A Calculus for Security Protocol Development

Gavin Lowe* Michael Auty†

August 21, 2006

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. Some properties require global, as opposed to local, reasoning; such properties are captured as invariants; rules are given for verifying that invariants hold.

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, message refinement rules, and invariant rules. The calculus is illustrated with the development of three well known protocols.

*Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD, UK; gavin.lowe@comlab.ox.ac.uk.

†QinetiQ, St Andrew's Road, Malvern, Worcestershire, WR14 3PS, UK.

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 have few 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. 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. 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. We provide separate rules for showing that such invariants are maintained.

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.

To help the reader understand the various elements involved in this calculus, we give a simple worked example in Section 2. 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 theorem, 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 give rules that can be used to verify that certain common forms of invariant are, indeed, maintained.

In Section 8 we look at three larger examples, namely the Adapted Needham Schroeder Public Key Protocol [Low95], the Otway Rees Protocol [OR87a], and the Yahalom Protocol [BAN89]; we derive each protocol using the rules presented earlier. Finally, in Section 9, 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.

The main clause of the invariant states that only a and b should learn na:

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

The macro $knows(na)$ represents the set of participants who know na; $defined(na)$ means that na exists, i.e. a value has been generated for this variable; and $honest(b)$ means that b is honest, i.e. follows the protocol. Here we specify that once na has been generated, if b is honest, then only a and b may learn it; it doesn't make sense to talk about who knows na before it is generated; and if b is dishonest, then there is nothing preventing him from passing na on to third parties.

We will also assume that a and b are different agents. We therefore define the following invariant:

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

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

$$honest(b) \Rightarrow session(b;\; na).$$

The predicate $session(b;\ na)$ 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:

$$
\begin{array}{l}
\mathit{Initiator}(a;\ b) \mathrel{\widehat{=}} \\
\left\{a \neq b\right\} \\
\left\{I\right\} \\
\ldots \\
\left\{I \wedge (\mathit{honest}(b) \Rightarrow \mathit{session}(b;\ na))\right\}
\end{array}
$$

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 $a \neq b$; this represents the precondition of the protocol. At the end of the protocol the invariant and $(\mathit{honest}(b) \Rightarrow \mathit{session}(b;\ na))$ must hold; this represents the postcondition of the protocol. The ellipses ("...") represent the part of the protocol that we still need to develop. The first line specifies that the protocol is being run by a, and that the variable b is initially defined in a's state. The string "*Initiator*" simply names this role.

The assertion "I" follows from the fact that $a \neq b \Rightarrow I$, since initially na is not defined. Formally, we have used the following rule, *strengthen precondition* (the "a :" indicates that the annotation relates to role a):

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

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

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

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

We now arrange for a to generate the nonce, as follows:

$$
\begin{array}{l}
\mathit{Initiator}(a;\ b) \mathrel{\widehat{=}} \\
\left\{a \neq b\right\}\ \left\{I\right\} \\
\mathsf{new}\ na\ \left\{I\right\} \\
\ldots \\
\left\{I \wedge (\mathit{honest}(b) \Rightarrow \mathit{session}(b;\ na))\right\}
\end{array}
$$

The $\mathsf{new}\ na$ event creates a new nonce and binds it to na in the local state. Afterwards the invariant will still hold: indeed, only a will know na. Later we will give a rule (Annotation Rule 41) that justifies this step.

We will often concatenate several events; for example, in the above annotation, the $\mathsf{new}\ na$ is concatenated with the part of the protocol still to be developed. The following proof rule, *sequential composition*, formalises 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 $a : \left\{pre\right\}\ e_1\ \left\{mid\right\}\ e_2\ \left\{post\right\}$.

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

$$\begin{array}{l}
Initiator(a;\ b) \mathrel{\widehat{=}} \\
\left\{a \neq b\right\}\ \left\{I\right\} \\
\mathsf{new}\ na\ \left\{I\right\} \\
\mathsf{send}\ maintains\ I \wedge contains\ na\ \left\{I\right\} \\
\ldots \\
\left\{I \wedge (honest(b) \Rightarrow session(b;\ na))\right\}
\end{array}$$

The message is formed as the conjunction of two abstract messages:

- $maintains\ I$ is an abstract message that specifies that I must be maintained, i.e., the invariant should hold after the message, assuming it held before; this justifies the following assertion of I.

- $contains\ na$ is an abstract message that specifies that the message should contain na, the intention being that b learns na from this message; this does not lead to any extra assertions, but helps to document the intention of the design.

We next arrange for a to receive a message that shows her 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:

$$
\begin{array}{l}
\mathit{Initiator}(a;\ b) \mathrel{\widehat{=}} \\
\big\{a \neq b\big\}\ \big\{I\big\} \\
\textsf{new}\ na\ \ \big\{I\big\} \\
\textsf{send}\ \mathit{maintains}\ I \wedge \mathit{contains}\ na\ \ \big\{I\big\} \\
\textsf{receive}\ \mathit{maintains}\ I \wedge \mathit{provesKnowledgeOfNR}(na, id = b) \\
\big\{I \wedge \exists\, B \neq a \bullet \mathit{session}(b \leadsto B;\ na)\big\} \\
\big\{I \wedge (\mathit{honest}(b) \Rightarrow \mathit{session}(b;\ na))\big\}
\end{array}
$$

This message centres around $\mathit{provesKnowledgeOfNR}(na, id = b)$, which informs the receiver a that somebody, say B, knows the value of na; that participant B is taking the role of b in the protocol (the "$id = b$" clause); further, B is not a herself: this extra clause ensures that messages are not reflected (the "NR" stands for "not reflected"). This gives us the $\exists\, B \neq a \bullet \mathit{session}(b \leadsto B;\ na)$ assertion. The message also maintains the invariant.

If b is honest, we can deduce that the only people who know na are a and b. Therefore, we can deduce that B, who has na in his state, must be b. This establishes the required postcondition.

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.

We now make a design step, by deciding how to keep na secret. We will assume that the agents share a secret key k, which will be used to encrypt na in the first message. We strengthen the invariant to specify that if b is honest then k remains secret:

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

The annotation above remains unchanged, except the first line is changed to $\mathit{Initiator}(a;\ b, k)$, reflecting that a has k in her initial state, and the precondition is changed to include the clause $\mathit{honest}(b) \Rightarrow \mathit{knows}(k) \subseteq \{a, b\}$.

Of course, strengthening I changes the meaning of the *maintains* I abstract message.

We can refine the first abstract message to $\{na\}_k$. It is intuitively obvious that this does not reveal k. Further, since only a and b know k, it does not reveal na to anyone else. Therefore, it maintains the invariant. We will give rules to formally justify this later.

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 this maintains the invariant, and refines *provesKnowledgeOfNR*$(na, id = b)$.

This gives us the concrete protocol below:

$$\mathsf{new}\ na;\quad \mathsf{send}\{na\}_k;\quad \mathsf{receive}\ hash(na, b).$$

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 Sections 6 and 7.

Of course, the above is not the only possible refinement of the abstract messages. For example, we could have implemented the first message by encrypting na with b's public key ($\{na\}_{PK(b)}$), 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[1] 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 $\perp \notin \mathit{Val}$, representing an undefined value.

We assume two inverse functions:

$$\_^{-1_{var}} : \mathit{Var} \nrightarrow \mathit{Var} \quad \text{and} \quad \_^{-1_{val}} : \mathit{Val} \nrightarrow \mathit{Val}$$

between variables and values. If K is a value representing a key then messages encrypted with K can be decrypted with $K^{-1_{val}}$, and vice versa. If k is a variable representing a key then it is intended that $k^{-1_{var}}$ holds the corresponding decrypting key. We assume that $(k^{-1_{var}})^{-1_{var}} = k$, 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:

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

Messages and templates are built up from atomic values by pairing, encryption and hashing[2]. 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 $M \trianglelefteq M'$ if M is textually included within M':

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

[1] We use the term "template" in a different sense from [DDMP04].

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

We define a similar relation over message templates:

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

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

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

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

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:

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

$$
\begin{aligned}
M \ll M' &\Leftarrow M = M', \\
M \ll (M_1, M_2) &\Leftarrow M \ll M_1 \vee M \ll M_2.
\end{aligned}
$$

We will make the *strong typing assumption*: i.e. that each honest agent will accept a value received only 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, $\{(Nonce, AgentIdentity)\}_{PublicKey}$ represents the type of nonces and agent identities encrypted with public keys.

We assume a function

$$type_{var} \; : \; Var \rightarrow Type$$

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

$$type_{val} \; : \; Val \rightarrow Type.$$

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

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

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

$$\begin{aligned} AbsMsg \; ::= \; & Template \mid AbsMsg \wedge AbsMsg \mid maintains \; P \mid \\ & contains \; x \mid provesKnowledgeOfNR(Var, id = Var) \mid \ldots. \end{aligned}$$

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 *proves-KnowledgeOfNR*(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:

$$[\![\_]\!]_{\_} \;:\; \mathit{AbsMsg} \times \mathit{Protocol} \rightarrow \mathbb{P}\,\mathit{Template}.$$

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:

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

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: abstract messages are refined to templates, and templates are instatiated to messages.

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 (finite) 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 $\mathsf{newpair}(x, y)$ represents the fresh generation of an 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 $x \in \mathrm{dom}\, \_^{-1} \wedge x^{-1} \neq x$). We will not use the $\mathsf{newpair}$ construct within any examples, and so do not give any rules concerning it; we include it here, to allow for such extensions in future.

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:

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

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

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

We lift the notion of refinement from event templates to programs pointwise:[3]

$$\begin{aligned} \mathit{prog} \sqsubseteq_\Pi \mathit{prog}' \Leftrightarrow\; &\mathit{length}\,\mathit{prog} = \mathit{length}\,\mathit{prog}' \land \\ &\forall\, i \in 1 \mathinner{..} \mathit{length}\,\mathit{prog} \bullet \mathit{prog}(i) \sqsubseteq_\Pi \mathit{prog}'(i). \end{aligned}$$

We write $\mathit{vars}(m)$ for the set of variables appearing in message template m. 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*:

$$\mathit{Binding} \;\widehat{=}\; \mathit{Var} \nrightarrow \mathit{Msg}.$$

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 [OR87b], or the Yahalom Protocol [BAN89]).

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 $\mathit{type}_{var}(x) = \mathit{type}_{msg}(X)$.

[3] $\mathit{prog}(i)$ 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 $\rho(x)$. If $x \notin \operatorname{dom} \rho$, 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 $(prog, \rho, id) : Prog \times Binding \times Var$, where $prog$ is the remaining sequence of event templates it needs to perform, ρ is a binding, and $id \in \operatorname{dom} \rho$ is a distinguished variable that represents the local agent's identity. Given a local state s, we will write "$s.prog$", "$s.\rho$" and "$s.id$" 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 $s.id$ is the *variable* that represents the agent's identity, not the value of that identity, which is stored in $s.\rho(s.id)$. We assume that an agent will use the same value of $s.\rho(s.id)$ in all of his nodes, i.e. he uses the same identity in all his runs.

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, and defined by the following grammar:

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

Note that events deal with actual concrete messages and values.

