqsubseteq_\Pi\; m.$$

Proof: The proof is an extension of the proof of Refinement Rule 3. There is nothing further to prove in the receive case. For the send case, we know that for the j in question, $\sigma_0(j).id = b$ from the additional condition; hence $\sigma(j).id = b$, as required for the second disjunct in the semantics of *provesKnowledgeOf*$(xs, id = b)$, and hence *provesKnowledgeOf*$(xs, id = b) \sqsubseteq_\Pi m$.
□

The following two rules extend Refinement Rule 3 to deal with the *provesKnowledgeOfNR* abstract message.

Refinement Rule 5 Suppose the conditions of Refinement Rule 3 are satisfied, and in addition, for some a, b:

1. only role b ever sends messages containing m';
2. only role a ever receives the message m in question;
3. $a \neq b$ is an invariant of the protocol;

4. either (a) $b \ll m'$, or (b) for some $x \in xs$, $associatedWith_x(b)$ is an invariant for a.

Then

$$provesKnowledgeOfNR(xs, id = b) \sqsubseteq_\Pi m.$$

Condition 4(b) could, under reasonable assumptions, be satisfied by taking x to be a shared secret or b's secret key, for example.

Proof: The proof is an extension of the proof of Refinement Rule 3. There is nothing further to prove in the receive case. For the send case, we proceed as follows. We have $\sigma_0(j).id = b$ from assumption 1, and $\sigma_0(i).id = a$ from assumption 2. Further, $\sigma(i).\rho(a) \neq \sigma(i).\rho(b)$ from assumption 3. If part (a) of assumption 4 holds, then the fact that nodes i and j see the same instantiation of m' gives us that $\sigma'(i).\rho(b) = \sigma''(j).\rho(b)$; alternatively, if part (b) holds, then the fact that $\sigma'(i).\rho(x) = \sigma''(j).\rho(x)$ again gives us $\sigma'(i).\rho(b) = \sigma''(j).\rho(b)$. Putting these together, and noting that the values of a and b in the two states don't change, we have

$$\begin{aligned}&\sigma(j).\rho(\sigma(j).id) = \sigma''(j).\rho(b) = \sigma'(i).\rho(b)\\ &\quad = \sigma(i).\rho(b) \neq \sigma(i).\rho(a) = \sigma(i).\rho(\sigma(i).id),\end{aligned}$$

as required for the second disjunct in the semantics of *provesKnowledgeOfNR*. Hence $provesKnowledgeOfNR(xs, id = b) \sqsubseteq_\Pi m$. □

Refinement Rule 6 Suppose the conditions of Refinement Rule 5 are satisfied, except assumption 4 is replaced by:

4′. either (a) $a \ll m'$, or (b) for some $x \in xs$, $associatedWith_x(a)$ is an invariant for a.

Then

$$provesKnowledgeOfNR(xs, id = b) \sqsubseteq_\Pi m.$$

Condition 4′(b) could, under reasonable assumptions, be satisfied by taking x to be a's public key, for example.

Proof: The proof is very similar to that for Refinement Rule 5, except the new condition gives us $\sigma'(i).\rho(a) = \sigma''(j).\rho(a)$, and assumption 3 gives us that $\sigma(j).\rho(a) \neq \sigma(j).\rho(b)$; putting the pieces together then gives us the desired result. □

The following rule relates *provesKnowledgeOfNR* with and without the $id = b$ clause; it shows that in the previous two rules, the message in question also refines $provesKnowledgeOfNR(xs)$.

Refinement Rule 7

$$provesKnowledgeOfNR(xs) \sqsubseteq provesKnowledgeOfNR(xs, id = b).$$

Proof: Direct from the semantics. $\square$

For each of the above rules, there is a corresponding rule that gives a refinement to a hashed message; we give just the analogue of Refinement Rule 5:

Refinement Rule 8 Suppose message template m is such that for some hashed message template $m' = hash(m'') \sqsubset m$,

$$\forall x \in xs \bullet x \ll m''.$$

Then

$$provesKnowledgeOfNR(xs, id = b) \sqsubseteq_{\Pi} m,$$

provided:

1. the protocol satisfies the disjoint encryption property;
2. no node sends and then receives an instantiation of m';
3. the intruder does not initially know any instantiation of m' unless he also knows the direct submessages;
4. only role b ever sends messages containing m';
5. only role a ever receives the message m in question;
6. $a \neq b$ is an invariant of the protocol;
7. $b \ll m'$.

Note that the above rule justifies the refinement

$$provesKnowledgeOfNR(na) \sqsubseteq hash(na, b)$$

from the introductory example.

6.5 *associate*

The abstract message $associate_x(y)$ represents the requirement that the values of x and y should be associated together within the state of the recipient. More precisely, it means that if the intruder does not learn x, then any other agent who obtains the value of x will also obtain the value of y. The intention is that x will normally be a secret. This message allows the sender to deduce that any other agent who receives x associates it with y.

6.5.1 Semantics

The semantics of $associate_x(y)$ is the set of messages m such that if participant i sends m, then anyone who subsequently has their x variable bound to the same value, will also have their y variable bound to the same value;

an exception is if the intruder can learn the value of x: if so, he can pass it on in a way that breaks the association.

$$
\begin{aligned}
&[\![associate_x(y)]\!]_\Pi = \\
&\quad \{m \mid \forall\, tr \frown \langle \sigma, i : \mathsf{send}\ m\rangle \frown tr' \frown \langle \sigma' \rangle \in traces(\Pi) \bullet \\
&\qquad\qquad \sigma'(0) \vdash \sigma(i).\rho(x) \vee \\
&\qquad\qquad \forall\, j > 0 \bullet \sigma'(j).\rho(x) = \sigma(i).\rho(x) \Rightarrow \sigma'(j).\rho(y) = \sigma(i).\rho(y)\}.
\end{aligned}
$$

Note that the second disjunct is equivalent to $associatedWith_x(y)(\sigma')[i]$.

If ys is a set of variables, then the abstract message $associate_x(ys)$ associates the values of all of the members of ys with x. The following semantic definition is an obvious extension of the one above.

$$
\begin{aligned}
&[\![associate_x(ys)]\!]_\Pi = \\
&\quad \{m \mid \forall\, tr \frown \langle \sigma, i : \mathsf{send}\ m\rangle \frown tr' \frown \langle \sigma' \rangle \in traces(\Pi) \bullet \\
&\qquad\qquad \sigma'(0) \vdash \sigma(i).\rho(x) \vee \\
&\qquad\qquad \forall\, j > 0 \bullet \sigma'(j).\rho(x) = \sigma(i).\rho(x) \Rightarrow \\
&\qquad\qquad\qquad\qquad \forall\, y \in ys \bullet \sigma'(j).\rho(y) = \sigma(i).\rho(y)\}.
\end{aligned}
$$

6.5.2 Proof rule

The following proof rule shows how *associate* can be used in annotations.

