Using Refinement to Analyse the Safety of an Authentication Protocol

Michael Butler,
Department of Electronics and Computer Science,
University of Southampton,
Highfield, Southampton SO17 1BJ, United Kingdom,
M.J.Butler@ecs.soton.ac.uk.

27 October 1998

Abstract

This paper describes an approach to the analysis of security protocols using Abrial's B method. B is a general purpose formal method based on standard set theory and predicate logic. The refinement rule we use means that we only check for safety properties such as authentication rather than liveness properties such as absence of denial of service. The contribution of this paper is the development of a style of modelling and reasoning with B that allows for a straightforward and thorough analysis of security protocols. This analysis contributes to the understanding of a protocol and could lead to an improvement in the design of security protocols.

1 Introduction

Attacks on security protocols can be very subtle and difficult to predict and guard against. For these reasons, formal analysis of security protocols is necessary and justified and provides a more rigorous analysis than testing.

Various formal approaches to the analysis of security protocols have been developed, the most well-known being the various knowledge and belief logics such as BAN [2]. These are specialised logics that are used to reason about what agents know and believe about each other. But these logics are not perfect and sometimes fail to identify protocol flaws. Other approaches involve the use of existing general purpose formal methods such as Hoare's CSP [18, 19] and the theorem-proving system Isabelle/HOL [16].

This paper describes another example of the use of a general purpose formal method to reason about security protocols. In this case, we use the B method developed by Abrial [1]. The B-method is supported by an industrial-strength tool and enjoys a growing industrial use particularly in the defence and transportation sectors [5, 7, 9].

B is a state-based formal method which means that a specification involves an explicit abstract model of the state of a system, along with operations on the abstract state. Set theory and predicate logic are used to specify the state and a command language is used to describe the operations. When describing a protocol using this approach, the state of a protocol may include a history of all messages ever exchanged between agents.

Refinement is a design method whereby the components of the abstract state of a system are replaced by more concrete data structures which are more easily implemented. A formal correspondence between the abstract specification and its refinement must be preserved. This is established by discharging certain proof obligations.

Although we use the standard B notation, we also add an extra ingredient to the method in that we formally define the possible event traces of a system specified in B. This allows us to formulate

$$
\begin{array}{|l|}
\hline
\text{MACHINE } M \\
\text{SETS } SS \\
\text{CONSTANTS } C \\
\text{VARIABLES } v \\
\text{INITIALISATION } init \\
\text{OPERATIONS} \\
\quad y_1 \longleftarrow op_1(x_1) \;\hat{=}\; S_1 \\
\quad y_2 \longleftarrow op_2(x_2) \;\hat{=}\; S_2 \\
\quad \ldots \\
\text{END} \\
\hline
\end{array}
$$

Figure 1: Abstract machine outline.

security properties in terms of event traces. Event-trace properties model safety properties of a system and are preserved by refinement. This extra ingredient is based on previous work by the author [3, 4] which helps to bridge the gap between state-based formal methods, such as B, with event-based methods such as CSP.

In this paper, we apply the approach to the Needham-Schroeder Public-Key Authentication Protocol [13]. The aim of this protocol is to provide authenticated exchange of secrets (nonces). The protocol uses public-key cryptography. Lowe [10] has shown that the original protocol is vulnerable to attack and proposes a fix to prevent this attack. We use the fixed version of the protocol in order to achieve authentication.

In Section 3, we present an abstract specification of a simple authenticated nonce-exchange service. In Section 4, we formalise the notion of event trace and show an authentication property of the service may be formulated and checked. In Section 6 we refine the abstract service in such a way that the communication and authentication of messages is achieved using the fixed version of the Needham-Schroeder Authentication Protocol. In Section 7, we also introduce into the refinement an explicit model of how an intruder could fake and replay messages to attempt to violate the authentication. We shall see that the intruder exhibits some unexpected behaviour which requires a slight modification of the original specification.

We will only be concerned with ensuring that the refined system preserves the safety properties of the specification and not with liveness properties. By showing that refinement of the (modified) abstract service is preserved, we show that the protocol does indeed achieve the desired authentication. This is the sense in which the protocol is analysed here. Precisely what is meant by authentication will be made clear in the abstract specification. The assumptions we make about the power of the intruder will be made explicit in the refinement.

We proceed with an overview of the B notation.