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:

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

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

- The semantics of $\mathsf{newpair}$ is very similar, except two values, which must be inverses, are involved:

$$
\begin{aligned}
&(\langle \mathsf{newpair}(x,y)\rangle \frown prog, \rho, id) \xrightarrow{\mathsf{newpair}(X,Y)} \\
&\qquad (prog, \rho \oplus \{x \mapsto X, y \mapsto Y\}, id), \\
&\text{provided } type_{val}(X) = type_{var}(x),\ type_{val}(Y) = type_{var}(y), \\
&\qquad\qquad\quad X^{-1} = Y.
\end{aligned}
$$

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

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

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

$$
\begin{aligned}
&(\langle \mathsf{receive}\, m\rangle \frown prog, \rho, id) \xrightarrow{\mathsf{receive}\, m[\rho']} (prog, \rho', id), \\
&\text{provided } \rho' \supseteq \rho,\ \mathrm{dom}\,\rho' = \mathrm{dom}\,\rho \cup vars(m), \\
&\qquad\qquad\quad \forall\, x \in \mathrm{dom}\,\rho' \bullet type_{var}(x) = type_{msg}(\rho'(x)), \\
&\qquad\qquad\quad \forall\, x, y \in \mathrm{dom}\,\rho' \bullet x^{-1} = y \Rightarrow \rho'(x)^{-1} = \rho'(y).
\end{aligned}
$$

  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 $\exists\, E \in Event \bullet s \xrightarrow{E} s'$.

The following lemma captures some properties of the operational semantics.

Lemma 1. If $(prog \frown prog', \rho, id) \longrightarrow^* (prog', \rho', id')$ then $\rho \subseteq \rho' \wedge \mathrm{dom}\,\rho' = \mathrm{dom}\,\rho \cup vars(prog) \wedge id' = id$.

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:

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

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

Lemma 2. $\mathit{wellTyped}(\rho) \wedge (\mathit{prog}, \rho, \mathit{id}) \xrightarrow{E} (\mathit{prog}', \rho', \mathit{id}) \Rightarrow \mathit{wellTyped}(\rho')$.

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 $\mathit{canUnpack}(\mathit{xs}, \mathit{ms})$ 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:

$$\begin{aligned} & \mathit{canUnpack}(\mathit{xs}, \{\}), \\ \mathit{canUnpack}(\mathit{xs}, \{v\} \cup \mathit{ms}) \ \Leftarrow\ & \mathit{canUnpack}(\mathit{xs} \cup \{v\}, \mathit{ms}), \\ & \quad \text{for } v \in \mathit{Var}, \\ \mathit{canUnpack}(\mathit{xs}, \{(m_1, m_2)\} \cup \mathit{ms}) \ \Leftarrow\ & \mathit{canUnpack}(\mathit{xs}, \{m_1, m_2\} \cup \mathit{ms}), \\ \mathit{canUnpack}(\mathit{xs}, \{\{m\}_k\} \cup \mathit{ms}) \ \Leftarrow\ & k^{-1} \in \mathit{xs} \wedge \\ & \mathit{canUnpack}(\mathit{xs}, \{m\} \cup \mathit{ms}), \\ \mathit{canUnpack}(\mathit{xs}, \mathit{hash}(m) \cup \mathit{ms}) \ \Leftarrow\ & \mathit{vars}(m) \subseteq \mathit{xs} \wedge \mathit{canUnpack}(\mathit{xs}, \mathit{ms}). \end{aligned}$$

Definition 3. 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 ($\mathrm{dom}\,\rho$) and the variables bound subsequently ($vars(prog')$ below):

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

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:

$$\forall\, prog' \frown \langle \mathsf{receive}\; m\rangle \leq prog \bullet \\ \quad canUnpack(\mathrm{dom}\,\rho \cup vars(prog'), \{m\}).$$

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

$$\forall\, prog' \frown \langle \mathsf{new}\; x\rangle \leq prog \bullet x \notin \mathrm{dom}\,\rho \cup vars(prog') \wedge \\ \forall\, prog' \frown \langle \mathsf{newpair}(x, y)\rangle \leq prog \bullet x, y \notin \mathrm{dom}\,\rho \cup vars(prog').$$

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.

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

Lemma 4. If $(prog, \rho, id) \in LocalState$ and $(prog, \rho, id) \longrightarrow^* (prog', \rho', id)$, then $(prog', \rho', id) \in LocalState$.

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 $B \vdash M$ if 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 $B \vdash M \Rightarrow B \vdash \mathit{hash}(M)$.

Below we write "*intruder*" for the identity of the intruder[4].

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 $0 \mathrel{..} n$ 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:

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

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 run the protocol multiple times, possibly with different roles.

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 $\_^{-1_{val}}$.

[4] 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.

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

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

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:

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

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:

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

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:

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

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

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

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 $\mathit{traces}(\Pi)$ for the set of all traces that can be observed of Π. We define $\mathit{States}(\Pi)$ to be all the reachable states, i.e. states appearing in some trace of Π.

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

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

4 Annotations

In this section we consider annotations in more detail. In Section 4.1 we formally define the meaning of an annotation and of an invariant. 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 and invariants

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

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

Similarly, if pka is a's public key then the assertion $\mathit{knows}(\mathit{pka}^{-1}) = \{a\}$ specifies that only a knows the corresponding secret key; this is interpreted as follows:

$$(\mathit{knows}(\mathit{pka}^{-1}) = \{a\})(\sigma)[i] \;\;\equiv\;\; \mathit{knows}(\mathit{PKA}^{-1})(\sigma) = \{A\}$$
$$\text{where } \mathit{PKA} = \sigma(i).\rho(\mathit{pka}),\; A = \sigma(i).\rho(a).$$

(The "$\_^{-1}$" is the inverse operation over Val, i.e. $\_^{-1_{val}}$.) Recall that if $x \notin \mathrm{dom}\,\sigma(i).\rho$, 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 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 5. Predicate P is an *invariant* of protocol Π for node i if it holds in all states:

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

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

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

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 $a : \left\{\mathit{pre}\right\} \mathit{es} \left\{\mathit{post}\right\}$, 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 6.

$$\begin{array}{l} a : \left\{\mathit{pre}\right\} \mathit{es} \left\{\mathit{post}\right\} \mathrel{\widehat{=}} \\ \quad \forall\, i \in 1 \mathinner{..} n \mid \sigma_0(i).\mathit{id} = a \bullet \\ \quad \forall\, \mathit{es}_0, \mathit{es}_1, \mathit{es}' \mid \sigma_0(i).\mathit{prog} = \mathit{es}_0 \frown \mathit{es}' \frown \mathit{es}_1 \wedge \mathit{es}' \sqsupseteq \mathit{es} \bullet \\ \qquad (\forall\, \sigma \in \mathit{States}(\Pi) \mid \sigma(i).\mathit{prog} = \mathit{es}' \frown \mathit{es}_1 \bullet \mathit{pre}(\sigma)[i]) \\ \qquad \Rightarrow \\ \qquad (\forall\, \sigma' \in \mathit{States}(\Pi) \mid \sigma'(i).\mathit{prog} = \mathit{es}_1 \bullet \mathit{post}(\sigma')[i]). \end{array}$$

The following lemma relates annotations to invariants.

Lemma 7. If

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

then P is an invariant of the protocol for a.

Note that we will often have to *assume* that the invariant holds in the initial state.

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 8 (Strengthen precondition).

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

Proof: Suppose

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

Then by the second hypothesis,

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

So 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. □

The proofs of the following rules are very similar.

Annotation Rule 9 (Weaken postcondition).

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

Annotation Rule 10 (Sequential composition).

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

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

Annotation Rule 11 (Conjunction of postconditions).

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

4.3 Annotation macros

In this section we give semantics to several annotation macros.

4.3.1 *knows*

The macro $\mathit{knows}(x)$ 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:

$$\mathit{knows}(x)(\sigma)[i] \;=\; \mathit{knows}(x)[\sigma(i).\rho](\sigma) \;=\; \mathit{knows}(\sigma(i).\rho(x))(\sigma).$$

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

The value X is known by the agent of honest node i if $\sigma(i).\rho(y) = X$ for some y; X is known by the intruder if $\sigma(0) \vdash X$.

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

Note that the value of $\mathit{knows}(X)$ 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 $\mathit{knows}(X)$ cannot decrease as an execution progresses.

4.3.2 *holds*

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

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

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

The following lemma shows that A can acquire X by freshly generating it, by receiving a message including X, or, if A is the intruder, by another agent sending it.

Lemma 12. Suppose A aquires X from event $j : E$:

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

Let $B = \sigma(j).\rho(\sigma(j).id)$. Then

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

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 j, the variable representing the agent's identity is b, that b is bound to B, and each x_l is bound to X_l. If B is dishonest then the notation means that B knows each of the X_l: a dishonest agent is not forced to bind values to variables in any predictable way.

$$\begin{array}{l}session(b \rightsquigarrow B;\; x_1 \rightsquigarrow X_1, \ldots, x_k \rightsquigarrow X_k)(\sigma) \mathrel{\widehat=} \\ \quad \exists\, j > 0 \bullet \sigma(j).id = b \wedge \sigma(j).\rho(b) = B \wedge \forall\, l \in 1\,..\,k \bullet \sigma(j).\rho(x_l) = X_l \\ \quad \vee \\ \quad B = intruder \wedge \forall\, l \in 1\,..\,k \bullet \sigma(0) \vdash X_l.\end{array}$$

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,

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

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 value. 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 *session* macro does not talk about *recentness* of sessions: if an agent a has a postcondition of the form $\mathit{session}(b;\ \ldots)$, then that does not necessarily guarantee that b's session was recent. Further it does not guarantee a 1-1 relationship between the runs of a and those of b, the so-called *injectivity* property [Low97]; this property is important, for example, in financial protocols. However, if a has a postcondition of the form $\mathit{session}(b;\ x, \ldots)$ where x is freshly generated by a, then clearly b's session is indeed recent, and there is a 1-1 relationship between a's and b's sessions.

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

Lemma 13. $\big(\mathit{session}(a \rightsquigarrow b;\ x \rightsquigarrow y) \Rightarrow b \in \mathit{knows}(y)\big)(\sigma)[i]$.

The following lemma relates the *session* macro to invariants. If an annotation for a includes a term of the form $\mathit{session}(b;\ \ldots)$, then a's annotation can, roughly speaking, be strengthened with b's invariant.

Lemma 14. Suppose I is invariant for role b, and let the free variables of I be a subset of $\{b, x_1, \ldots, x_k, y_1, \ldots, y_m\}$. Then the following assertion is invariant for all nodes:

$$\mathit{session}(b;\ x_1, \ldots, x_k) \wedge \mathit{honest}(b) \ \Rightarrow\ \exists\, y_1, \ldots, y_m \bullet I$$

Proof: Pick a state σ and a node identifier i. We need to show

$$\big(\mathit{session}(b;\ x_1, \ldots, x_k) \wedge \mathit{honest}(b) \Rightarrow \exists\, y_1, \ldots, y_m \bullet I\big)(\sigma)[i].$$

Let $B = \sigma(i).\rho(b)$, and $X_l = \sigma(i).\rho(x_l)$ for $l = 1, \ldots k$. Suppose $(\mathit{session}(b;\ x_1, \ldots, x_k) \wedge \mathit{honest}(b))(\sigma)[i]$, i.e.,

$$\mathit{session}(b \rightsquigarrow B;\ x_1 \rightsquigarrow X_1, \ldots, x_k \rightsquigarrow X_k)(\sigma) \wedge \mathit{honest}(B)(\sigma).$$

Then for some $j > 0$, $\sigma(j).\rho(b) = B$ and $\sigma(j).\rho(x_l) = X_l$ for each l. Now, from the invariance of I we have $I(\sigma)[j]$ which implies $(\exists\, y_1, \ldots, y_m \bullet I)(\sigma)[j]$. But $\sigma(i).\rho$ and $\sigma(j).\rho$ agree on all the free variables of $\exists\, y_1, \ldots, y_m \bullet I$, and so $(\exists\, y_1, \ldots, y_m \bullet I)(\sigma)[i]$ also holds, as required. $\Box$

4.3.4 *honest*

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

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

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

4.3.5 *defined*

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

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

Note that

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

4.3.6 *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 $associatedWith_{x \rightsquigarrow X}(y_1 \rightsquigarrow Y_1, \ldots, y_n \rightsquigarrow Y_n)(b)$ to indicate that if agent b has X stored in variable x, then he has $Y_1, \ldots, Y_n$ stored in variables $y_1, \ldots, y_n$:

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

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

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

Within annotations, we will use the shorthand

$$\begin{array}{l} associatedWith_{x}(y_1, \ldots, y_n) \;\widehat{=} \\ \quad associatedWith_{x \rightsquigarrow x}(y_1 \rightsquigarrow y_1, \ldots, y_n \rightsquigarrow y_n). \end{array}$$

Recall the convention that substitution does not apply on the left of the $\rightsquigarrow$ symbol; hence $associatedWith_x(y_1, \ldots, y_n)(\sigma)[i]$ 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 inseparably associated with its values for $y_1, \ldots, y_n$. If i does not have x in its state, then $associatedWith_x(ys)(\sigma)[i]$ holds vacuously: the left hand side of the implication becomes $\sigma(j).\rho(x) = \bot$, which is false.