Annotation Rule 16

$$
\begin{aligned}
&a : \left\{true\right\} \\
&\quad \mathsf{send}\ associate_x(y) \\
&\quad \left\{\Box \left(honest(knows(x)) \Rightarrow associatedWith_x(y)\right)\right\}
\end{aligned}
$$

Note that the postcondition implies that, assuming $honest(knows(x))$, if there is a session for some b in which b's x variable is bound to a's value for x, then in that session b's y variable is bound to a's value for y.

Proof: Suppose

$$
\begin{aligned}
&\sigma_0(i).id = a \wedge \sigma_0(i).prog = es_0 \frown e' \frown es_1 \wedge e' \sqsupseteq \mathsf{send}\ associate_x(y) \wedge \\
&\forall\, \sigma \in States(\Pi) \mid \sigma(i).prog = e' \frown es_1 \bullet true(\sigma)[i].
\end{aligned}
$$

Let σ' be such that $\sigma'(i).prog = es_1$, and let σ be the state immediately before the event corresponding to e'. Suppose $\sigma' \Longrightarrow \sigma''$. Let $X = \sigma(i).\rho(x)$. From the semantics of *associate*,

$$\sigma''(0) \vdash X \vee associatedWith_x(y)(\sigma'')[i].$$

If $honest(knows(x))(\sigma'')[i]$ then $\sigma''(0) \nvdash X$, so $associatedWith_x(y)(\sigma'')[i]$; hence

$$(honest(knows(x) \Rightarrow associatedWith_x(y))(\sigma'')[i]$$

for all σ'' such that $\sigma' \Longrightarrow \sigma''$. Hence

$$\Box(honest(knows(x) \Rightarrow associatedWith_x(y))(\sigma')[i],$$

as required. $\Box$

The following rule is an easy extension of the rule above.

Annotation Rule 17

$$\begin{array}{l} a : \left\{true\right\} \\ \quad \textsf{send}\ associate_x(ys) \\ \quad \left\{\Box\left(honest(knows(x)) \Rightarrow associatedWith_x(ys)\right)\right\}. \end{array}$$

6.5.3 Refinement rules

We now prove a refinement rule relating to *associate*. We show, subject to some provisos, including that x is a fresh secret, that if an encrypted component contains x and all the elements of ys, either as direct sub-messages or as the decrypting key, then that message refines $associate_x(ys)$. For example, subject to the provisos:

$$\begin{array}{lll} associate_x(y,z) \ \sqsubseteq \ \{x,y,z\}_k, & \\ associate_x(y,z) \ \sqsubseteq \ \{x,y\}_{z^{-1}}, & \\ associate_x(y,z) \ \sqsubseteq \ \{y,z\}_x, & \quad \text{(provided } x \text{ is symmetric).} \end{array}$$

Further, any message containing such an encrypted component, possibly with additional fields, will refine the same abstract message.

Refinement Rule 9 Suppose message template m is such that for some encrypted message template $m' = \{m''\}_k \sqsubset m$,

$$\forall\, y \in ys \cup \{x\} \bullet y \ll m'' \vee y = k^{-1}. \tag{2}$$

Then

$$associate_x(ys) \ \sqsubseteq_\Pi \ m,$$

provided:

1. the protocol satisfies the disjoint encryption property;
2. if $x = k$, then it is a *symmetric* key;
3. x is freshly generated in the node that sends this message;
4. all other nodes that receive the variable x do so within the component m'.

Proof: Following the definition of $associate_x(ys)$, suppose

$$tr \frown \langle \sigma, i : \mathsf{send}\ m[\sigma(i).\rho] \rangle \frown tr' \frown \langle \sigma' \rangle \in traces(\Pi).$$

Let $X = \sigma(i).\rho(x)$. We must show that

$$\sigma'(0) \vdash X \vee$$
$$\forall j > 0 \bullet \sigma'(j).\rho(x) = X \Rightarrow \forall y \in ys \bullet \sigma'(j).\rho(y) = \sigma(i).\rho(y).$$

Suppose this is not the case. It is clearly true initially (no node holds X initially), so consider the first state σ' at which it becomes false. Then

$$\sigma'(0) \nvdash X.$$

By assumption 3, and the fact that send events do not change bindings, the event leading to σ' must be a receive event, say $j : \mathsf{receive}\ M$, from state σ'' to σ'. By assumption 4, j must receive X (bound to x) in this message within an instantiation of m', namely $M' = m'[\sigma'(j).\rho]$, with $\sigma'(j).\rho(x) = X$, but $\sigma'(j).\rho(y) \neq \sigma(i).\rho(y)$ for some $y \in ys$.

Consider how the intruder produced this component M'. By the assumption that this is the first point at which the condition fails, and the disjoint encryption assumption, no honest agent can have sent this component earlier; further, it is not in the intruder's initial knowledge, since X is fresh; hence $M' \notin \sigma''(0)$. Hence, if $M' = \{M''\}_K$, then by Lemma 10, $\sigma''(0) \vdash M'' \wedge \sigma''(0) \vdash K$. Consider equation (2) for x: if $x \ll m''$ then $X \ll M''$, so $\sigma''(0) \vdash X$ by Lemma 14; if $x = k^{-1}$, then $X = K^{-1} = K$, by assumption 2, and so again $\sigma''(0) \vdash X$. Hence, in both cases $\sigma'(0) \vdash X$, giving a contradiction; and so we deduce the desired result. □

6.6 *canExtract* and *keepSecret*

6.6.1 Semantics

The abstract message $canExtract_b(x)$ represents those concrete templates m, such that when they are sent, no agent other than b can learn the value of x; this should remain true until this agent performs another event; however, if the intruder knew the value of x when the message was sent, or the intruder is b, then he can pass off x to other agents so that they can learn it, so the above condition no longer holds. More precisely, suppose an instantiation of m is sent from state σ by node i (i.e. $i : \mathsf{send}\ m[\sigma(i).\rho]$), and i performs no subsequent events before state σ', and the intruder didn't know i's value X of x in state σ (i.e. $X = \sigma(i).\rho(x)$), and the intruder isn't i's value B of b; then those nodes that know X in σ' are at most those who knew either X

before, and B.