2 Overview of B

B AMN is a model-oriented formal notation and is part of the B method developed by Abrial [1]. A system in B is specified as an abstract machine (M) which has the form outlined in Figure 1. An abstract machine consists of some sets (SS), constants (C) and variables (v) which are modelled using standard set-theoretic constructs. Operations act on the variables and have input parameters (x_i) and output parameters (y_i). The initialisation ($init$) and operations (S_i) are written in the generalised substitution notation of B AMN.

In the B AMN notation, changes to the state variables are specified using assignment statements ($v := E$). A guarded operation is written as a statement of the form:

SELECT G THEN S END,

where the guard G is some condition on the state variables and S is some statement. Such an operation is *enabled* only when G holds.

A nondeterministic choice operation is written as:

$$\text{ANY } x \text{ WHERE } P \text{ THEN } S \text{ END}.$$

This nondeterministically chooses some value x satisfying P and then behaves as S. Such a statement is enabled only when there is a value for x satisfying P. In a refinement of this statement, the choice of x may be made more specific. A special case of the ANY construct is the generalised assignment which selects a value from a set U:

$$x :\in U \quad \text{is short for} \quad \text{ANY } y \text{ WHERE } y \in U \text{ THEN } x := y \text{ END}.$$

An operation has the form

$$y \longleftarrow op(x) \ \hat{=} \ S,$$

where x represents the input parameter(s) and y represents the output parameter(s).

In this paper, we take the view that an abstract machine proceeds by firstly initialising the state variables and then, repeatedly, an enabled operation is selected and executed. The choice of which enabled operation to execute is made externally by the environment of the system. On the other hand, the choice of outcome in a nondeterministic operation is made internally by the system. An execution of an operation op with input value e resulting in output value f models occurrence of the event $op(e)(f)$.

3 Abstract specification – authenticated nonce sharing

In this section, we introduce the abstract service which allows agents to share nonces. We will not be concerned with why or when an agent might want to share a nonce as this sort of detail will be application specific. We will simply specify a facility that allows sharing to happen. The specification is presented in a literate style, with formal text interspersed with informal text.

The service that allows agents to exchange nonces is specified as a system called *NonceExchange*:

$$\text{MACHINE} \quad \mathit{NonceExchange}.$$

Agents will be allowed to exchange nonces, so we require the following types:

$$\text{SETS} \quad \mathit{Agent};\ \mathit{Nonce}.$$

The state of the abstract service consists of a set of nonces for each pair of agents. This is modelled by a variable that is a function mapping pairs of agents to sets of nonces:

$$\text{VARIABLES} \quad \mathit{share} : (\mathit{Agent} \times \mathit{Agent}) \rightarrow \mathbb{P}\,\mathit{Nonce}.$$

For agents a and b, the set $\mathit{share}(a, b)$ models the set of nonces that agent a has said it is willing to share with agent b.

Each shared set is initialised to be empty:

$$\text{INITIALISATION} \quad \mathit{share} := (\lambda\, a, b \mid a, b \in \mathit{Agent} \cdot \{\}\ ).$$

The system has two operations, one for sharing a nonce and one for reading a shared nonce. The *ShareNonce* operation is specified as:

$$\mathit{ShareNonce}(a, b, n) \ \hat{=} \ \mathit{share}(a, b) := \mathit{share}(a, b) \cup \{n\}.$$

Occurrence of the event $\mathit{ShareNonce}(a, b, n)$ means that agent a is indicating its willingness to share nonce n with agent b thus n is added to the set $\mathit{share}(a, b)$.

An agent b has access to nonces shared with it through the *ReadNonce* operations:

$$n \longleftarrow \mathit{ReadNonce}(b, a) \ \hat{=} \ n :\in \mathit{share}(a, b).$$

$$
\begin{array}{rcl}
[skip]P & \hat{=} & P \\
[\text{SELECT } G \text{ THEN } S \text{ END}]P & \hat{=} & G \Rightarrow [S]P \\
[\text{ANY } x \text{ WHERE } Q \text{ THEN } S \text{ END}]P & \hat{=} & \forall x \cdot (Q \Rightarrow [S]P) \\
[S;\ T]P & \hat{=} & [S]([T]P)
\end{array}
$$

Figure 2: AMN semantics.