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

Lemma 15.

$$\begin{pmatrix} session(b \rightsquigarrow B;\ x \rightsquigarrow X, y_1 \rightsquigarrow Y_1, \ldots, y_m \rightsquigarrow Y_m) \wedge \\ honest(B) \wedge associatedWith_{x \rightsquigarrow X}(z_1 \rightsquigarrow Z_1, \ldots, z_n \rightsquigarrow Z_n) \end{pmatrix} \Rightarrow \\ session(b \rightsquigarrow B;\ x \rightsquigarrow X, y_1 \rightsquigarrow Y_1, \ldots, y_m \rightsquigarrow Y_m, \\ z_1 \rightsquigarrow Z_1, \ldots, z_n \rightsquigarrow Z_n).$$

4.3.7 *uniquelyBound*

The annotation macro $uniquelyBound(x \rightsquigarrow X)$ means that the (proper) value X is bound only to the variable x, and is not a proper submessage of any variable:

$$\begin{aligned} & uniquelyBound(x \rightsquigarrow X)(\sigma) \mathrel{\widehat{=}} \\ & \quad X \neq \bot \wedge \\ & \quad \forall j > 0;\ y \in Var \bullet (\sigma(j).\rho(y) = X \Rightarrow y = x) \wedge X \not\lhd \sigma(j).\rho(y), \end{aligned}$$

where $\lhd$ is the strict version of the submessage relation $\unlhd$. Note that $uniquelyBound(x \rightsquigarrow \bot)(\sigma)$ is *false*.

We define the standard shorthand:

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

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

We also define the shorthand

$$uniquelyBound(x^{-1}) \mathrel{\widehat{=}} uniquelyBound(x^{-1_{var}} \rightsquigarrow x^{-1_{val}}),$$

so that $(uniquelyBound(x^{-1}))(\sigma)[i]$ means $uniquelyBound(y \rightsquigarrow X^{-1})(\sigma)$ where $y = x^{-1}$ and $X = \sigma(i).\rho(x)$, i.e. the inverse of the value X held in node i's variable x is only stored in other nodes' variable $y = x^{-1}$.

Note that $uniquelyBound(x)(\sigma)[i]$ implies $x \in \operatorname{dom} \sigma(i).\rho$, and $uniquelyBound(x^{-1})(\sigma)[i]$ implies $x \in \operatorname{dom} \sigma(i).\rho$.

4.4 new x

In this section we give a proof rule for new x.

Annotation Rule 16 (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: Following the definition of annotations, suppose

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

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:

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

We consider the two conjuncts of the postcondition separately. For the first conjunct, we need to show $(knows(x) = \{a\})(\sigma')[i]$, i.e., $knows(X)(\sigma') = \{A\}$ 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 12 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 : new X event. Further,

$\hat{\sigma}(i).prog = \langle \mathsf{new}\, x\rangle \frown es_1$. 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: under certain circumstances, a particular value X will be bound to only a single variable x within different agents' states.

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

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

We now capture the disjoint encryption assumption.

Definition 17 (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:

$$
\begin{aligned}
&\{m_1\}_{k_1} \trianglelefteq \sigma_0(i_1).prog(j_1) \wedge \{m_2\}_{k_2} \trianglelefteq \sigma_0(i_2).prog(j_2) \wedge \\
&type(\{m_1\}_{k_1}) = type(\{m_2\}_{k_2}).
\end{aligned}
$$

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 condition 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. Theirs is an inter-protocol property; ours is an intra-protocol property.

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 18. If $B \vdash M' \wedge \{M\}_K \trianglelefteq M'$ then $B \vdash M \wedge B \vdash K$ or $\{M\}_K \trianglelefteq B$.

We now prove the result alluded to above.

Theorem 19. Suppose:

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

   $X \ntrianglelefteq \sigma_0(0)$.

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

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

4. Trace tr ends in state where the intruder does not know X:

   $(\mathrm{last}\, tr)(0) \nvdash X$.

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

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

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

$$\mathit{uniquelyBound}(x \leadsto X)(\mathrm{last}\ \mathit{tr}). \tag{1}$$

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 : $\mathsf{receive}\, M$. Let σ_1 be the final state, last tr. The intruder does not know X in σ_1, 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 $\sigma_1(0) \nvdash M_1$ because $\sigma_1(0) \nvdash X$. Hence by Lemma 18, $\{M_1\}_K \preceq \sigma_1(0)$.

  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 assumption 3; and cannot have stored $\{M_1\}_K$ within a variable as the result of a $\mathsf{receive}$ 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 $type(\{m_1\}_k) = type(\{m'_1\}_{k'})$, 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.

$\square$

6 Abstract messages

In this section we consider abstract messages in more detail.

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

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

We begin by considering refinement in more detail, and prove an annotation rule using refinement. In Section 6.2 we consider the conjunction of abstract messages; in Sections 6.3, 6.4 and 6.5 we consider, respectively, the abstract messages *contains* x, *maintains* P, and *provesKnowledgeOf* and its variants. For each such type of abstract message, we give a formal semantics, annotation rules governing how it can be used in annotations, and refinement rules showing how it can be refined to a concrete message.

6.1 Refinement

Recall 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 20. Refinement is a preorder.

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

Annotation Rule 21 (Refine message).

$$\frac{a : \{pre\}\,\mathsf{send}\,m\{post\} \quad m \sqsubseteq_\Pi m'}{a : \{pre\}\,\mathsf{send}\,m'\{post\}} \qquad \frac{b : \{pre\}\,\mathsf{receive}\,m\{post\} \quad m \sqsubseteq_\Pi m'}{b : \{pre\}\,\mathsf{receive}\,m'\{post\}}$$

Proof: We prove just the rule for sent messages. 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 $a : \{pre\}\,\mathsf{send}\,m'\{post\}$. □

6.2 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:

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

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

The following two rules relate conjunction to refinement.

Refinement Rule 23 (Refinement by conjunction).
$m \;\sqsubseteq\; m \wedge m'$.

Refinement Rule 24 (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 corollary.

Annotation Rule 25 (Conjunction of messages).

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

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

6.3 *contains*

The abstract message $contains\, x$ represents those messages that contain the variable x as a submessage:

$$[\![contains\, x]\!]_\Pi \mathrel{\widehat{=}} \{m \mid x \trianglelefteq m\}.$$

This abstract message is used for documenting the intention of a design, rather than for part of the security analysis. Therefore we do not give any annotation rules for it.

The following refinement rule is very obvious:

Refinement Rule 26 (*contains*). $contains\, x \sqsubseteq_\Pi m$ provided $x \trianglelefteq m$.

6.4 *maintains*

The abstract message $maintains\, P$ represents the set of messages that maintain the property P: if such a message is sent or received in a state σ that satisfies P, then all subsequent states σ', before this node performs another event, must also satisfy P.

$$
\begin{array}{l}
[\![\mathit{maintains}\ P]\!]_\Pi \mathrel{\widehat{=}} \\
\quad \{ m \mid \forall\, tr \frown \langle \sigma, i : \mathsf{send}\ m[\sigma(i).\rho] \rangle \frown tr' \frown \langle \sigma' \rangle \in \mathit{traces}(\Pi) \bullet \\
\qquad\qquad tr' \restriction i = \langle\rangle \wedge P(\sigma)[i] \Rightarrow P(\sigma')[i] \\
\qquad \wedge \\
\qquad \forall\, tr \frown \langle \sigma, i : \mathsf{receive}\ m[\sigma'(i).\rho], \sigma' \rangle \in \mathit{traces}(\Pi) \bullet \\
\qquad\qquad P(\sigma)[i] \Rightarrow P(\sigma')[i] \\
\qquad \wedge \\
\qquad \forall\, tr \frown \langle \sigma, i : \mathsf{receive}\ m[\sigma''(i).\rho], \sigma'' \rangle \frown tr' \frown \langle \sigma' \rangle \in \mathit{traces}(\Pi) \bullet \\
\qquad\qquad tr' \restriction i = \langle\rangle \wedge P(\sigma)[i] \Rightarrow P(\sigma')[i] \}.
\end{array}
$$

(The first conjunct deals with send events; the second clause deals with receive events and the immediately succeeding state; the third clause deals with receive events and subsequent states: unfortunately the latter two clauses cannot be easily combined.)

The following rule shows how *maintains* P can be used to prove the maintenance of P:

Annotation Rule 27 (*maintains*).

$$
\begin{array}{l}
\{P\}\ \mathsf{send}\ \mathit{maintains}\ P\ \{P\} \\
\{P\}\ \mathsf{receive}\ \mathit{maintains}\ P\ \{P\}
\end{array}
$$

We will use the *maintains* P abstract message as a kind of magic, particularly for the maintenance of invariants: we will use it to specify that the invariant is maintained, without specifying the mechanism used to maintain it. Our experience is that reasoning about invariants requires reasoning about the protocol as a whole and so is best done separately from the annotation of a single role.

6.5 *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. This is most useful when the recipient can be sure that the intruder does not know x.

provesKnowledgeOf specifies nothing about who may learn data from this message.

6.5.1 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:

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

The following is an obvious extension:

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

Note that the abstract message *provesKnowledgeOf*$(x_1, \ldots, x_k)$ is not the same as *provesKnowledgeOf*$(x_1) \wedge \ldots \wedge$ *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 *provesKnowledgeOf*$(x_1, \ldots, x_k, id = b)$ tells the recipient that the other agent was following a role with identity variable b:

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

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:

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

6.5.2 Annotation rules

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

Annotation Rule 28 (provesKnowledgeOf.1).

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

Proof: Suppose

$$
\begin{array}{l}
\sigma_0(i).id = a \wedge \sigma_0(i).prog = es_0 \frown e' \frown es_1 \wedge \\
e' \sqsupseteq \mathsf{receive}\; provesKnowledgeOf(x_1, \ldots, x_k) \wedge \\
\forall\, \sigma \in States(\Pi) \mid \sigma(i).prog = e' \frown es_1 \bullet true(\sigma)[i].
\end{array}
$$

Let σ' be such that $\sigma'(i).prog = es_1$, and let σ be the state immediately before the event corresponding to e'. Let $X_l = \sigma'(i).\rho(x_l)$ for $l \in 1 \mathinner{..} k$. Then, from the semantics of *provesKnowledgeOf*(x), there are two possibilities:

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

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

  from the definition of *session*. So

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

  as required.

- Case for some $j > 0$, $j \neq i \wedge \forall l \in 1 \mathinner{\ldotp\ldotp} k \bullet \sigma(j).\rho(x_l) = X_l$. Let $b = \sigma(j).id$ and $B = \sigma(j).\rho(b)$. Then $session(b \rightsquigarrow B;\ x_1 \rightsquigarrow X_1, \ldots, x_k \rightsquigarrow X_k)(\sigma)$ and so

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

$\square$

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

Annotation Rule 29 (provesKnowledgeOf.2).

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

Annotation Rule 30 (provesKnowledgeOfNR.1).

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

Annotation Rule 31 (provesKnowledgeOfNR.2).