$$
\begin{array}{l}
[\![\mathit{canExtract}_b(x)]\!]_\Pi \mathrel{\widehat{=}} \\
\quad \{ m \mid \forall\, tr \frown \langle \sigma, i : \mathsf{send}\ m[\sigma(i).\rho] \rangle \frown tr' \frown \langle \sigma' \rangle \in \mathit{traces}(\Pi) \mid \\
\qquad\qquad tr' \upharpoonright i = \langle\rangle \wedge B \neq \mathit{intruder} \wedge \sigma(0) \not\vdash X \bullet \\
\qquad\qquad \mathit{knows}(X)(\sigma') \subseteq \mathit{knows}(X)(\sigma) \cup \{B\} \\
\qquad\qquad \text{where } X = \sigma(i).\rho(x),\ B = \sigma(i).\rho(b) \}.
\end{array}
$$

The abstract message $\mathit{canExtract}_b(\mathit{xs})$, where xs is a set of variables, places the same condition for all elements of xs:

$$
[\![\mathit{canExtract}_b(\mathit{xs})]\!]_\Pi \mathrel{\widehat{=}} \bigcap_{x \in \mathit{xs}} [\![\mathit{canExtract}_b(x)]\!]_\Pi .
$$

Note that

$$
\mathit{canExtract}_b(x_1, \ldots, x_m) = \mathit{canExtract}_b(x_1) \wedge \ldots \wedge \mathit{canExtract}_b(x_m).
$$

The abstract message $\mathit{canExtract}_{bs}(x)$ specifies that only the agents in the set bs can learn the value of x:

$$
\begin{array}{l}
[\![\mathit{canExtract}_{bs}(x)]\!]_\Pi \mathrel{\widehat{=}} \\
\quad \{ m \mid \forall\, tr \frown \langle \sigma, i : \mathsf{send}\ m[\sigma(i).\rho] \rangle \frown tr' \frown \langle \sigma' \rangle \in \mathit{traces}(\Pi) \mid \\
\qquad\qquad tr' \upharpoonright i = \langle\rangle \wedge \mathit{intruder} \notin \mathit{Bs} \wedge \sigma(0) \not\vdash X \bullet \\
\qquad\qquad \mathit{knows}(X)(\sigma') \subseteq \mathit{knows}(X)(\sigma) \cup \mathit{Bs} \\
\qquad\qquad \text{where } X = \sigma(i).\rho(x), \mathit{Bs} = \sigma(i).\rho(bs) \}.
\end{array}
$$

The following is an obvious extension:

$$
[\![\mathit{canExtract}_{bs}(\mathit{xs})]\!]_\Pi \mathrel{\widehat{=}} \bigcap_{x \in \mathit{xs}} [\![\mathit{canExtract}_{bs}(x)]\!]_\Pi .
$$

The abstract message $\mathit{keepSecret}(x)$ specifies that x must not be revealed to anyone. It can be defined as syntactic sugar, as follows:

$$
\mathit{keepSecret}(x) \mathrel{\widehat{=}} \mathit{canExtract}_{\{\}}(x).
$$

This gives the following semantic equation:

$$
\begin{array}{l}
[\![\mathit{keepSecret}(x)]\!]_\Pi = \\
\quad \{ m \mid \forall\, tr \frown \langle \sigma, i : \mathsf{send}\ m[\sigma(i).\rho] \rangle \frown tr' \frown \langle \sigma' \rangle \in \mathit{traces}(\Pi) \mid \\
\qquad\qquad tr' \upharpoonright i = \langle\rangle \wedge \sigma(0) \not\vdash X \bullet \\
\qquad\qquad \mathit{knows}_{id}(X)(\sigma') = \mathit{knows}_{id}(X)(\sigma) \\
\qquad\qquad \text{where } X = \sigma(i).\rho(x) \}.
\end{array}
$$

The *keepSecret* message is extended to sets of variables in the obvious way:

$$
\mathit{keepSecret}(\mathit{xs}) \mathrel{\widehat{=}} \mathit{canExtract}_{\{\}}(\mathit{xs}),
$$

so

$$
[\![\mathit{keepSecret}(\mathit{xs})]\!]_\Pi = \bigcap_{x \in \mathit{xs}} [\![\mathit{keepSecret}(x)]\!]_\Pi .
$$

6.6.2 Annotation rules

The following proof rule shows how *canExtract* can be used in annotations.

Annotation Rule 18 (canExtract) For all sets as and bs of variables representing agent identities,

$$
\begin{array}{l}
a : \left\{ knows(x) = as \wedge honest(as \cup bs) \right\} \\
\quad \mathsf{send}\ canExtract_{bs}(x) \\
\quad \left\{ as \subseteq knows(x) \subseteq as \cup bs \right\}
\end{array}
$$

Proof: Suppose

$$
\begin{array}{l}
\sigma_0(i).id = a \wedge \sigma_0(i).prog = es_0 \frown e' \frown es_1 \wedge e' \sqsupseteq \mathsf{send}\ canExtract_b(x) \wedge \\
\forall\, \sigma \in States(\Pi) \mid \sigma(i).prog = e' \frown es_1 \bullet \\
\quad (knows(x) = as \wedge honest(as \cup bs))(\sigma)[i].
\end{array}
$$

Let σ' be such that $\sigma'(i).prog = es_1$, and let σ be the event immediately before the event corresponding to e'. Let

$$
X = \sigma(i).\rho(x), \qquad As = \sigma(i).\rho(as), \qquad Bs = \sigma(i).\rho(bs).
$$

Then substituting in the assumption corresponding to the precondition, we have

$$
knows(X)(\sigma) = As \wedge honest(As \cup Bs).
$$

So from the definition of $canExtract_b(x)$

$$
knows(X)(\sigma') \subseteq knows(X)(\sigma) \cup Bs = As \cup Bs.
$$

But $\sigma'(i).\rho(x) = \sigma(i).\rho(x)$, and similarly for as and bs, and so

$$
(knows(x) \subseteq as \cup bs)\,(\sigma')[i]
$$

as required. $\square$

The following rule shows how *keepSecret* can be used in annotations.

Annotation Rule 19 (keepSecret)

$$
\begin{array}{l}
\left\{ knows(x) = as \wedge honest(as) \right\} \\
\mathsf{send}\ keepSecret(x) \\
\left\{ knows(x) = as \right\}
\end{array}
$$

Proof: Similar to the proof of the previous rule. $\square$

6.6.3 Refinement rules

The *canExtract* messages are monotonic in their main argument, and anti-monotonic in their key argument:

Refinement Rule 10 If $xs \subseteq xs' \land bs \supseteq bs'$ then

$$canExtract_{bs}(xs) \sqsubseteq canExtract_{bs'}(xs').$$

Proof: Direct from the semantics. $\square$

The following is an immediate corollary:

Refinement Rule 11

$$canExtract_{bs}(xs) \sqsubseteq keepSecret(xs).$$

The following rule allows the agents' identities argument of *canExtract* to be expanded with the addition of identities of honest agents who may already know the relevant message.