Occurrence of the event $ReadNonce(b, a)(n)$ indicates to agent b that agent a has said it is willing to share n with agent b. The value for n is chosen nondeterministically from the set $share(a, b)$. If $share(a, b)$ is empty, the event cannot occur. In a refinement of this specification, the choice of a value for n could be made more specific, e.g., on a FIFO basis. But, for the purposes of our analysis, we are only concerned with the fact that it is possible for event $ReadNonce(b, a)(n)$ to occur if $n \in share(a, b)$.

Nonces are authenticated in the following sense:

> Agent b can read a nonce n indicated to be from agent a only if a has previously agreed to share n with b.

This claim is formalised in the next section. This property will be preserved by refinement.

4 Trace properties of systems

The semantics of generalised substitutions are defined by weakest-precondition formulae: for operation S and postcondition P, the formula $[S]P$ (weakest precondition of S with respect to P[1]) characterises those initial states from which S is guaranteed to terminate in a state satisfying P. The semantics of assignment are simply substitution: $[x := E]P$ stands for P with all free occurrences of x replaced by E. The semantics of several other AMN constructs are specified in Figure 2. Note that this is only a subset of the full language.

The condition under which an operation S is enabled is called its *guard*, written $gd(S)$. This is defined as follows [12][2]:

$$gd(S) \quad \hat{=} \quad \neg\, [S]\, false.$$

From this we get the following rules for calculating the guards of guarded statements, unbounded choice statements and assignment statements:

$$
\begin{array}{rcl}
gd(\text{ SELECT } G \text{ THEN } S \text{ END }) & = & G \wedge gd(S) \\
gd(\text{ ANY } x \text{ WHERE } P \text{ THEN } S \text{ END }) & = & \exists x \cdot (P \wedge gd(S)) \\
gd(\ x :\in U\ ) & = & U \neq \{\} \\
gd(\ x := E\ ) & = & true \\
gd(\ skip\ ) & = & true.
\end{array}
$$

Execution of an operation a with input value e resulting in output value f models occurrence of the event $a(e)(f)$. A trace is a finite sequence of such events. We write $\alpha(M)$ for the set of operation names in machine M. Assume that system M has operations $M.a$ for each $a \in \alpha M$. We promote $M.a$ to traces of events, writing $M.t$ for the sequential composition of operations corresponding to event trace t. For trace t, $M.t$ is defined as follows:

$$
\begin{array}{rcl}
M.\langle\rangle & \hat{=} & skip \\
M.(\langle a \rangle \frown t) & \hat{=} & M.a\ ;\ M.t\ .
\end{array}
$$

[1] $[S]P$ is equivalent to Dijkstra's $wp(S, P)$ [6].

[2] $[S]$ *false* represents those initial states in which S could establish any postcondition, i.e., behave miraculously. An operation is said to be enabled when it cannot behave miraculously, i.e., when $\neg\, [S]$ *false* holds.

Now t is a trace of M if $M.t$ is enabled after the initialisation:

Definition 1 (System Traces) $t \in traces(M) \quad \Leftrightarrow \quad gd(M.init\ ;\ M.t) = true.$

The event traces of a system model its safety behaviour since they describe those traces of events that a system *may* engage in (as opposed to *must* engage in). Hoare uses event traces as the semantic model for CSP processes in the earlier part of his book [8]. He then elaborates this model to the so-called 'failures-divergence' model to deal with liveness behaviour. This more elaborate model may also be applied to state-based systems [12] though we don't need it here since we are only concerned with the safety of the protocol.

In CSP, trace *properties* are used to provide abstract specifications of processes. A trace property is a condition on a variable tr, and a process satisfies a specification if all of its traces meet the condition. For example, consider the specification

$$S(tr) \quad \hat{=} \quad tr \text{ prefix } \langle a, b, a, b, a, b, \ldots \rangle.$$

This is satisfied by any process that continually alternates between an a-event then a b-event. It is also satisfied by a process that performs only a finite number of alternations between a-events and b-events and then stops.

Such trace specification can also be applied to state-based systems. The following rule may be used to verify that a system M satisfies a trace specification S. The rule relies on an invariant I that relates the state variables of M with its event traces.

Rule 1 (Trace Property) *System M satisfies trace property S provided there is an invariant I such that each of the following conditions hold:*

1. $I \ \Rightarrow\ S$
2. $[M.init\ ;\ \ tr := \langle\rangle]\ I$
3. *for each operation a of M,*

$$I \quad \Rightarrow \quad [M.a\ ;\ \ tr := tr \frown \langle a(x)(y) \rangle\ ]\ I,$$

where x and y represent the input and output parameters of $M.a$.