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

6.5.3 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 $provesKnowledgeOf(xs)$. For example, subject to the provisos

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

Further, any message containing such an encrypted component, possibly with additional fields, will refine the same abstract message. We will need the following lemma.[5]

Lemma 32. If $B \vdash M \wedge M' \ll M$ then $B \vdash M'$.

[5] Recall that $M' \ll M$ means that M' is a submessage of M that can be obtained from M simply by splitting pairs, i.e. without performing any decryption.

Refinement Rule 33. Suppose message template m is such that for some encrypted message template $m' = \{m''\}_y \trianglelefteq m$,

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

Then

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

provided:

1. the protocol satisfies the disjoint encryption property;
2. no role sends and then receives message templates that both contain m'; and
3. either (a) at least one field of m' is freshly generated by the recipient; or (b) the intruder does not initially hold 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 $l \in 1 \mathinner{..} k$. 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:

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

  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 $M' \trianglelefteq \sigma_0(0)$. 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.

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

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

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

- Case $M' \ntrianglelefteq \sigma_0(0)$. Then $M' \ntrianglelefteq \sigma''(0)$, since no subsequent message contains M', by assumption. Hence by Lemmas 18 and 32, 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:

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

  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 33 to deal with the *provesKnowledgeOf*$(xs, id = b)$ abstract message.

Refinement Rule 34. Suppose the conditions of Refinement Rule 33 are satisfied, and in addition only role b ever sends messages containing m', i.e.:

$$\forall\, j \in 1\,..\,n \bullet$$
$$(\exists\, m \bullet \textsf{send}\ m \text{ in } \sigma_0(j).prog \wedge m' \trianglelefteq m) \Rightarrow \sigma_0(j).id = b.$$

Then

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

The following two rules extend Refinement Rule 33 to deal with the *provesKnowledgeOfNR* abstract message.

Refinement Rule 35. Suppose the conditions of Refinement Rule 33 are satisfied, and in addition, for some a, b:

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

Then

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

Condition 7(b) could, under reasonable assumptions, be satisfied by taking x to be a shared secret or b's secret key, for example.

Refinement Rule 36. Suppose the conditions of Refinement Rule 35 are satisfied, except assumptions 6 and 7 are replaced by:

6′. $a \neq b$ is an invariant for b;

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

Then

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

Condition 7′(b) could, under reasonable assumptions, be satisfied by taking x to be a's public key, for example.

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 35:

Refinement Rule 37. Suppose message template m is such that for some hashed message template $m' = hash(m'') \trianglelefteq m$,

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

Then

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

provided:

1. the protocol satisfies the disjoint encryption property;
2. no role sends and then receives message templates that both contain m';
3. the intruder does not initially hold 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 for a;
7. $b \ll m'$.

Note that the above rule justifies the refinement

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

from the introductory example.

7 Maintaining invariants

In this section we state and prove some rules that can be used for verifying that an invariant is maintained. In particular, we give rules for showing that three particular classes of invariants are maintained: invariants dealing with long-term secrets that are not sent in the protocol; invariants dealing with short-term, freshly-generated secrets; and invariants dealing with the association between values.

7.1 Non-transmitted secrets

We give here a rule that can be used for verifying that the values of particular variables remain secret. More precisely, it deals with values that are never sent in any messages, such as long-term keys in most protocols. For such values x, we are interested in properties of the form

$$honest(as) \Rightarrow knows(x) \subseteq as,$$

i.e., x is known only by the agents in as, assuming they are honest; of course, if one of as is dishonest then we can deduce nothing: the intruder may pass on the value of x to other agents.

Invariant Rule 38. Consider some node i. Suppose

1. The protocol satisfies the disjoint encryption property.

2. No agent sends a message m such that $x \trianglelefteq m$.
3. Assuming as is honest, initially x is held only by as, and is uniquely bound:

$$\Big(honest(as) \Rightarrow holds(x) \subseteq as \wedge uniquelyBound(x)\Big)(\sigma_0)[i].$$

Then $honest(as) \Rightarrow holds(x) \subseteq as$ is an invariant for i, and hence $honest(as) \Rightarrow knows(x) \subseteq as$ is also an invariant.

Proof: Recall that $uniquelyBound(x)(\sigma_0)[i]$ implies that either x or x^{-1} is in $\operatorname{dom} \sigma_0(i)$. If $x \in \operatorname{dom} \sigma_0(i).\rho$ then let $X \mathrel{\widehat{=}} \sigma_0(i).\rho(x)$; otherwise, let $X \mathrel{\widehat{=}} (\sigma_0(i).\rho(x^{-1}))^{-1}$. Let $As \mathrel{\widehat{=}} \sigma_0(i).\rho(as)$. If $\neg\, honest(As)$ then the result holds trivially, so assume $honest(As)$.

We begin by showing that the intruder does not learn X; more precisely, we show $intruder \notin holds(X)$ in all states. This is true initially by assumption 3. Suppose, for a contradiction, that the intruder does come to hold X; then this will necessarily be from a send event, so suppose

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

with $intruder \in holds(X)(\sigma') - holds(X)(\sigma)$. Then necessarily $X \trianglelefteq M$. But in σ, the conditions of Theorem 19 hold, so $uniquelyBound(x \rightsquigarrow X)(\sigma)$. Hence X must instantiate x in M, which contradicts assumption 2. Hence the intruder never learns X.

Finally, no agent other than those in As learns X: X cannot be learnt from a new or $\mathsf{newpair}$ event, since such events generate fresh values; and since the intruder never holds X, we must have $X \not\trianglelefteq M$ for all messages M that are received. □

We can use the above rule to verify the invariant

$$honest(b) \Rightarrow knows(k) \subseteq \{a, b\}$$

from the introductory example of Section 2. This introduces an extra initial assumption, corresponding to Assumption 3, above:

$$honest(a, b) \Rightarrow holds(k) \subseteq \{a, b\} \wedge uniquelyBound(k).$$

Both of these conjuncts 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 $K \mathrel{\widehat{=}} \sigma(i).\rho(k)$, and assume $honest(A, B)$:

- 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 $knows(k) \subseteq \{a, b\}$, but is prevented by the additional assumption $holds(k) \subseteq \{a, b\}$.

- 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 *uniquely-Bound*(k) condition prevents this.

We also show that the invariant of Invariant Rule 38 is maintained by new events; this is necessary for use in annotations.

Annotation Rule 39. For $y \neq x$:

$$\begin{array}{l} \Big\{ honest(as) \Rightarrow knows(x) \subseteq as \Big\} \\ \mathsf{new}\; y \\ \Big\{ honest(as) \Rightarrow knows(x) \subseteq as \Big\} \end{array}$$

Note that we are assuming that x is in the initial state of the agent in question, so there cannot be new x events. This rule follows directly from Annotation Rule 16.

7.2 Transmitted secrets

We now prove a rule that is useful for verifying the secrecy of values that *are* transmitted by the protocol. For such values x, we are interested in properties of the form

$$defined(x) \wedge honest(as) \Rightarrow knows(x) \subseteq as.$$

It does not make sense to talk about who knows x before it is generated.

In essence, the rule below says that if the value x is always hashed, or encrypted with a key whose decrypting key remains secret, then x itself remains secret. However, the rule is slightly more complicated than one might expect. Consider the following protocol, which is intended to keep na secret:

$$\begin{array}{lll} \text{Message 1.} & a \rightarrow b & : \; a, \{na\}_{PK(b)} \\ \text{Message 2.} & b \rightarrow a & : \; \{b, na\}_{PK(a)} \end{array}$$

Suppose $honest(a, b)$. Then, from a's perspective, na is encrypted with either $PK(a)$ or $PK(b)$, both of whose decrypting keys are, presumably, secret. However, the intruder can replace a's identity in message 1 with his own, and thereby receive na encrypted with his own public key in the subsequent message 2, and so learn na.

The rule below avoids the problem exhibited above by insisting that the identities of the agents to whom the secret may be disclosed are associated with the secret itself (the $associatedWith_x(as')$ condition in assumption 3).

Invariant Rule 40. Consider some node i with role a, and some set of roles as. Suppose

1. The protocol satisfies the disjoint encryption property.
2. Either (a) initially only the agents as hold x, and do so well-bound:

   $$(holds(x) \subseteq as \land uniquelyBound(x))(\sigma_0)[i],$$

   or (b) role a generates x freshly.
3. Every occurrence of x in a message, say sent by role b, satisfies one of the following:
   (a) x is encrypted by some k such that for some set of roles $as' \subseteq as$:

   $honest(as') \Rightarrow knows(k^{-1}) \subseteq as'$ is invariant for b,
   and
   $honest(as) \Rightarrow associatedWith_x(as')(b)$ is invariant for i.

   (b) x is within a hash.

Then $defined(x) \land honest(as) \Rightarrow knows(x) \subseteq as$ is an invariant for i.

In assumption 3, we will normally 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.

Note that Invariant Rule 38 is a special case of this; assumption 3 holds vacuously under the assumptions of that rule.

Proof: Let $X \mathrel{\hat=} \sigma(i).\rho(x)$ (either the value held initially, or the value generated by i, depending upon which case of assumption 2 holds), $A = \sigma_0(i).\rho(a)$, and $As \mathrel{\hat=} \sigma_0(i).\rho(as)$. The result holds vacuously if $\neg\, honest(As)$, so suppose $honest(As)$.

We begin by showing that the intruder does not learn X. More precisely, for every state 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., $\sigma'(0) \nvdash K^{-1}$ for every σ' such that $\sigma \longrightarrow^* \sigma'$); or (b) hashed.

This is true initially by assumption 2. The only way it can become false subsequently is via a send event, so suppose for a contradiction

$$tr \frown \langle \sigma, j : \mathsf{send}\, M, \sigma' \rangle \in traces(\Pi), \qquad \text{with } \sigma(j).id = b,$$

and the above statement is true in σ but false in σ'. Then it must be that $X \trianglelefteq M$, not encrypted or hashed as above. But in σ, the conditions of Theorem 19 hold, so $uniquelyBound(x \rightsquigarrow X)(\sigma)$, so, in particular, $\sigma(j).\rho(x) = X$. Hence, by assumption 3, every occurrence of X in M satisfies one of the following:

(a) X is encrypted by some key K instantiating k such that $(knows(k^{-1}) \subseteq as')$ holds as an invariant for b, and $associatedWith_x(as')(b)(\sigma)[i]$. Then $\sigma(j).\rho(as') = \sigma(i).\rho(as') \subseteq \sigma(i).\rho(as) = As$; hence $knows(K^{-1}) \subseteq As$ invariably, so the intruder cannot learn K^{-1}, giving a contradiction.

(b) X is hashed; the result is immediate in this case.

Hence the intruder never learns X, so we can use Theorem 19, 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, and so we deduce that $defined(x) \wedge honest(as) \Rightarrow knows(x) \subseteq as$ is an invariant for i. □

We can apply the above rule to the example from Section 2 to show that

$$honest(b) \wedge defined(na) \Rightarrow knows(na) \subseteq \{a, b\}$$

is invariant. Note that the message $\{na\}_k$ satisfies condition 3(a): because a herself sends this message, the $associatedWith$ condition is automatically satisfied. The message $hash(na, b)$ satisfies condition 3(b).

We also show that the invariant of Invariant Rule 40 is maintained by new events.

Annotation Rule 41.

1. If $a \in as$ then

$$\begin{aligned} a : &\Big\{ defined(x) \wedge honest(as) \Rightarrow knows(x) \subseteq as \Big\} \\ &\mathsf{new}\ x \\ &\Big\{ defined(x) \wedge honest(as) \Rightarrow knows(x) \subseteq as \Big\} \end{aligned}$$

2. For $y \neq x$:

$$\begin{aligned} a : &\Big\{ defined(x) \wedge honest(as) \Rightarrow knows(x) \subseteq as \Big\} \\ &\mathsf{new}\ y \\ &\Big\{ defined(x) \wedge honest(as) \Rightarrow knows(x) \subseteq as \Big\} \end{aligned}$$

Note that in the first case, *defined*(x) will be false initially, so the precondition will be trivially true.