Refinement Rule 12

$$a : canExtract_{as}(x) \sqsubseteq \\ \Big\{knows(x) \supseteq bs \land honest(bs)\Big\}\, canExtract_{as \cup bs}(x).$$

Proof: Let $a \mathrel{\hat=} \sigma_0(i).id$, $X \mathrel{\hat=} \sigma(i).\rho(x)$, $As \mathrel{\hat=} \sigma_0(i).\rho(as)$ and $Bs \mathrel{\hat=} \sigma_0(i).\rho(bs)$. Following the definition of conditional refinement, let

$$ms = \{m \mid \forall\, tr \frown \langle\sigma,\, i : \mathsf{send}\; m\rangle \in traces(\Pi) \bullet \\ (knows(x) \supseteq bs \land honest(bs))(\sigma)[i]\}.$$

Suppose $m \in [\![canExtract_{as \cup bs}(x)]\!]_\Pi \cap ms$; we need to show $m \in [\![canExtract_{as}(x)]\!]_\Pi$.

Following the semantics of $canExtract_{as}(x)$, suppose

$$tr \frown \langle\sigma,\, i : \mathsf{send}\; m[\sigma(i).\rho]\rangle \frown tr' \frown \langle\sigma'\rangle \in traces(\Pi),$$

and $tr \upharpoonright i = \langle\rangle \land intruder \notin As \land \sigma(0) \nvdash X$. But $intruder \notin Bs$, because $m \in ms$ and so $honest(bs)(\sigma)[i]$. Then

$$knows(X)(\sigma') \subseteq knows(X)(\sigma) \cup As \cup Bs,$$

since $m \in [\![canExtract_{as \cup bs}]\!]_\Pi$. But $knows(X)(\sigma) \supseteq Bs$ since $m \in ms$ and so $(knows(x) \supseteq bs)(\sigma)[i]$. Hence

$$knows(X)(\sigma') \subseteq knows(X)(\sigma) \cup As,$$

and so $m \in [\![canExtract_{as}(x)]\!]_\Pi$. Hence we have shown that $[\![canExtract_{as}(x)]\!]_\Pi \supseteq [\![canExtract_{as \cup bs}(x)]\!]_\Pi \cap ms$, and so the result holds. $\square$

We now prove a lemma which shows, essentially, that if the variable x is never sent in any message, then it is not learnt by any agent. We then use the lemma to prove an annotation rule for $\mathit{keepSecret}(x)$.

Lemma 15 Consider some node i. Suppose

1. The protocol satisfies the disjoint encryption property.
2. No agent sends a message m such that $x \sqsubset m$.
3. Initially i's value for x is bound only to x in other nodes:

$$\mathit{uniquelyBound}(x)(\sigma_0)[i].$$

4. Initially x is held only by honest agents as:

$$(\mathit{holds}(x) = as \wedge \mathit{honest}(as))(\sigma_0)[i].$$

Then $\mathit{holds}(x) = as$ is an invariant for i.

Proof: Let $X \mathrel{\widehat{=}} \sigma_0(i).\rho(x)$ and $As \mathrel{\widehat{=}} \sigma_0(i).\rho(as)$.

We begin by showing that the intruder does not learn X; more precisely, we show $\mathit{intruder} \notin \mathit{holds}(X)$ in all states. This is true initially by assumption 4. Suppose, for a contradiction, that the intruder does come to hold X; then this will necessarily be from a send event, so suppose

$$tr \frown \langle \sigma, j : \mathsf{send}\, M, \sigma' \rangle \in \mathit{traces}(\Pi),$$

with $\mathit{intruder} \in \mathit{holds}(X)(\sigma')$, $\mathit{intruder} \notin \mathit{holds}(X)(\sigma)$. Then necessarily $X \sqsubset M$. But in σ, the conditions of Theorem 1 hold, so $\mathit{uniquelyBound}(x \rightsquigarrow X)(\sigma)$. Hence X must instantiate x in M, which contradicts assumption 2.

Hence the intruder never learns X, so we can use Theorem 1, again, to deduce that $\mathit{uniquelyBound}(x \rightsquigarrow X)$ holds in all states.

Finally, no agent other than those in As learns X: for all $\mathsf{receive}\, m$ templates, $x \not\sqsubset m$, so $x \notin \mathit{vars}(m)$; but bindings are updated for such events only for variables in $\mathit{vars}(m)$, so x can never become bound unless it is bound initially. $\square$

The following refinement rule adapts the above lemma to a form more convenient for proving refinements.

Refinement Rule 13 If

1. The protocol satisfies the disjoint encryption property.
2. No agent sends a message m such that $x \sqsubset m$.
3. Initially a's value for x is bound only to x:

$$\mathit{uniquelyBound}(x).$$

4. Initially x is held only by honest agents as:

$$\mathit{holds}(x) = as \wedge \mathit{honest}(as).$$

Then every message refines $\mathit{keepSecret}(x)$.

We can use the above rule to verify the refinements of $\mathit{keepSecret}(k)$ in the example of Section 2. However, we need the following additional assumptions corresponding to conditions of the refinement rule:

$\mathit{uniquelyBound}(k) \wedge \mathit{holds}(k) = \{a, b\}.$

Both of these conditions turn out to be necessary; suppose the local node is i, and let $A \mathrel{\widehat{=}} \sigma(i).\rho(a)$, $B \mathrel{\widehat{=}} \sigma(i).\rho(b)$ and $K \mathrel{\widehat{=}} \sigma(i).\rho(k)$:

- Suppose either A or B has some other role, c say, in which K is bound to some other variable, k' say, and suppose in the role c, k' is sent as plaintext; then the value K of k would not remain secret; the $\mathit{uniquely}$-$\mathit{Bound}(k)$ condition prevents this.

- Suppose the intruder initially knows K encrypted with some value that he subsequently learns; then clearly he will also subsequently learn K; this is not prevented by the assumption $\mathit{knows}(k) = \{a, b\}$, but is prevented by the additional assumption $\mathit{holds}(k) = \{a, b\}$.

We now prove a refinement rule for *canExtract*. We begin with a lemma that shows, essentially, that if every occurrence of x is either (a) encrypted with a key that only some subset of as knows, or (b) hashed, then no agent other than those in as learns x

Lemma 16 Consider some node i with role a, and some set of roles as. Suppose

1. The protocol satisfies the disjoint encryption property.
2. For every role b, for every occurrence of x in a message sent by b, one of the following holds:
   (a) x is encrypted by some k such that for some set of roles $as' \subseteq as$, $\mathit{knows}(k^{-1}) \subseteq as'$ (as an invariant for b); and x is associated with as' for b (as an invariant for i):

   $$\begin{array}{l} \exists\, as' \subseteq as \bullet \\ \quad \mathit{knows}(k^{-1}) \subseteq as' \text{ is invariant for } b \wedge \\ \quad \mathit{associatedWith}_x(as')(b) \text{ is invariant for } i. \end{array}$$

   (Normally, we will take $as' = \{a\}$ if the encryption is with a's public key, or $as' = \{a, b\}$ if the encryption is with a symmetric key shared by a and b. The $\mathit{associatedWith}_x(as')$ condition ensures that b has the same values for as' as i does.)