Using Rule 1, it can be shown that the specification of the authenticated nonce-exchange service given in Section 3 satisfies the following property which says that for any agents a, b, if b reads that a nonce n came from agent a, then agent a must have previously shared n with b:

$$1 \le i \le \#tr \ \wedge\ tr(i) = ReadNonce(b, a)(n) \quad \Rightarrow \quad ShareNonce(a, b, n) \in tr[1..i-1].$$

The proof uses the following invariant:

$$\forall\, a, b, n \cdot share(a, b) \ = \ \{\ n \mid ShareNonce(a, b, n) \in tr\ \}.$$

To conclude this section, we note that specification satisfaction is conjunctive, that is, if M satisfies $Spec_1$ and M satisfies $Spec_2$, then M satisfies $Spec_1 \wedge Spec_2$. This allows us to use a modular style of reasoning to show that M satisfies some conjunction of predicates: we apply Rule 1 for each predicate $Spec_i$, possibly using a different invariant for each $Spec_i$.

5 Refinement of systems

An abstract machine is refined by applying the standard technique of data refinement to its state: an abstraction invariant is used to relate the state variables of the abstract system to those of the refined system and data refinement should hold between correspondingly-named operations in the abstract and refined systems [3].

If S is a statement that acts on variables a, T is a statement that acts on variables c, and AI is an abstraction invariant then we write

$$S \sqsubseteq_{AI} T$$

for 'S is data-refined by T under abstraction invariant AI'. Data refinement is defined as follows [1]:

Rule 2 (Data Refinement) *$S \sqsubseteq_{AI} T$ if for each postcondition q independent of concrete variable c:*

$$AI \;\wedge\; [S]q \quad\Rightarrow\quad [T](\exists\, a \cdot AI \wedge q).$$

Rule 3 (System Refinement I) *A system M is refined by system N under abstraction invariant AI if N has an operation $N.a$ corresponding to each operation $M.a$ and for each such a:*

$$M.a \sqsubseteq_{AI} N.a.$$

Furthermore, the initialisation of M must be data refined by the initialisation of N.

As well as having an operation corresponding to each of the abstract operations, a refined system may also have some auxiliary 'internal' operations. These are operations that can change the state of the refined machine but whose occurrence is not directly observable to the environment of the system. Viewed in terms of the abstract state, such operations should have no effect, that is, they should be data-refinements of the *skip* statement on the abstract state.

Rule 4 (System Refinement II) *Each internal operation $N.h$ of refined machine N must satisfy:*

$$\mathit{skip} \sqsubseteq_{AI} N.h.$$

The definition of the traces of N is extended slightly such that each visible operation is interspersed with arbitrary finite iterations of the internal operations of N, represented here as H^*:

$$\begin{array}{rcl} N.\langle\rangle & \hat{=} & H^* \\ N.(\langle a \rangle \frown t) & \hat{=} & H^* \;;\; N.a \;;\; N.t \end{array}$$

It can be shown [3] that if M is refined by N, then

$$\mathit{traces}(M) \supseteq \mathit{traces}(N).$$

That is, refinement preserves the safety behaviour of a specification.

It is easy to show that Rule 2 allows the guard of the refined operation to be stronger than its corresponding abstract operation. This is consistent with analysis of the safety of a system: a refined system that can do less is safer than the abstract system. Further conditions, as described in [3], would be required to ensure that a refinement also preserves liveness behaviour. These place constraints on the enabledness of the refined actions and also require the internal operations not to cause livelock (i.e., infinite internal computation).

6 Introducing the protocol

In this section, we introduce a system called *NeedhamSchroeder* which is intended to be a refinement of the *NonceExchange* system and uses the Needham-Schroeder protocol to exchange pairs of nonces.