7.3 A rule for *associatedWith*

We now prove a rule that allows us to prove that certain $associatedWith_x(ys)$ properties hold as invariants. More precisely, we are interested in properties of the form

$$honest(as) \wedge knows(x) \subseteq as \Rightarrow associatedWith_x(ys).$$

Of course, if a dishonest agent learns the value of x, he can replay it to cause another agent to associate it with incorrect values for ys.

The main condition of the rule below is that whenever a role receives x for the first time, it must also receive each of ys, in an inseparable way.

Invariant Rule 42. Suppose that

1. The protocol satisfies the disjoint encryption property;
2. x is freshly generated by role a, in a state where it already has the variables ys bound;
3. Whenever a role b receives x for the first time, it is within a message template m such that for some encrypted message template $m' = \{m''\}_k \trianglelefteq m$,

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

   and if $x = k$, then it is a *symmetric* key.

Then

$$honest(as) \wedge knows(x) \subseteq as \Rightarrow associatedWith_x(ys)$$

is an invariant for a.

The following rule is a generalisation of the above rule: each variable y is replaced in the component m' by some variable $\hat{y}$ such that $\hat{y}$ is associated with y.

Invariant Rule 43. Suppose that conditions 1 and 2 of Invariant Rule 42 hold, and in addition

3. Suppose a role b receives x for the first time, in a message template m created by a role c. Then there is some encrypted message template $m' = \{m''\}_k \trianglelefteq m$, such that

   $$x \ll m'' \vee x = k = k^{-1}$$

And for each $y \in ys$, there is a variable $\hat{y}$ such that

$$\forall\, y \in ys \bullet \hat{y} \ll m'' \vee \hat{y} = k^{-1},$$

and $associatedWith_{\hat{y}}(y)(b)$ is an invariant for c.

Then

$$I \;\widehat{=}\; honest(as) \wedge knows(x) \subseteq as \Rightarrow associatedWith_x(ys)$$

is an invariant for a.

Rule 43 implies Rule 42 by taking $\hat{y} = y$; the $associatedWith_{\hat{y}}(y)$ clause then holds trivially.

Proof: Consider some node i with role a. We prove I is invariant for i by induction on the length of the trace. I holds before x is generated. By assumption 2, I holds immediately after x is generated.

So suppose I holds in state σ. Let $X = \sigma(i).\rho(x)$ and $As = \sigma(i).\rho(as)$. By the fact that send events do not change bindings, and the definition of *associatedWith*, we need only show that I is maintained by receive events that cause x to become bound to X.

So suppose event j : receive M leads from state σ to state σ', and causes x to be bound to X for j. By assumption 3, j must receive X in this message within an instantiation of m', namely $M' = m'[\sigma'(j).\rho]$. Suppose $(honest(As) \wedge knows(X) \subseteq As)(\sigma')$ (or else the result is immediate). Then the intruder does not know X in state σ, so by Lemma 18, M' must have been created by some honest node, say node l. By the disjoint encryption assumption, M' must have been created to instantiate m'. So $\sigma'(l).\rho(x) = X$. Hence, by the inductive hypothesis, $\sigma'(l).\rho(y) = \sigma'(i).\rho(y)$, for each $y \in ys$. Also, $\sigma'(l).\rho(\hat{y}) = \sigma'(j).\rho(\hat{y})$ since m' contains $\hat{y}$, by assumption 3. Hence, by the $associatedWith_{\hat{y}}(y)(b)$ assumption, $\sigma'(l).\rho(y) = \sigma'(j).\rho(y)$. Putting the results together, we get $\sigma'(j).\rho(y) = \sigma'(i).\rho(y)$, as required. $\square$

We also show that the above invariant is maintained by new events. The rules require ys to be defined *before* x.

Annotation Rule 44.

$$\left\{ defined(ys) \wedge (honest(as) \wedge knows(x) \subseteq as \Rightarrow associatedWith_x(ys)) \right\}$$
new x
$$\left\{ honest(as) \wedge knows(x) \subseteq as \Rightarrow associatedWith_x(ys) \right\}$$

For $y \neq x$ and $y \in ys$:

$$\left\{ \neg\, defined(x) \wedge (honest(as) \wedge knows(x) \subseteq as \Rightarrow associatedWith_x(ys)) \right\}$$
new y
$$\left\{ honest(as) \wedge knows(x) \subseteq as \Rightarrow associatedWith_x(ys) \right\}$$

For $z \notin \{x\} \cup ys$:

$$\Big\{ honest(as) \wedge knows(x) \subseteq as \Rightarrow associatedWith_x(ys) \Big\}$$
$$\textsf{new } z$$
$$\Big\{ honest(as) \wedge knows(x) \subseteq as \Rightarrow associatedWith_x(ys) \Big\}$$

Comment Note that Invariant Rule 40, for *knows*, has a premise that talks about *associatedWith*; and conversely Invariant Rule 42, for *associatedWith*, has a premise that talks about *knows*. Can these two rules be used together, or would that constitute circular reasoning? The proof of the rule for *associatedWith* requires that the secrecy condition holds in one state in order for the association to hold in the *next* state; and similarly, the proof of the rule for *knows* requires the association to hold in one state in order for the secrecy condition to hold in the *next* state. So if both hold, in one state, then they will both hold in the following state, and so on inductively. Hence the two rules may be used together.

8 Examples

We illustrate the calculus by applying it to three well-known protocols, the Adapted Needham Schroeder Public Key Protocol, the Otway Rees Protocol, and the Yahalom Protocol.

8.1 The Needham Schroeder Public Key Protocol

In this section we give a derivation of the Adapted Needham Schroeder Public Key Protocol [Low95], as in Figure 3. We give derivations for both roles of the protocol.

The protocol works by combining two public-key encrypted nonce challenges. Identity information is included to ensure that the nonces are associated with the correct identities, to avoid man-in-the-middle attacks.

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 $skb = pkb^{-1}$.

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 agents, and assuming b is honest, only the appropriate agents know the secret keys, that only a and b learn na, and that na is associated with a:

$$\begin{aligned}
I_a \;\widehat{=}\;& a \neq b \land \\
& \mathit{knows}(\mathit{ska}) = \{a\} \land \\
& (\mathit{honest}(b) \Rightarrow \mathit{knows}(\mathit{pkb}^{-1}) = \{b\}) \land \\
& (\mathit{honest}(b) \land \mathit{defined}(na) \Rightarrow \mathit{knows}(na) \subseteq \{a, b\}) \land \\
& (\mathit{honest}(b) \Rightarrow \mathit{associatedWith}_{na}(a)).
\end{aligned}$$

(Note that a's state does not include b's secret key skb, so the invariant talks about pkb^{-1} instead.) We assume that the first three conjuncts of the invariant hold initially; na is not defined initially, so the last two conjuncts hold vacuously:

$$\mathit{pre}_a \;\widehat{=}\; a \neq b \land \mathit{knows}(\mathit{ska}) = \{a\} \land (\mathit{honest}(b) \Rightarrow \mathit{knows}(\mathit{pkb}^{-1}) = \{b\}).$$

$$\begin{array}{l}
\mathit{Initiator}(a;\ b, \mathit{pka}, \mathit{ska}, \mathit{pkb}) \;\widehat{=} \\
\Big\{\mathit{pre}_a\Big\}\ \Big\{I_a\Big\} \\
\mathsf{new}\ na\ \Big\{I_a \Big\langle \text{Annotation Rules 39, 41 and 44} \Big\rangle\Big\} \\
\mathsf{send}\ \mathit{maintains}\ I_a \land \mathit{contains}\ na\ \Big\{I_a\Big\} \\
\mathsf{receive}\ \mathit{maintains}\ I_a \land \mathit{provesKnowledgeOf}(na, nb, b, id = b) \\
\Big\{I_a \land \exists\, B \bullet \mathit{session}(b \rightsquigarrow B;\ na, nb, b) \Big\langle \text{Annotation Rule 29} \Big\rangle\Big\} \\
\left\{\begin{array}{l} I_a \land (\mathit{honest}(b) \Rightarrow \mathit{session}(b;\ na, nb, a)) \\ \left\langle\begin{array}{l} \text{if } \mathit{honest}(b) \text{ then } \mathit{knows}(na) \subseteq \{a, b\} \text{ so } B \neq \mathit{intruder}\text{, above;} \\ \mathit{associatedWith}_{na}(a) \text{ from invariant; Lemma 15} \end{array}\right\rangle \end{array}\right\} \\
\mathsf{send}\ \mathit{maintains}\ I_a \\
\Big\{I_a \land (\mathit{honest}(b) \Rightarrow \mathit{session}(b;\ na, nb, a))\Big\}
\end{array}$$

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

An annotation of the protocol for agent a is shown in Figure 1; justifications are given in angle brackets. The main step is that a sends a message that contains na, and then 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; without the $\mathit{associatedWith}_{na}(a)$ clause of the invariant, we would not be able to make this latter deduction, and we would end up with a much weaker authentication guarantee. Note that b's session must be recent, and correspond to a single session of a, because of the agreement on the fresh variable na.

For b's perspective, the invariant is very similar to that for a:

$$\begin{aligned} I_b \;\widehat{=}\;& honest(a) \Rightarrow knows(pka^{-1}) = \{a\} \wedge \\ & knows(skb) = \{b\} \wedge \\ & honest(a) \wedge defined(nb) \Rightarrow knows(nb) \subseteq \{a, b\} \wedge \\ & honest(a) \Rightarrow associatedWith_{nb}(b, na), \end{aligned}$$

We assume the first two conjuncts of the invariant:

$$pre_b \;\widehat{=}\; (honest(a) \Rightarrow knows(pka^{-1}) = \{a\}) \wedge knows(skb) = \{b\}$$

The protocol from b's perspective is shown in Figure 2. The main step is that b sends a message that contains nb, and 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. Note that a's session must be recent, and correspond to a single session of b, because of the agreement on the fresh variable nb.

$Responder(b;\; a, pkb, skb, pka) \;\widehat{=}$
$\{pre_b\}\ \{I_b\}$
receive $maintains\ I_b\ \{I_b\}$
new $nb\ \{I_b\}$
send $maintains\ I_b \wedge contains\ nb\ \{I_b\}$
receive $maintains\ I_b \wedge provesKnowledgeOfNR(nb, id = a)$
$\{I_b \wedge \exists A \bullet session(a \rightsquigarrow A;\; nb \rightsquigarrow nb) \wedge A \neq b\ \langle\text{Annotation Rule 31}\rangle\}$
$$\left\{\begin{array}{l} I_b \wedge (honest(a) \Rightarrow session(a;\; nb, na, b)) \\ \left\langle\begin{array}{l}\text{if } honest(a) \text{ then } knows(nb) \subseteq \{a, b\} \text{ so } A = a \text{ above;} \\ associatedWith_{nb}(b, na) \text{ from invariant; Lemma 15}\end{array}\right\rangle \end{array}\right\}$$

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

We can strengthen the postconditions in the two annotations, using Lemma 14. Recall that

$$honest(b) \wedge defined(na) \Rightarrow knows(na) \subseteq \{a, b\}$$

is invariant for a. Hence we can use Lemma 14 to deduce that

$$\begin{aligned} & session(a;\; b, na, nb) \wedge honest(a) \Rightarrow \\ & \quad (honest(b) \wedge defined(na) \Rightarrow knows(na) \subseteq \{a, b\}) \end{aligned}$$

is invariant for b. Combining with b's postcondition, and simplifying, we see that

$$honest(a) \Rightarrow knows(na) \subseteq \{a, b\}$$

is a postcondition for b. Likewise, we may add

$$honest(b) \Rightarrow knows(nb) \subseteq \{a, b\}$$

to the postcondition in the annotation for a.