   (b) x is within a hash.
3. Either (a) node i holds x initially, and initially only the honest agents as hold x, and do so well-bound:

   $$(\mathit{holds}(x) \subseteq as \wedge \mathit{honest}(as) \wedge \mathit{uniquelyBound}(x))(\sigma_0)[i],$$

   or (b) node i generates x freshly.

Then $knows(x) \subseteq as$ is an invariant for i.

Note that Lemma 15 is a special case of this; assumption 2 holds vacuously under the assumptions of that lemma.

Proof: Let $X \mathrel{\widehat{=}} \sigma(i).\rho(x)$ (either the value held initially, or the value generated by i, depending upon which case of assumption 3 holds), $A = \sigma_0(i).\rho(a)$, and $As \mathrel{\widehat{=}} \sigma_0(i).\rho(as)$.

We begin by showing that the intruder does not learn X. More precisely, for every σ, we show the following:

> Every occurrence of X within $\sigma(0)$ is either (a) encrypted by some key K such that K^{-1} is invariably unknown by the intruder (i.e., for every σ' such that $\sigma \Longrightarrow \sigma'$, $\sigma'(0) \nvdash K^{-1}$); or (b) hashed.

This is true initially by assumption 3. The only way it can become false subsequently is via a send event, so suppose

$$tr \frown \langle \sigma, j : \mathsf{send}\, M, \sigma' \rangle \in traces(\Pi),$$

and the above statement is true in σ but false in σ'. Then it must be that $X \sqsubset M$, not encrypted or hashed as above. But in σ, the conditions of Theorem 1 hold, so $uniquelyBound(x \rightsquigarrow X)(\sigma)$, so, in particular, $\sigma(j).\rho(x) = X$. Hence, by assumption 2, every occurrence of X in M is either: (a) encrypted by some key K such that $(knows(K^{-1}) \subseteq as')(\sigma)[j]$ holds as an invariant; or (b) hashed. In case (a), by the assumption $associatedWith_x(as')(b)(\sigma)[i]$, we have that $\sigma(j).\rho(as') = \sigma(i).\rho(as') \subseteq \sigma(i).\rho(as) = As$; hence the intruder cannot learn K^{-1}, giving a contradiction. In case (b), the result is immediate.

Hence the intruder never learns X, so we can use Theorem 1, again, to deduce that $uniquelyBound(x \rightsquigarrow X)$ holds in all states. Further, because every occurrence of X is encrypted by some K such that $knows(K^{-1}) \subseteq As$, or hashed, only members of As can learn X, by the assumption that the protocol is feasible. Hence $knows(x) \subseteq as$ is an invariant for i. $\square$

Refinement Rule 14 If the conditions of Lemma 16 are satisfied, then every message refines $canExtract_{as}(x)$.

We can apply the above rule to the example from Section 2 to show that every message satisfies $canExtract_{a,b}(na)$. Note that the message $\{na\}_k$ satisfies condition 2(a): because a himself sends this message, the binding condition is automatically satisfied. The message $hash(na, b)$ satisfies condition 2(b).

7 The Needham Schroeder Public Key Protocol

In this section we give a derivation of the Adapted Needham Schroeder Public Key Protocol [Low95]. We give derivations for both roles of the protocol.

The protocol works by combining two one-way authentication tests, based on decryption of a public-key encrypted nonce. Identity information is included to ensure that the nonces are associated with the correct identities, to avoid man-in-the-middle attacks: we capture this association using the *associate* abstract message.

The protocol makes use of public keys. Below, we will write pka and pkb for a's and b's public keys, and ska and skb for the corresponding secret keys, so $ska = pka^{-1}$ and $skb = pkb^{-1}$.

7.1 Perspective of participant a

We start by considering the perspective of agent a. The protocol will make use of an invariant that says that a and b are distinct honest agents, that only the appropriate agents know the secret keys, that only a and b learn na, and that na is associated with a:

$$\begin{aligned} I \ \widehat{=}\ & a \neq b \wedge honest(a, b) \wedge \\ & knows(ska) = \{a\} \wedge knows(skb) = \{b\} \wedge \\ & defined(na) \Rightarrow knows(na) \subseteq \{a, b\} \wedge \\ & defined(na) \Rightarrow associatedWith_{na}(a). \end{aligned}$$

Note that a's state will include b's public key pkb, but not his secret key skb; recall that in this case the notation $knows(skb) = \{b\}$, in an annotation for a, means that only b knows the secret key corresponding to the public key held in pkb.

We will use the following message invariant:

$$MI \ \widehat{=}\ keepSecret(ska, skb) \wedge canExtract_{a,b}(na).$$

The first two conjuncts of the invariant are clearly invariant statements. The two conjuncts concerning the secrecy of the secret keys will be maintained because of the first conjunct of the message invariant; this suffices for send messages by Annotation Rule 19; the receive messages maintain these conjuncts because of Annotation Rule 5. The conjunct of the invariant concerning the secrecy of na is maintained because of the second conjunct of the message invariant; this suffices because of Annotation Rule 18 for send messages and Annotation Rule 5 for receive messages. We will deal with the association of na with a below.

An annotation of the protocol for agent a is shown in Figure 1. We will need to add some extra initial assumptions later. The protocol proceeds as follows:

- Firstly, a generates a new nonce na.
- Next, a sends a message from which only b can extract na, and which associates na with a.

- Then, a receives a message that proves knowledge of na, nb and b, from somebody in role b; because only a and b know na, we can deduce that it must be b who has the corresponding session; because na is associated with a, we can deduce that b's session must be with a.
- Finally a sends a message that keeps na secret (this message is mainly relevant to b from a security point of view).