MACHINE *NeedhamSchroeder*

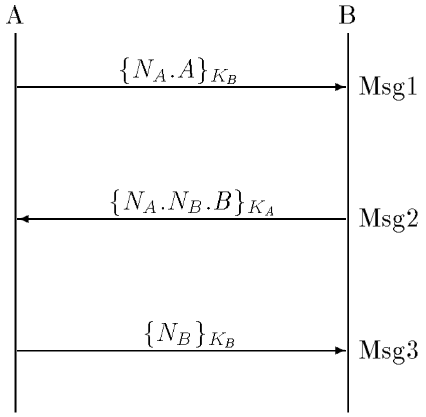

Figure 3: The Fixed Needham-Schroeder Protocol

Firstly, we introduce the following variables to the refined system:

$$\text{VARIABLES} \quad \mathit{real\_share}, \mathit{believe} : (\mathit{Agent} \times \mathit{Agent}) \to \mathbb{P}\,\mathit{Nonce}.$$

The set $\mathit{real\_share}(a,b)$ contains those nonces that a has really said it is willing to share with b through a $\mathit{ShareNonce}(a,b,n)$ event, so the corresponding operation is defined as follows:

$$\mathit{ShareNonce}(a,b,n) \quad \hat{=} \quad \mathit{real\_share}(a,b) := \mathit{real\_share}(a,b) \cup \{n\}.$$

The set $\mathit{believe}(b,a)$ contains those nonces that b believes a is willing to share with it and the refined $\mathit{ReadNonce}$ operation selects nonces from that set:

$$n \longleftarrow \mathit{ReadNonce}(b,a) \quad \hat{=} \quad n :\in \mathit{believe}(b,a).$$

The abstraction invariant used for the refinement will say that

$$\mathit{believe}(b,a) \subseteq \mathit{share}(a,b).$$

This is sufficient to ensure that the refined *ReadNonce* is indeed a correct data refinement of the abstract one. However, later we will strengthen this invariant to ensure that it is preserved by the other operations of *NeedhamSchroeder*. The variables *real_share* and *believe* are initialised so that they return the empty set for each pair of agents.

A run of the Needham-Schroeder Protocol will be used to exchange nonces between pairs of agents, copying nonces from $\mathit{real\_share}(a,b)$ to $\mathit{believe}(a,b)$. The protocol is described diagrammatically in Figure 3[3]. In this diagram, $\{N_A.A\}_{K_B}$ represents $N_A.A$ encrypted with the public key of agent B, $N_A.A$ is a composite message consisting of two parts and N_A is a nonce chosen by agent A. Only B can decrypt a message $\{m\}_{K_B}$ since only it knows its private key. The principle of the protocol is that when agent B receives the nonce N_A from A, it challenges A by responding with a message encrypted for A that includes N_A, as well as another nonce N_B. When A receives the response from B, it can be assured that N_B really came from B rather than an intruder since only B could have decrypted the first message. Similarly when B receives the third message from A it can be assured that the original N_A really came from A since only A could have decrypted the second message and know the nonce N_B. The protocol depends on the nonces being *fresh*, that is, they should not have been used previously in the network.

We introduce types representing public keys and the three forms of message that are exchanged in the protocol:

[3] We only consider the core of the protocol here and ignore messages that are used to distribute public keys.

SETS

$$
\begin{aligned}
& PubKey;\\
& Msg ::= \quad m1(Nonce \times Agent \times PubKey)\\
& \qquad\qquad \mid m2(Nonce \times Nonce \times Agent \times PubKey)\\
& \qquad\qquad \mid m3(Nonce \times PubKey).
\end{aligned}
$$

Here, Msg is a disjoint union where each of the disjoint types is distinguished by the tags $m1$, $m2$, and $m3$. Note that this model makes strong typing assumptions about encrypted messages which means, for example, that

$$m1(n, a, k) = m1(n', a', k') \;\; \Rightarrow \;\; n = n' \;\wedge\; a = a' \;\wedge\; k = k'.$$

For message m, $nonces(m)$ is the set of nonces contained in m:

$$
\begin{aligned}
nonces(m1(na, \_, \_)) &= \{na\}\\
nonces(m2(na, nb, \_, \_)) &= \{na, nb\}\\
nonces(m3(nb, \_)) &= \{nb\}.
\end{aligned}
$$

The underscore in this definition signifies an arbitrary value.

For message m, $key(m)$ is the key used to encrypt the message:

$$key(m1(\_, \_, k)) = k \qquad key(m2(\_, \_, \_, k)) = k \qquad key(m3(\_, k)) = k.$$

The public key of each agent is given by a constant function K:

CONSTANTS $\quad K : Agent \rightarrowtail PubKey.$

We assume that different agents cannot have the same public key, so K is an injective function.