Putting the two annotations together, we may refine the protocol to obtain the normal definition, as in Figure 3. We have a number of proof obligations in order to justify this.

Message 1. $a \rightarrow b \ : \ \{a, na\}_{pkb}$
Message 2. $b \rightarrow a \ : \ \{b, na, nb\}_{pka}$
Message 3. $a \rightarrow b \ : \ \{nb\}_{pkb}$.

Figure 3: The Adapted Needham Schroeder Public Key Protocol: concrete version

Firstly, we can use Invariant Rule 38 to show that each message maintains the parts of the invariants dealing with the secret keys. This introduces the following additional initial assumptions for a:

$$holds(ska) \subseteq \{a\} \wedge uniquelyBound(ska) \wedge$$
$$honest(b) \Rightarrow holds(pkb^{-1}) \subseteq \{b\} \wedge uniquelyBound(pkb^{-1}),$$

and symmetric assumptions for b. (Recall that $uniquelyBound(pkb^{-1})$ means that the inverse of the value of a's variable pkb is stored only in other nodes' variable skb.)

Next, we can use Invariant Rule 40 to show that each message maintains the part of a's invariant concerning $knows(na)$. 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 $associatedWith_{na}(a)$ clause of a's invariant.

We can show that each message maintains the part of b's invariant concerning $knows(nb)$ in a very similar way.

Next we can use Invariant Rule 42 to show that

$$honest(a, b) \wedge knows(na) \subseteq \{a, b\} \Rightarrow associatedWith_{na}(a)$$

is invariant for a, in particular, because a is included in the encrypted component of message 1, where b first receives na. Hence

$$honest(b) \Rightarrow associatedWith_{na}(a)$$

is invariant, because of the earlier invariant about na.

We can similarly use Invariant Rule 42 to show that

$$honest(a) \Rightarrow associatedWith_{nb}(b, na)$$

is invariant for b, because b and na are included in the encrypted component of message 2, where a first receives nb.

Next, we can show

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

using Refinement Rule 34, noting that na is freshly generated by the recipient a. We can similarly show

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

using Refinement Rule 36, noting that nb is freshly generated by the recipient b, that $a \neq b$ is an invariant for a, and that $associated_{nb}(b)$ is an invariant for b.

Of course, that's not the only way to refine the abstract protocol. For example, one can also refine it to:

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

The "1" and "2" are message tags, to enforce disjoint encryption[6]. (To verify this refinement, one would need an extension of Invariant Rule 42 dealing with hash functions.)

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 invariant, the *associatedWith* statement would be replaced by $associatedWith_{nb}(b, na)$; 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

[6] In order to make this fit our definition of disjoint encryption, we need to assume that $type_{val}(1) \neq type_{val}(2)$.

and na, but cannot be sure that a associates that session with him. Further, we wouldn't be able to prove that the messages keep nb secret, 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 [Low95].

8.2 The Otway Rees Protocol

We now give a derivation of a variant of the Otway Rees Protocol [OR87a]. We vary the protocol slightly, so as to enforce the disjoint encryption condition: see Figure 7.

The protocol aims to establish a shared key kab between two agents, a and b, with the help of a trusted third party s, with whom a and b share long-term keys kas and kbs, respectively. Each of a and b creates a fresh nonce, na and nb, respectively, which stands for the other's identity in the key delivery message. a creates a second nonce, m, which acts as a run identifier.

We begin by considering the invariant from a's perspective. The long-term key kas is a secret shared between a and s. Further, it is necessary to keep na secret, in order that it can stand for b's identity in the key delivery message. Finally, na is associated with a, b and m.

$$\begin{aligned}
I_a \;\hat{=}\;& \mathit{honest}(s) \Rightarrow \mathit{knows}(\mathit{kas}) = \{a, s\} \wedge \\
& \mathit{honest}(s) \wedge \mathit{defined}(\mathit{na}) \Rightarrow \mathit{knows}(\mathit{na}) \subseteq \{a, s\} \wedge \\
& \mathit{honest}(s) \Rightarrow \mathit{associatedWith}_{\mathit{na}}(a, b, m), \\
\mathit{pre}_a \;\hat{=}\;& \mathit{honest}(s) \Rightarrow \mathit{holds}(\mathit{kas}) = \{a, s\} \wedge \mathit{uniquelyBound}(\mathit{kas}).
\end{aligned}$$

We will use Invariant Rule 38 to prove the secrecy of kas; the precondition anticipates that, by assuming one of the conditions of that rule. Note that the *uniquelyBound* clause implies that a particular value can never be used to instantiate both kas and kbs, even in different local states; in particular, this means that a different key should be used by a particular agent in the a and b roles. We do not believe this condition is strictly necessary, and could be avoided by a generalisation of Invariant Rule 38 that talks about multiple variables; we leave this to future work.

The annotation for a is in Figure 4. The main step is that a receives a message that proves knowledge of kab and na; a can deduce that s has a session, in which he associates kab with a and b.

b's perspective is very similar to a's, so we simply sketch the details. The invariant and precondition are as follows:

$$\begin{aligned}
I_b \;\hat{=}\;& \mathit{honest}(s) \Rightarrow \mathit{knows}(\mathit{kbs}) = \{b, s\} \wedge \\
& \mathit{honest}(s) \wedge \mathit{defined}(\mathit{nb}) \Rightarrow \mathit{knows}(\mathit{nb}) \subseteq \{b, s\} \wedge \\
& \mathit{honest}(s) \Rightarrow \mathit{associatedWith}_{\mathit{nb}}(a, b, m),
\end{aligned}$$

$$
\begin{array}{l}
\mathit{Initiator}(a;\ b, s, kas) \mathrel{\widehat{=}} \\
\left\{\mathit{pre}_a\right\}\ \left\{I_a\right\} \\
\textsf{new}\ na, m\ \ \left\{I_a\right\} \\
\textsf{send}\ \mathit{maintains}\ I_a \wedge \mathit{contains}\ na \wedge \mathit{contains}\ m\ \ \left\{I_a\right\} \\
\textsf{receive}\ \mathit{maintains}\ I_a \wedge \mathit{provesKnowledgeOfNR}(na, kab, id = s) \\
\left\{I_a \wedge \exists\, S \neq a \bullet \mathit{session}(s \rightsquigarrow S;\ na, kab) \left\langle \text{Annotation Rule 31} \right\rangle\right\} \\
\left\{\begin{array}{l}
I_a \wedge (\mathit{honest}(s) \Rightarrow \mathit{session}(s;\ a, b, na, m, kab)) \\
\left\langle\begin{array}{l}
\text{if } \mathit{honest}(s) \text{ then } \mathit{knows}(na) \subseteq \{a, s\} \text{ so } S = s; \\
\mathit{associatedWith}_{na}(a, b, m) \text{ from } I_a\text{; Lemma 15}
\end{array}\right\rangle
\end{array}\right\}
\end{array}
$$

Figure 4: The Otway Rees Protocol: a's perspective

$$
\mathit{pre}_b \mathrel{\widehat{=}} \mathit{honest}(s) \Rightarrow \mathit{holds}(kbs) = \{b, s\} \wedge \mathit{uniquelyBound}(kbs).
$$

The annotation is in Figure 5. Compared with a, b has an extra initial receive and final send; these are mainly for forwarding messages in the final protocol.

$$
\begin{array}{l}
\mathit{Responder}(b;\ a, s, kbs) \mathrel{\widehat{=}} \\
\left\{\mathit{pre}_b\right\}\ \left\{I_b\right\} \\
\textsf{receive}\ \mathit{maintains}\ I_b\ \ \left\{I_b\right\} \\
\textsf{new}\ nb\ \ \left\{I_b\right\} \\
\textsf{send}\ \mathit{maintains}\ I_b \wedge \mathit{contains}\ nb\ \ \left\{I_b\right\} \\
\textsf{receive}\ \mathit{maintains}\ I_b \wedge \mathit{provesKnowledgeOfNR}(nb, kab, id = s) \\
\left\{I_b \wedge \exists\, S \neq b \bullet \mathit{session}(s \rightsquigarrow S;\ nb, kab)\right\} \\
\left\{I_b \wedge \mathit{session}(s;\ a, b, nb, m, kab)\right\} \\
\textsf{send}\ \mathit{maintains}\ I_b \\
\left\{I_b \wedge \mathit{session}(s;\ a, b, nb, m, kab)\right\}
\end{array}
$$

Figure 5: The Otway Rees Protocol: b's perspective

We now consider the perspective of s. The long-term keys are secrets, as above. The session key kab is a secret shared between a, b and s.

$$
\begin{aligned}
I_s \mathrel{\widehat{=}}\ & \mathit{honest}(a) \Rightarrow \mathit{knows}(kas) \subseteq \{a, s\} \wedge \\
& \mathit{honest}(b) \Rightarrow \mathit{knows}(kbs) \subseteq \{b, s\} \wedge \\
& \mathit{honest}(a, b) \wedge \mathit{defined}(kab) \Rightarrow \mathit{knows}(kab) \subseteq \{a, b, s\},
\end{aligned}
$$

$$
\begin{aligned}
pre_s \;\widehat{=}\; & honest(a) \Rightarrow holds(kas) = \{a, s\} \wedge uniquelyBound(kas) \wedge \\
& honest(b) \Rightarrow holds(kbs) = \{b, s\} \wedge uniquelyBound(kbs).
\end{aligned}
$$

The annotation is in Figure 6. s receives a message authenticating a and b, and sends a message that contains the session key kab.

$$
\begin{array}{l}
Server(s;\ a, b, kas, kbs) \;\widehat{=} \\
\left\{pre_s\right\}\ \left\{I_s\right\} \\
\textsf{receive}\ maintains\ I_s \wedge provesKnowledgeOfNR(b, kas, na, m, id = a) \\
\qquad\qquad\qquad\qquad \wedge\ provesKnowledgeOfNR(a, kbs, nb, m, id = b) \\
\left\{\begin{array}{l} I_s \wedge \exists\, A \neq s \bullet session(a \leadsto A;\ b, kas, na, m) \wedge \\ \exists\, B \neq s \bullet session(b \leadsto B;\ a, kbs, nb, m) \end{array}\right\} \\
\left\{I_s \wedge session(a;\ b, kas, na, m) \wedge session(b;\ a, kbs, nb, m)\right\} \\
\textsf{new}\ kab \\
\textsf{send}\ maintains\ I_s \wedge contains\ kab \\
\left\{I_s \wedge session(a;\ b, kas, na, m) \wedge session(b;\ a, kbs, nb, m)\right\}
\end{array}
$$

Figure 6: The Otway Rees Protocol: s's perspective

We can now apply Lemma 14 to s's invariant to strengthen the postconditions for a and b with the condition

$$honest(a, b, s) \Rightarrow knows(kab) \subseteq \{s, a, b\},$$

i.e., a and b receive a guarantee of secrecy.

We now refine the abstract messages, to obtain the concrete protocol described in standard notation in Figure 7.

$$
\begin{array}{lccl}
\text{Message 1.} & a \rightarrow b & : & m, a, b, \{na, m, a, b\}_{kas} \\
\text{Message 2.} & b \rightarrow s & : & m, a, b, \{na, m, a, b\}_{kas}, \{a, b, nb, m\}_{kbs} \\
\text{Message 3.} & s \rightarrow b & : & m, \{na, kab\}_{kas}, \{kab, nb\}_{kbs} \\
\text{Message 4.} & b \rightarrow a & : & m, \{na, kab\}_{kas}
\end{array}
$$

Figure 7: The Otway Rees Protocol: concrete version

Note that message 1 contains an encrypted component that the recipient, b, is unable to decrypt, but which is simply forwarded to s in the following message; and likewise with message 3. In our language of protocols (*Prog*), b's role would be written as:

$\textsf{receive}\ m, a, b, x$
$\textsf{new}\ nb$
$\textsf{send}\ m, a, b, x, \{a, b, nb, m\}_{kbs}$
$\textsf{receive}\ m, y, \{kab, nb\}_{kbs}$
$\textsf{send}\ m, y$

The variables x and y get bound to the encrypted components in a normal run.