Each assertion is justified in the annotation by reference to the relevant rule.

$$\{I\}$$

new na

$$\{I \wedge knows(na) = \{a\} \langle \text{Annotation Rule 7} \rangle\}$$

send $MI \wedge associate_{na}(a)$

$$\left\{\begin{array}{l} I \wedge \Box(honest(knows(na)) \Rightarrow associatedWith_{na}(a)) \\ \langle \text{Annotation Rule 16} \rangle \end{array}\right\}$$

receive $MI \wedge provesKnowledgeOf(na, nb, b, id = b)$

$$\left\{\begin{array}{l} I \wedge (honest(knows(na)) \Rightarrow associatedWith_{na}(a)) \wedge \\ \exists B \bullet session(b \rightsquigarrow B;\ na, nb, b) \langle \text{Annotation Rule 13} \rangle \end{array}\right\}$$

$$\left\{\begin{array}{l} I \wedge session(b;\ na, nb, a) \\ \langle knows(na) \subseteq \{a, b\} \text{ so } B \neq intruder\text{, above; Lemma 7} \rangle \end{array}\right\}$$

send MI

$$\{I \wedge session(b;\ na, nb, a)\}$$

Figure 1: The Adapted Needham Schroeder Public Key Protocol: a's perspective

7.2 Perspective of participant b

For b's perspective, the invariant and message invariant are very similar to as for a:

$$\begin{array}{rcl} I & \widehat{=} & a \neq b \wedge honest(a, b) \wedge \\ & & knows(ska) = \{a\} \wedge knows(skb) = \{b\} \wedge \\ & & defined(nb) \Rightarrow knows(nb) \subseteq \{a, b\} \wedge \\ & & defined(nb) \Rightarrow associatedWith_{nb}(b), \\ MI & \widehat{=} & keepSecret(ska, skb) \wedge canExtract_{a,b}(nb). \end{array}$$

The message invariant ensures that the second and third lines on the invariant are maintained, as in the case for a.

The protocol from b's perspective is shown in Figure 2. The protocol proceeds as follows:

- b receives a message that satisfies the message invariant; this step is necessary to fit in with a's initial send, and to allow b to learn na;
- b generates a fresh nonce nb;
- b sends a message from which only a can extract nb, and which associates b and na with nb;
- Finally b receives a message that proves knowledge of nb in role a; because only a and b know na, we can deduce that it must be a who has the corresponding session; because of the association, we can deduce that that session involves b and na.

$$\Big\{I\Big\}$$
$\mathsf{receive}\ MI$
$$\Big\{I \Big\langle \text{Annotation Rule 5}\Big\rangle\Big\}$$
$\mathsf{new}\ nb$
$$\Big\{I \wedge \mathit{knows}(nb) = \{b\} \Big\langle \text{Annotation Rule 7}\Big\rangle\Big\}$$
$\mathsf{send}\ MI \wedge \mathit{associate}_{nb}(b, na)$
$$\left\{\begin{array}{l} I \wedge \Box(\mathit{honest}(\mathit{knows}(nb)) \Rightarrow \mathit{associatedWith}_{nb}(b, na)) \\ \Big\langle \text{Annotation Rule 17}\Big\rangle \end{array}\right\}$$
$\mathsf{receive}\ MI \wedge \mathit{provesKnowledgeOfNR}(nb, id = a)$
$$\left\{\begin{array}{l} I \wedge (\mathit{honest}(\mathit{knows}(nb)) \Rightarrow \mathit{associatedWith}_{nb}(b, na)) \wedge \\ \exists\, A \bullet \mathit{session}(a \rightsquigarrow A;\ nb \rightsquigarrow nb) \wedge A \neq b \Big\langle \text{Annotation Rule 15}\Big\rangle \end{array}\right\}$$
$$\left\{\begin{array}{l} I \wedge \mathit{session}(a;\ nb, na, b) \\ \Big\langle \mathit{knows}(nb) \subseteq \{a, b\} \text{ so } A = a \text{ above; Lemma 7}\Big\rangle \end{array}\right\}$$

Figure 2: The Adapted Needham Schroeder Public Key Protocol: b's perspective

We can strengthen the postconditions in the two annotations. Note that the annotation for a shows that $\mathit{session}(a;\ nb, na, b) \Rightarrow \mathit{knows}(na) \subseteq \{a, b\}$; hence we may add the conjunct $\mathit{knows}(na) \subseteq \{a, b\}$ to the postcondition in the annotation for b. Similarly, the annotation for b shows that $\mathit{session}(b;\ na, nb, a) \Rightarrow \mathit{knows}(nb) \subseteq \{a, b\}$; hence we may add the conjunct $\mathit{knows}(nb) \subseteq \{a, b\}$ to the postcondition in the annotation for a.

7.3 Concrete messages

Putting the two annotations together, we may refine the protocol to obtain the normal definition:

$$
\begin{array}{lll}
\text{Message 1.} & a \to b & : \{a, na\}_{pkb} \\
\text{Message 2.} & b \to a & : \{b, na, nb\}_{pka} \\
\text{Message 3.} & a \to b & : \{nb\}_{pkb}.
\end{array}
$$

We have a number of proof obligations in order to justify this.

Firstly, we can use Refinement Rule 13 to show that each message refines $\mathit{keepSecret}(\mathit{ska}, \mathit{skb})$. This introduces the following additional initial assumptions:

$$
\begin{array}{l}
\mathit{uniquelyBound}(\mathit{ska}) \wedge \mathit{holds}(\mathit{ska}) = \{a\} \wedge \\
\mathit{uniquelyBound}(\mathit{skb}) \wedge \mathit{holds}(\mathit{skb}) = \{b\}.
\end{array}
$$

Next, we can use Refinement Rule 14 to show that each message refines $\mathit{canExtract}_{a,b}(na)$, from a's perspective. In message 1, na is encrypted by pkb, whose inverse skb is known only to b; clearly na is associated with b in a's state. In message 2, na is encrypted by pka, whose inverse ska is known only to a; na is associated with a in b's state because of the $\mathit{associatedWith}_{na}(a)$ clause of a's invariant. We can show that each message refines $\mathit{canExtract}_{a,b}(nb)$ in a very similar way.

Next, we can show

$$\mathit{provesKnowledgeOf}(na, nb, b, id = b) \sqsubseteq \{b, na, nb\}_{pka}$$

using Refinement Rule 4, noting that na is freshly generated by the recipient a. We can similarly show

$$\mathit{provesKnowledgeOfNR}(nb, id = a) \sqsubseteq \{nb\}_{pkb}$$

using Refinement Rule 6, noting that nb is freshly generated by the recipient b, and $\mathit{associated}_{nb}(b)$.

Finally we can show

$$
\begin{array}{rl}
\mathit{associate}_{na}(a) & \sqsubseteq \{a, na\}_{pkb}, \\
\mathit{associate}_{nb}(b, na) & \sqsubseteq \{b, na, nb\}_{pka}.
\end{array}
$$

using Refinement Rule 9.

The alert reader may be worried that some of the above reasoning is circular: the refinement for $\mathit{associate}_{na}(a)$ requires that na not be known by the intruder; but the refinement for $\mathit{canExtract}_{a,b}(na)$ requires that na is associated with a (and likewise for nb). However, the proof of the rule for *associate* requires that na be secret in one state in order for the association to hold in the *next* state; and similarly, the proof of the rule for *canExtract* requires the association to hold in one state in order for na to remain secret in the *next* state. So if both hold, in one state, then they will both hold in the following state, and so on inductively.