For each message in a protocol run, an internal operation is introduced to model the sending of that message. We record all messages ever sent in the network using a history variable with each message being tagged by its purported sender and receiver. Later we will introduce an operation modelling the ability of the intruder to send fake messages and these will also be recorded in the history variable. In order to prove the correctness of the refinement, it will be useful to be able to distinguish between real and fake messages so each message is also tagged as being real or fake in the history variable (though, of course, this information will not be available to agents):

SETS $\quad Status \;::=\; fake \mid real$

VARIABLES $\quad history : seq(Agent \times Agent \times Msg \times Status).$

We write '$m \;\; in \;\; history$' to say that message m has been sent from some agent to some agent at some point in $history$:

$$m \;\; in \;\; history \;\; \hat{=} \;\; \exists\, a, b, s \cdot (a, b, m, s) \in history.$$

The set of all nonces appearing in $history$ is defined by:

$$nonces(history) \;\; \hat{=} \;\; \bigcup \{\; nonces(m) \;\mid\; m \;\; in \;\; history \;\}.$$

Sending of the first message in a protocol run between a and b is modelled by the internal operation $Msg1$. The nonce na is selected from the set of nonces that a is willing to share with b ($real\_share(a, b)$). The nonce must also be fresh ($na \notin nonces(history)$). The sending of the message is modelled by extending the history variable appropriately:

$$
\begin{aligned}
& \text{INTERNAL } Msg1(a, b) \;\; \hat{=}\\
& \quad \text{ANY } na \text{ WHERE}\\
& \qquad na \in real\_share(a, b) \;\wedge\; na \notin nonces(history)\\
& \quad \text{THEN}\\
& \qquad history := history \frown \langle\, (a,\; b,\; m1(na, a, K_b),\; real) \,\rangle\\
& \quad \text{END}
\end{aligned}
$$

Note that the message from a to b is recorded as being *real*.

Provided a $msg1$ has previously been sent from a to b, then b may send a $msg2$ to a. The message contains two nonces: na from the $msg1$ and nb which is selected from the set of nonces which b is willing to share with a and which must be fresh:

$$
\begin{array}{l}
\text{INTERNAL } Msg2(a,b) \ \hat{=} \\
\quad \text{ANY } na, nb \text{ WHERE} \\
\quad\quad (a,\ b,\ m1(na,a,K_b),\ \_) \in history \ \wedge \\
\quad\quad nb \in real\_share(b,a) \ \wedge \ nb \notin nonces(history) \\
\quad \text{THEN} \\
\quad\quad history := history \frown \langle\ (b,\ a,\ m2(na,nb,b,K_a),\ real)\ \rangle \\
\quad \text{END}
\end{array}
$$

Notice that the status (real or fake) of the $msg1$ in *history* is irrelevant in the guard. This models the fact that a receiving side cannot tell whether a message is real or fake.

The a side will accept a nonce nb as really coming from b if a $msg2$ has been sent from b to a and, prior to that, a has sent a corresponding $msg1$ to b. To model this situation, we make use of the *subsequence* ordering on traces: s is a subsequence of t, written '$s \operatorname{subseq} t$', if s is some projection of t which contains fewer elements than t but where the elements appear in the same order as in t. For example

$$\langle a,b\rangle \text{ subseq } t \quad \Leftrightarrow \quad \exists\, t_1,t_2,t_3 \cdot (t_1 \frown \langle a\rangle \frown t_2 \frown \langle b\rangle \frown t_3) = t.$$

The operation $BelieveNB(a,b)$ models the a side believing that the nonce nb really came from b once the first two messages have been sent:

$$
\begin{array}{l}
\text{INTERNAL } BelieveNB(a,b) \ \hat{=} \\
\quad \text{ANY } na, nb \text{ WHERE} \\
\quad\quad \langle\ (a,\ b,\ m1(na,a,K_b),\ real), \\
\quad\quad\quad (b,\ a,\ m2(na,nb,b,K_a),\ \_)\ \rangle \text{ subseq } history \\
\quad \text{THEN} \\
\quad\quad believe(a,b) := believe(a,b) \cup \{nb\} \\
\quad \text{END}
\end{array}
$$

Notice that in the guard, the status of the $msg1$ must be *real*. This is because the a side will know whether it has really sent a $msg1$ originally. The status of the $msg2$ is irrelevant because the a side (the receiving side) cannot tell whether the $msg2$ is real or fake.