We have rearranged the order of the fields in the second encrypted components of messages 2 and 3 to enforce the disjoint encryption property, in particular to ensure that those components have a different form from the other components in the same messages.

We have a number of proof obligations. Firstly, we can use Invariant Rule 38 to show that each message maintains the parts of the invariants dealing with the secrecy of kas and kbs.

Next, we can use Invariant Rule 40 to show that each message satisfies the part of s's invariant dealing with the secrecy of kab. Note that each message that contains kab is sent by s, and encrypted with kas or kbs. We have $knows(kas) \subseteq \{a, s\}$ is invariant for s, and clearly $associatedWith_{kab}(\{a, s\})(s)$ is invariant for s; and likewise for kbs.

We can similarly use Invariant Rule 40 to show that each message keeps na secret, from a's perspective. Each message that contains na is encrypted with kas. We have $knows(kas) \subseteq \{a, s\}$ is invariant for both a and s. Also, $associatedWith_{kas}(\{a, s\})(a)$ clearly holds as an invariant for a. However, the proof reveals an additional initial assumption for a:

$$associatedWith_{kas}(\{a, s\})(s),$$

i.e., for any instance i of the role a, any instance of s that has its kas variable bound to the same as i's kas variable, also has its a variable bound to the same as i's a variable — s uses the right long-term keys with the right agents!

The proof that each message keeps nb secret is identical, and reveals a corresponding initial assumption for b:

$$associatedWith_{kbs}(\{b, s\})(s).$$

Next we can use Invariant Rule 42 to show that

$$honest(s) \Rightarrow associatedWith_{na}(a, b, m)$$

is invariant for a, in particular, from the component $\{na, m, a, b\}_{kas}$. We can similarly use Invariant Rule 42 to show that

$$honest(s) \Rightarrow associatedWith_{nb}(a, b, m)$$

is invariant for b, from the component $\{a, b, nb, m\}_{kbs}$.

We can use Refinement Rule 35 to show that message 2 refines $provesKnowledgeOfNR(b, kas, na, m, id = a)$, in particular from the component $\{na, m, a, b\}_{kas}$ (for condition 3(b) of Refinement Rule 33, we need to assume that the intruder does not initially know any such component, in particular any such component encrypted with the value of kas in question; otherwise he could immediately fake such a component). We can similarly show that message 2 refines $provesKnowledgeOfNR(a, kbs, nb, m, id = b)$.

Finally, we can show that message 4 refines $provesKnowledgeOfNR(na, kab, id = s)$ using Refinement Rule 35; note that condition 4(b) is satisfied, because $associatedWith_{na}(b)$ is an invariant for a. Similarly, message 3 refines $provesKnowledgeOfNR(nb, kab, id = s)$.

8.3 The Yahalom Protocol

We now consider the Yahalom Protocol, as described in Figure 8. Our derivation of the protocol is more complicated than those of the previous two protocols, because of some subtleties of the protocol. Several variables are used in place of agents' identities at various points in the protocol: nb is used to stand for a, from b's point of view, in the second component of message 4; kas is used to stand for a and/or s at various points in the protocol; and likewise kbs is used to stand for b and/or s. These associations need to be captured within the invariants. Further, the need for several parts of the invariants only becomes apparent in later parts of the proof: coming up with the correct invariants was very much an iterative process.

$$\begin{array}{lll}
\text{Message 1.} & a \rightarrow b & : a, na \\
\text{Message 2.} & b \rightarrow s & : b, \{a, na, nb\}_{kbs} \\
\text{Message 3.} & s \rightarrow a & : \{b, kab, na, nb\}_{kas}, \{a, kab\}_{kbs} \\
\text{Message 4.} & a \rightarrow b & : \{a, kab\}_{kbs}, \{nb\}_{kab}
\end{array}$$

Figure 8: The Yahalom Protocol: concrete version

We begin by considering a's perspective. The invariant states that kas is a shared secret between a and s, and that s associates kas with a. Anticipating a use of Invariant Rule 38, the precondition assumes one of the conditions of that rule.

$$\begin{aligned} I_a \;&\widehat{=}\; \mathit{honest}(s) \Rightarrow \mathit{knows}(\mathit{kas}) \subseteq \{a, s\} \wedge \\ &\quad \mathit{associatedWith}_{\mathit{kas}}(a)(s), \\ \mathit{pre}_a \;&\widehat{=}\; \mathit{honest}(s) \Rightarrow \mathit{holds}(\mathit{kas}) \subseteq \{a, s\} \wedge \mathit{uniquelyBound}(\mathit{kas}) \wedge \\ &\quad \mathit{associatedWith}_{\mathit{kas}}(a)(s). \end{aligned}$$

The annotation from a's perspective is in Figure 9. The main step is that a receives a message that proves knowledge of b, kab, na, nb, kas; a can deduce that the message came from s, because of the use of the shared secret kas; further, a can deduce that s is in a session with a, because of the $\mathit{associatedWith}_{\mathit{kas}}(a)(s)$ assumption.

$$\begin{array}{l} \mathit{Initiator}(a;\; b, s, \mathit{kbs}) \;\widehat{=} \\ \left\{\mathit{pre}_a\right\} \left\{I_a\right\} \\ \mathsf{new}\; na \left\{I_a\right\} \\ \mathsf{send}\; \mathit{maintains}\; I_a \wedge \mathit{contains}\; na \left\{I_a\right\} \\ \mathsf{receive}\; \mathit{maintains}\; I_a \wedge \mathit{provesKnowledgeOfNR}(b, \mathit{kab}, na, nb, \mathit{kas}, id = s) \\ \left\{\begin{array}{l} I_a \wedge \exists\, S \neq a \bullet \mathit{session}(s \rightsquigarrow S;\; b, \mathit{kab}, na, nb, \mathit{kas}) \\ \left\langle \text{Annotation Rule 31} \right\rangle \end{array}\right\} \\ \left\{\begin{array}{l} I_a \wedge (\mathit{honest}(s) \Rightarrow \mathit{session}(s;\; a, b, \mathit{kab}, na, nb, \mathit{kas})) \\ \left\langle\begin{array}{l} \text{if } \mathit{honest}(s) \text{ then } \mathit{knows}(\mathit{kas}) \subseteq \{a, s\} \text{ so } S = s; \\ \mathit{associatedWith}_{\mathit{kas}}(a)(s) \text{ from } I_s \end{array}\right\rangle \end{array}\right\} \\ \mathsf{send}\; \mathit{maintains}\; I_a \\ \left\{I_a \wedge (\mathit{honest}(s) \Rightarrow \mathit{session}(s;\; a, b, \mathit{kab}, na, nb, \mathit{kas}))\right\} \end{array}$$

Figure 9: The Yahalom Protocol: a's perspective

The invariant for b says: that b and s are distinct agents; that kbs is a shared secret between b and s; that s associates kbs with b and s[7]; that nb is a shared secret between a, b and s; and that all agents who hold nb associate

[7]This association means that an agent cannot act in both the roles b and s, using the same key for kbs in each case; if this assumption is not made then there is an attack against the protocol.

it with a, b and s:

$$\begin{aligned}
I_b \;\widehat{=}\;& b \neq s \wedge \\
& honest(s) \Rightarrow knows(kbs) \subseteq \{b, s\} \wedge \\
& associatedWith_{kbs}(b)(s) \wedge associatedWith_{kbs}(s)(s) \wedge \\
& honest(a, s) \wedge defined(nb) \Rightarrow \\
& \quad knows(nb) \subseteq \{a, b, s\} \wedge associatedWith_{nb}(a, b, s), \\
pre_b \;\widehat{=}\;& b \neq s \wedge \\
& honest(s) \Rightarrow holds(kbs) \subseteq \{b, s\} \wedge uniquelyBound(kbs) \wedge \\
& associatedWith_{kbs}(b)(s) \wedge associatedWith_{kbs}(s)(s).
\end{aligned}$$

The annotation for b is as in Figure 10. The crucial step is that b receives a message that shows there is a session of s using a and kab; and also shows that there is a session of somebody, A say, in the role of a using nb and kab: the fact that nb is associated with a allows us to deduce that A is a.

$$\begin{array}{l}
Responder(b;\; a, s, kbs) \;\widehat{=} \\
\left\{pre_b\right\} \left\{I_b\right\} \\
\textsf{receive}\; maintains\; I_b \left\{I_b\right\} \\
\textsf{new}\; nb \left\{I_b\right\} \\
\textsf{send}\; maintains\; I_b \wedge contains\; nb \left\{I_b\right\} \\
\textsf{receive}\; maintains\; I_b \wedge provesKnowledgeOfNR(a, kab, kbs, id = s) \\
\qquad \wedge\; provesKnowledgeOf(nb, kab, id = a) \\
\left\{\begin{array}{l}
I_b \wedge \exists\, S \neq b \bullet session(s \rightsquigarrow S;\; a, kab, kbs) \wedge \left\langle \text{Annotation Rule 31} \right\rangle \\
\exists\, A \bullet session(a \rightsquigarrow A;\; nb, kab) \left\langle \text{Annotation Rule 29} \right\rangle
\end{array}\right\} \\
\left\{\begin{array}{l}
I_b \wedge \\
honest(s) \Rightarrow session(s;\; a, b, kab, kbs) \wedge \\
\left\langle \begin{array}{l} \text{if } honest(s) \text{ then } knows(kbs) \subseteq \{b, s\}, \text{ so } S = s; \\ associatedWith_{kbs}(b)(s) \text{ so } b \rightsquigarrow b \end{array} \right\rangle \\
honest(a, s) \Rightarrow session(a;\; nb, kab) \\
\left\langle \text{if } honest(a, s) \text{ then } associatedWith_{nb}(a) \text{ so } A = a \right\rangle
\end{array}\right\}
\end{array}$$

Figure 10: The Yahalom Protocol: b's perspective

The invariant for s says: s is distinct from a and b; kas, kbs and kab are shared secrets between the appropriate agents; and that a associates kas

with a and s:

$$\begin{aligned}
I_s \;\widehat{=}\; & a \neq s \wedge b \neq s \wedge \\
& \mathit{honest}(a) \Rightarrow \mathit{knows}(\mathit{kas}) \subseteq \{a, s\} \wedge \\
& \mathit{honest}(b) \Rightarrow \mathit{knows}(\mathit{kbs}) \subseteq \{b, s\} \wedge \\
& \mathit{associatedWith}_{\mathit{kas}}(s)(a) \wedge \mathit{associatedWith}_{\mathit{kas}}(a)(a) \\
& \mathit{honest}(a, b) \wedge \mathit{defined}(\mathit{kab}) \Rightarrow \mathit{knows}(\mathit{kab}) \subseteq \{a, b, s\}, \\
\mathit{pre}_s \;\widehat{=}\; & a \neq s \wedge b \neq s \wedge \\
& \mathit{honest}(a) \Rightarrow \mathit{holds}(\mathit{kas}) \subseteq \{a, s\} \wedge \mathit{uniquelyBound}(\mathit{kas}) \wedge \\
& \mathit{honest}(b) \Rightarrow \mathit{holds}(\mathit{kbs}) \subseteq \{b, s\} \wedge \mathit{uniquelyBound}(\mathit{kbs}) \wedge \\
& \mathit{associatedWith}_{\mathit{kas}}(s)(a) \wedge \mathit{associatedWith}_{\mathit{kas}}(a)(a).
\end{aligned}$$

The annotation is as in Figure 11. The crucial step is that s receives a message which he can deduce came from b, using a, na and nb.

$$\begin{array}{l}
\left\{\mathit{pre}_s\right\} \left\{I_s\right\} \\
\mathsf{receive}\ \mathit{maintains}\ I_s \wedge \mathit{provesKnowledgeOfNR}(a, na, nb, kbs, id = b) \\
\left\{I_s \wedge \exists\, B \neq s \bullet \mathit{session}(b \rightsquigarrow B;\ a, na, nb, kbs)\right\} \\
\left\{\begin{array}{l} I_s \wedge (\mathit{honest}(b) \Rightarrow \mathit{session}(b;\ a, na, nb, kbs)) \\ \left\langle \text{if } \mathit{honest}(b) \text{ then } \mathit{knows}(kbs) \subseteq \{b, s\} \text{ so } B = b \right\rangle \end{array}\right\} \\
\mathsf{new}\ \mathit{kab} \\
\mathsf{send}\ \mathit{maintains}\ I_s \wedge \mathit{contains}\ \mathit{kab} \\
\left\{I_s \wedge (\mathit{honest}(b) \Rightarrow \mathit{session}(b;\ a, na, nb, kbs))\right\}
\end{array}$$

Figure 11: The Yahalom Protocol: s's perspective

We can now apply Lemma 14 to s's invariant to strengthen the postconditions of a and b with

$$\mathit{honest}(a, b, s) \Rightarrow \mathit{knows}(\mathit{kab}) \subseteq \{a, b, s\}.$$

We now refine the abstract messages to obtain the concrete protocol of Figure 8.