It is worth considering how the development would proceed if we were developing the standard Needham Schroeder Public Key Protocol [NS78], which does not contain a b inside the encryption of message 2. In this case, in b's annotation, the *associate* statement would be replaced by $associate_{nb}(na)$; the $B = b$ clauses are then removed from the subsequent assertions, and the final *session* assertion becomes $session(a;\ nb, na)$: in other words, b can be sure that a is running a session using the nonces nb and na, but cannot be sure that a associates that session with him. Further, we wouldn't be able to prove that the messages refine $canExtract(nb)$, because nb is not associated with b, and so b would receive no guarantee that nb remains secret. Both of these correspond to the well-known attack.

8 Conclusions

We have created a calculus for protocol development, based upon the idea of annotating protocols: we add assertions to the protocol description, stating properties that will be true when that point in the protocol is reached. A novel feature of our calculus is the idea of abstract messages, which state what a message is intended to achieve, rather than giving a concrete implementation.

We have presented proof rules that can be used to justify assertions, and refinement rules that allow abstract messages to be implemented. We have produced a semantic model, and used it to formalise the meaning of annotations, and to verify the rules. We have illustrated the calculus by using it to develop the Adapted Needham Schroeder Public Key Protocol.

An essential ingredient in proving many of our message refinement rules was Theorem 1, which said, roughly, that under the disjoint encryption assumption, secret values remain uniquely bound. This seems to be a powerful result, which we have not seen stated previously, and which might be of use in other formalisms.

Recall that our model of a global state admits the possibility of multiple protocols operating in the same environment. When we use message refinement rules that place restrictions upon the protocol, those restrictions apply to all protocols in the environment. Most of those restrictions are about the way that particular variables are used; if we use different variable names in different protocols, then such restrictions will automatically be satisfied by all protocols other than the primary one. The remaining condition is that of disjoint encryption; in order for this to be satisfied, we should arrange for the other protocols to have no messages of the same textual form as those in the primary protocol: this is very similar to the result of [GTF00].

8.1 Future Work

We intend to undertake more case studies in protocol development. A goal would be to produce developments of a significant number of protocols, perhaps most of those from Clark and Jacob's library [CJ97]. These case studies will help us to identify additional useful abstract messages, together with their associated proof rules. For the existing abstract messages, we believe that we have most necessary annotation rules; however, we expect to find many more refinement rules, because there are many ways of achieving a particular requirement.

These case studies will also help us to develop techniques and experience, showing the best way to approach a protocol development. Based on the examples we have carried out so far, we would offer the following suggestions:

- Proving the secrecy of a fresh value seems to be easier when one argues from the point of view of the agent that generates that value; secrecy of other values can be proven at the end, as we did with the Adapted Needham Schroeder Public Key Protocol.
- A development does not seem to be a linear process: it is often necessary to add initial assumptions, or to add conditions to the invariant, in order to refine the abstract messages to concrete messages.

In [GT00a, GT00b], Guttman and Thayer introduce the idea of *authentication tests*, capturing various patterns whereby an agent may be authenticated. An *outgoing authentication test* is where an agent a sends out a fresh value x such that only b can extract it, and then receives back a message that proves knowledge of x; this is captured as an annotation as follows:

$$\begin{array}{l} \mathsf{new}\ x \\ \mathsf{send}\ \mathit{canExtract}_b(x) \\ \mathsf{receive}\ \mathit{provesKnowledgeOf}(x) \\ \Big\{\mathit{session}(b;\ x)\Big\} \end{array}$$

This is roughly the example from Section 2, and is also the pattern used by both roles for the Needham Schroeder Protocol. An *incoming authentication test* is where a sends out a fresh value x, and receives it back in a form that only b could have created. An *unsolicited authentication test* is where a receives a message that only b could have created. We would like to capture these latter two patterns as annotations.

We would like to provide tool support, both for the initial annotation of the protocol, and for the refinement of abstract messages to concrete messages. A prototype tool has been developed for the latter stage (although this is not consistent with the current refinement rules).

8.2 Related Work

Datta et al. [DDMP03] investigate the derivation of protocols from smaller, well-used ideas (such as Diffie-Hellman key exchange, and authentication using a signed nonce challenge), using development techniques such as composition of protocols, refinements (changing the form of particular messages) and transformations (changing the structure of the protocol). They informally develop a family of protocols using these techniques. They then formally verify the development of one of them, using a logic, founded on the cord calculus [DMP01]. Like us, their logic annotates protocols with assertions; however, whereas our logic concentrates on the states of agents, particularly the values stored in variables, their logic concentrates upon the events performed, and in particular their relative order.

Saïdi [Sai02] investigates the synthesis of protocols from a specification based on BAN Logic; he derives the Needham-Schroeder Public Key protocol by applying simple inference rules.

Canetti and Krawczyk [CK02] also develop a composable notion of key exchange leading to secure channels; this allows for individual components such as key exchange to be separated from a single protocol, and so be reused by many protocols.

Acknowledgements

We would like to thank Michael Goldsmith, Bill Roscoe and Sadie Creese for helpful comments on this work.

References

[BAN89] Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication. *Proceedings of the Royal Society of London A*, 426:233–271, 1989. A preliminary version appeared as Digital Equipment Corporation Systems Research Center report No. 39, 1989.

[CJ97] John Clark and Jeremy Jacob. A survey of authentication protocol literature: Version 1.0. Available via `http://www.cs.york.ac.uk/~jac/papers/drareview.ps.gz`, 1997.

[CK02] Ran Canetti and Hugo Krawczyk. Universally composable notions of key exchange and secure channels. In *Theory and Application of Cryptographic Techniques*, pages 337–351, 2002.

[Coh00] Ernie Cohen. Taps: A first-order verifier for cryptographic protocols. In *Proceedings of 13th IEEE Computer Security Foundations Workshop*, pages 144–158, 2000.

[DDMP03] A. Datta, A. Derek, J. Mitchell, and D. Pavlovic. A derivation system for security protocols and its logical formalization. In *Proceedings of The 16th Computer Security Foundations Workshop*. IEEE Computer Society Press, 2003.

[DMP01] Nancy Durgin, John Mitchell, and Dusko Pavlovic. A compositional logic for proving security properties of protocols. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, pages 241–255, 2001.

[DY83] D. Dolev and A.C. Yao. On the security of public-key protocols. *Communications of the ACM*, 29(8):198–208, August 1983.

[GNY90] Li Gong, Roger Needham, and Raphael Yahalom. Reasoning about belief in cryptographic protocols. In Deborah Cooper and Teresa Lunt, editors, *Proceedings 1990 IEEE Symposium on Research in Security and Privacy*, pages 234–248. IEEE Computer Society, 1990.

[GT00a] Joshua D. Guttman and F. Javier Thayer Fábrega. Authentication tests. In *Proceedings of 2000 IEEE Symposium on Security and Privacy*, 2000.

[GT00b] Joshua D. Guttman and F. Javier Thayer Fábrega. Authentication tests and the structure of bundles. *Theoretical Computer Science*, 283(2):333–380, 2000.

[GTF00] Joshua Guttman and Javier Thayer Fábrega. Protocol independence through disjoint encryption. In *Proceedings of The 13th Computer Security Foundations Workshop*. IEEE Computer Society Press, 2000.

[HLS03] James Heather, Gavin Lowe, and Steve Schneider. How to prevent type flaw attacks on security protocols. *Journal of Computer Security*, 11(2):217–244, 2003.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Low95] Gavin Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56:131–133, 1995.

[Low96] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proceedings of TACAS*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer Verlag, 1996. Also in *Software—Concepts and Tools*, 17:93–102, 1996.

[Low98] Gavin Lowe. Casper: A compiler for the analysis of security protocols. *Journal of Computer Security*, 6:53–84, 1998.

[MCJ97] Will Marrero, Edmund Clarke, and Somesh Jha. A model checker for authentication protocols. In *Proceedings of the DIMACS Workshop on Design and Formal Verification of Security Protocols*, 1997. Available via URL `http://dimacs.rutgers.edu/Workshops/Security/program2/program.html`.

[Mea96] Catherine Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

[MMS97] John C. Mitchell, Mark Mitchell, and Ulrich Stern. Automated analysis of cryptographic protocols using Murϕ. In *IEEE Symposium on Security and Privacy*, 1997.

[NS78] Roger Needham and Michael Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[OR87] D. Otway and O. Rees. Efficient and timely mutual authentication. *Operating Systems Review*, 21(1):8–10, January 1987.

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

[Sai02] Hassen Saidi. Towards automatic synthesis of security protocols. In *In Logic-Based Program Synthesis Workshop, AAAI 2002 Spring Symposium*, 2002.

[SBP01] Dawn Xiaodong Song, Sergey Berezin, and Adrian Perrig. Athena: a novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*, 9(1, 2):47–74, 2001.

[THG99] F. Javier Thayer Fábrega, Jonathan C. Herzog, and Joshua D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7(2, 3):191–230, 1999.

A Index of notation

Notation	Description	Section
$\Box$	Annotation macro; $\Box\, P$ means that P holds in this and all subsequent states.	4.3

$\vdash$	Message derivation relation; $B \vdash M$ means that the intruder can produce M from the set of messages B.	3.5
$\upharpoonright$	Operator projecting a sequence of events onto those performed by a particular node.	3.6
$\sqsubseteq$	Message refinement relation.	3.2
$\sqsubset$	Submessage relation; $M \sqsubset M'$ if M is textually included within M'.	3.1
$\preceq$	Submessage relation, including encrypting and decrypting keys as submessages.	3.1
$\ll$	Direct submessage relation; $M \ll M'$ if M is a submessage of M' that can be obtained without performing any decryption.	3.1
$\longrightarrow$	Local state transition relation; $s \xrightarrow{E} s'$ means that from local state s, event E can be performed to reach state s'.	3.3
$\longrightarrow$	Global state transition relation; $\sigma \xrightarrow{i:E} \sigma'$ means that from global state σ, event E can be performed by node i to reach state σ'.	3.6
$\Longrightarrow$	Global trace transition relation; $\sigma \stackrel{tr}{\Longrightarrow} \sigma'$ means that from global state σ, the trace tr can be performed to reach state σ'.	3.6
$P(\sigma)[i]$	Predicate P, as interpreted by node i in state σ.	4.1
ρ	Binding component of a local state.	3.3
σ_0	Initial global state.	3.6
AbsMsg	Type of abstract messages.	3.2
associate	Abstract message; $associate_x(y)$ means that the value of x is associated inseparably with the value of y.	6.5
associated-With	Annotation macro; $associatedWith_x(y)$ means that the value of x is associated inseparably with the value of y.	4.3
Binding	Type of bindings, i.e. mappings from variables to values.	3.3
canExtract	Abstract message; $canExtract_b(x)$ means that only b may learn the value of x from the message.	6.6
defined	Annotation macro; $defined(x)$ means that the variable x has a value associated with it.	4.3

Event	Type of events.	3.3
Event-Template	Event templates.	3.3
GlobalState	Global states.	3.6
holds	Annotation macro; $holds(X)$ gives the identities of agents who hold X as a submessage of a message they know.	4.3
honest	Annotation macro; $honest(as)$ means that the agents as are honest, i.e. follow the protocol.	4.3
id	Identity or role variable of a local state.	3.3
intruder	Identity of the intruder.	3.5
isNew	Function testing whether a value is new in a particular state.	3.6
keepSecret	Abstract message; $keepSecret(x)$ means that no agent may learn x from this message.	6.6
knows	Annotation macro; $knows(x)$ gives the set of identities of agents who know the value of x.	4.3
$knows_{id}$	$knows_{id}(x)$ gives the set of indices of nodes that know the value of x.	4.3
LocalState	Type of states of local agent or nodes.	3.4
Msg	Type of messages.	3.1
new	Event or event template, representing a new value being generated.	3.3
newpair	Event or event template, representing a new asymmetric key pair being generated.	3.3
Prog	Program, i.e. sequence of event templates, performed by a node.	3.3
prog	Program component of a local state.	3.3
proves-KnowledgeOf	Abstract message; $provesKnowledgeOf(x)$ shows that some agent knows x.	6.4
proves-Knowledge-OfNR	Abstract message; $provesKnowledgeOfNR(x)$ shows that some agent other than the local agent knows x.	6.4
receive	Event or event template, representing a message being received.	3.3
send	Event or event template, representing a message being sent.	3.3

session	Annotation macro; $session(b;\ x)$ means that b is taking part in a session, and agrees with the local agent on the value of x.	4.3
States	Function giving the reachable states of a protocol.	3.6
Template	Type of message templates.	3.1
traces	Function giving the traces of a protocol.	3.6
Type	Types of messages.	3.1
$type_*$	Functions giving the type of variables, atomic values, templates and messages.	3.1
TypeName	Names of atomic types.	3.1
uniquely-Bound	Annotation macro; $uniquelyBound(x)$ means that the current node's value for x is bound only to x in other nodes.	4.3
Val	Type of atomic values.	3.1
Var	Type of variables.	3.1
vars	Function giving the variables of a message template, event template or program.	3.3