We use Invariant Rule 38 to verify the parts of the invariants dealing with the secrecy of *kas* and *kbs*: condition 3 is satisfied because of the preconditions.

We then use Invariant Rule 40 to verify the part of s's invariant dealing with the secrecy of *kab*: condition 3 is satisfied because only s sends messages containing *kab*, and they are encrypted with *kas* or *kbs*, which satisfy the relevant conditions.

We can likewise use Invariant Rule 40 to verify the part of b's invariant dealing with the secrecy of nb; we check each message containing nb in turn:

- nb is encrypted with kbs in message 2; $honest(b, s) \Rightarrow knows(kbs) \subseteq \{b, s\}$ is invariant for b; $associatedWith_{nb}(b, s)(b)$ is invariant for b.

- nb is encrypted with kas in message 3; $honest(a, s) \Rightarrow knows(kas) \subseteq \{a, s\}$ is invariant for s; $associatedWith_{nb}(a, s)(s)$ is invariant for b.

- nb is encrypted with kab in message 4; $honest(a, b, s) \Rightarrow knows(kab) \subseteq \{a, b, s\}$ is invariant for a; $associatedWith_{nb}(a, b, s)(a)$ is invariant for b.

The various $associatedWith_{kas}$ and $associatedWith_{kbs}$ clauses hold because they hold initially, and kas and kbs never get rebound.

We can use Invariant Rule 43 to verify that the $associatedWith_{nb}(a, b, s)$ clause of b's invariant is maintained:

- When s first receives nb, in message 2, it is in an encrypted component, created by b, that contains a, and is encrypted with kbs (kbs stands as an alias for b and s; using the notation of Rule 43, take $\hat{a} = a$, and $\hat{b} = \hat{s} = kbs$); note that the side conditions hold: $associatedWith_{kbs}(b)(s)$ and $associatedWith_{kbs}(s)(s)$ are invariant for b.

- When a first receives nb, in message 3, it is in an encrypted message, created by s, that contains b, and is encrypted by kas (kas stands as an alias for a and s); note that the side conditions hold: $associatedWith_{kas}(s)(a)$ and $associatedWith_{kas}(a)(a)$ are invariant for s.

Finally, we can prove that the *provesKnowledgeOf* abstract messages are suitably refined:

- Message 3 (specifically the first component) refines $provesKnowledgeOfNR(b, kab, na, nb, kas, id = s)$ for a, by Refinement Rule 36: note that $associatedWith_{kas}(a)(s)$ is invariant for a, as required by condition $7'$.

- Similarly, message 4 (specifically the first component) refines $provesKnowledgeOfNR(a, kab, kbs, id = s)$ for b, by Refinement Rule 36: note that $associatedWith_{kbs}(b)(s)$ is invariant for b.

- Further, message 4 (specifically the second component) refines $provesKnowledgeOf(nb, kab, id = a)$ for b, by Refinement Rule 34.

- Finally, message 2 refines $provesKnowledgeOfNR(a, na, nb, kbs, id = b)$ for s, by Refinement Rule 36: note that $associatedWith_{kbs}(s)(b)$ is invariant for s.

9 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 three protocols.

An essential ingredient in proving many of our message refinement rules was Theorem 19, 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 one time that it is not possible to use different variable names is when long-term values, typically long-term keys, are shared between protocols and have to satisfy a *uniquelyBound* condition; however, such values normally have very similar requirements in different protocols. 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].

9.1 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 might help us to identify additional useful abstract messages and proof rules. However, we believe that we have most of the abstract messages and rules that we need: we did not need any additional abstract messages for the second and third case studies we undertook (the Otway Rees and Yahalom Protocols) over those we needed for the first (the Needham Schroeder Public Key Protocol); we needed very few additional rules for the latter case studies, and those we did need were extensions of existing rules (for example, with Invariant Rule 43 extending Rule 42).

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 using Lemma 14.

- 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. This was particularly true in our development of the Yahalom protocol, where several parts of the invariant only became evident during the message refinement stage.

The main case studies we have looked at in the current paper have been existing protocols: we have attempted a rational reconstruction of them. For brevity, we have presented the invariants in one go, rather like a rabbit out of a magician's hat. When using the calculus to develop a new protocol, we would suggest that a two-stage approach would be more appropriate, similar to the approach in Section 2:

- In the first stage, aim to achieve the authentication requirements, typically via a nonce challenge; this will often necessitate noting that certain values should be kept secret, but will not necessitate deciding how they will be kept secret.

- In the second stage, decide how to achieve the secrecy requirements, by deciding what keys to use for encryption.

It would be interesting to use the calculus to study the relationship between different protocols: we conjecture that several different protocols could correspond to the same annotation at a high level of abstraction, corresponding to the first stage, above.

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:

$I \mathrel{\widehat{=}} \mathit{honest}(b) \land \mathit{defined}(x) \Rightarrow \mathit{knows}(x) \subseteq \{a, b\}$
$\mathsf{new}\ x$
$\mathsf{send}\ \mathit{maintains}\ I \land \mathit{contains}\ x$
$\mathsf{receive}\ \mathit{maintains}\ I \land \mathit{provesKnowledgeOfNR}(x, \mathit{id} = b)$
$\left\{\mathit{session}(b;\ x)\right\}$

We have seen this pattern in each of our case studies.

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). Most of the proofs are not difficult, but involve checking lots of details: a tool could help keep track of the proof obligations, and discharge many of them automatically.

Most of our invariant and refinement rules assumed that the protocol satisfies disjoint encryption. It is interesting to ask whether we can do away with this assumption. Say that two variables x and y are *directly-confusable* if there are two encrypted components in the protocol that have the same type but have, respectively, x and y in a particular position. Say that two variables are *confusable* if they are related by the transitive closure of the above relation. We conjecture that Theorem 19 can be extended to say that, under similar circumstances, the value of a variable x can become bound to another variable y only if x and y are confusable. The proof rules that build on Theorem 19 could be similarly extended.

Our language of security protocols is currently slightly limited. It would be useful to extend the model with *functions*, such as the functions that return an agent's public or secret key, or the key shared between two agents. It would also be useful to allow *tests* performed by agents on data that they receive, where the agent aborts if the test fails. Further, our model does not include *timestamps*.

Our $\mathit{session}(b;\ \ldots)$ predicates say nothing about how far b has progressed in the session; it would be useful to be able to capture this fact. Doing so would allow us to obtain stronger results about protocols in some cases. For example, in the analysis of the Yahalom Protocol from Section 8.3, a had a postcondition of the form

$$honest(s) \Rightarrow session(s;\ a, b, na, nb, \ldots).$$

Further, s had a postcondition of the form

$$honest(b) \Rightarrow session(b;\ a, na, nb, \ldots).$$

Can we strengthen a's postcondition with

$$honest(b, s) \Rightarrow session(b;\ a, na, nb)\ ?$$

Using the present rules, the answer is no: the postcondition we proved for a does not show that s actually completed his run; and the postcondition we proved for s only refers to the case where he has completed the run. However, looking at the protocol, we can see that a is assured that s progressed to at least message 3; and s receives a guarantee about b's session as soon as s receives message 2; hence the above postulated postcondition is indeed true. It would be useful to be able to formalise this argument, via a suitable strengthing of Lemma 14.

9.2 Related Work

Datta et al. [DDMP03, DDMP05] investigate the derivation of protocols from smaller, well-used ideas, such as Diffie-Hellman key exchange, and authentication using a signed nonce challenge. They use 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: their authentication requirements are typically expressed in terms of *matching conversations* [DvOW92]. They place emphasis on composition and refinement of protocols, whereas we have chosen to concentrate on development of complete protocols.

These ideas are extended in [DDMP04]. The authors consider protocols containing *function variables*, which are intended to represent some cryptographic operation. They then prove properties of the protocol under assumptions about the function variables. Finally they instantiate the function variables with actual cryptographic operations, prove that the cryptographic operations satisfy the assumptions about the function variables, and hence deduce properties about the resulting protocol. There is a clear analogy between their function variables and our abstract messages: however, we have chosen to consider a small number of particular abstract messages, and to prove annotation and refinement rules about them, whereas they consider arbitrary function variables.

The logic is adapted in a different direction in [DDM+05, DDMW06], namely to deal with computational soundness. The logic is given a probabilistic polynomial-time semantics, and is proved sound with respect to this semantics.

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.

[DDM+05] Anupam Datta, Ante Derek, John C. Mitchell, Vitaly Shmatikov, and Mathieu Turuani. Probabilistic polynomial-time semantics for a protocol security logic. In *Proceedings of 32nd International Colloquium on Automata, Languages and Programming*, pages 16–29, 2005.

[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 IEEE Computer Security Foundations Workshop*, 2003.

[DDMP04] Anupam Datta, Ante Derek, John C. Mitchell, and Dusko Pavlovic. Abstraction and refinement in protocol derivation. In *Proceedings of 17th IEEE Computer Security Foundations Workshop*, pages 30–45, 2004.

[DDMP05] Anupam Datta, Ante Derek, John C. Mitchell, and Dusko Pavlovic. A derivation system and compositional logic for security protocols. *Journal of Computer Security (Special Issue of Selected Papers from CSFW-16)*, 13:423–482, 2005.

[DDMW06] Anupam Datta, Ante Derek, John C. Mitchell, and Bogdan Warinschi. Computationally sound compositional logic for key exchange protocols. In *Proceedings of 19th IEEE Computer Security Foundations Workshop*, 2006.

[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.

[DvOW92] Whitfield Diffie, Paul C. van Oorschot, and Michael J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2:107–125, 1992.

[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 IEEE 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.

[Low97] Gavin Lowe. A hierarchy of authentication specifications. In *Proceedings of 10th IEEE Computer Security Foundations Workshop*, 1997.

[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.

[OR87a] D. Otway and O. Rees. Efficient and timely mutual authentication. *Operating Systems Review*, 21(1):8–10, January 1987.

[OR87b] 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**
$\vdash$	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
$\trianglelefteq$	Submessage relation; $M \trianglelefteq M'$ 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
$P(\sigma)[i]$	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
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
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
knows	Annotation macro; $knows(x)$ gives the set of identities of agents who 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.	6.5
proves-Knowledge-OfNR	Abstract message; $provesKnowledgeOfNR(x)$ shows that some agent other than the local agent knows x.	6.5
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