The a side will send a $msg3$ under the same conditions as the $BelieveNB(a,b)$ operations using nb as the nonce:

$$
\begin{array}{l}
\text{INTERNAL } Msg3(a,b) \ \hat{=} \\
\quad \text{ANY } na, nb \text{ WHERE} \\
\quad\quad \langle\ (a,\ b,\ m1(na,a,K_b),\ real), \\
\quad\quad\quad (b,\ a,\ m2(na,nb,b,K_a),\ \_)\ \rangle \text{ subseq } history \\
\quad \text{THEN} \\
\quad\quad history := history \frown \langle\ (a,\ b,\ m3(nb,K_b),\ real)\ \rangle \\
\quad \text{END}
\end{array}
$$

Once all three messages of a protocol run have been sent and all the nonces match up, the b side believes that the original nonce na really did come from a:

$$
\begin{array}{l}
\text{INTERNAL } BelieveNA(a,b) \ \hat{=} \\
\quad \text{ANY } na, nb \text{ WHERE} \\
\quad\quad \langle\ (a,\ b,\ m1(na,a,K_b),\ \_), \\
\quad\quad\quad (b,\ a,\ m2(na,nb,b,K_a),\ real), \\
\quad\quad\quad (a,\ b,\ m3(nb,K_b),\ \_) \qquad\qquad \rangle \text{ subseq } history \\
\quad \text{THEN} \\
\quad\quad believe(b,a) := believe(b,a) \cup \{na\} \\
\quad \text{END}
\end{array}
$$

The only message sent by the b side was the $msg2$ so that is the only message that it can know as real. It cannot tell whether the other two messages are fake or real.

Note that our model places no further constraints on the sending of messages other than they be preceded by certain other messages (and no such constraint in the case of a $msg1$). This allows for interleaved protocol runs between the same agents. If we wanted to avoid having interleaved protocol runs between agents, we could further strengthen the enabling condition of the operations. For example, we could strengthen the guard of the $Msg1$ operation to the effect that it is only enabled if there are no outstanding $msg2$'s awaited from b. These constraints could be introduced in a further refinement step if required.

Furthermore, since we are only checking safety refinement and not more general safety and liveness refinement, we do not assume that an operation that remains enabled eventually gets executed rather we simply assume that operations are executed only if they are enabled. This means that we are not assuming that all messages eventually get transmitted.

If we were to leave the refined machine, *NeedhamSchroeder*, as it is, it would be easy to show that it is a refinement of the abstract machine *NonceExchange*. This could be done using Rules 3 and 4 with the abstraction invariant

$$share = real\_share \ \wedge \ \forall\, a, b \cdot believe(b, a) \subseteq share(a, b).$$

Freshness of nonces is not required for this refinement to work. However, we will introduce a further operation to *NeedhamSchroeder* in the next section to model the behaviour of the intruder. There, freshness of nonces will be required for the refinement to work.

7 The intruder

We assume that one of the agents is an intruder, $\mathcal{I}$, who can generate a message destined for an agent b that purports to be from an agent a.

CONSTANTS $\mathcal{I} : Agent$.

We assume that the intruder will not necessarily choose fresh nonces when faking messages. We wish to restrict the selection of nonces to reflect the inability of the intruder to decode a nonce that was not encrypted with the intruder's public key. We assume that the intruder cannot 'guess' a nonce that has been transmitted by an honest agent encrypted with another honest agent's key. This assumption will be modelled by restricting the choice of nonce that the intruder can make so that it can never select such a nonce. Or equivalently, we assume the probability of the intruder being able to guess such a nonce is so small as to be negligible. We also assume that the intruder can replay any previously transmitted message (between any pair of agents) without having to decrypt it.

The intruder can deduce a nonce N from a message m appearing in the history variable that is encrypted with its public key, so we define those nonces that are visible to the intruder as follows:

$$visible(history) \ \hat{=} \ \bigcup\{\ nonces(m) \ \mid \ m \ \ in \ \ history \wedge key(m) = K_{\mathcal{I}}\ \}.$$

Those nonces that are really exchanged between honest agents in the history are defined by

$$honest(history) \ \hat{=} \ \bigcup\{\ nonces(m) \ \mid \ \exists\, a, b \cdot a, b \neq \mathcal{I} \wedge (a, b, m, real) \in history\ \}.$$

The set of secret nonces are those that are really exchanged between honest agents except those that are visible to the intruder:

$$secret(history) \ \hat{=} \ honest(history) \setminus visible(history).$$

Now, when sending a fake message, the intruder can send a message containing a secret nonce ($nonces(m) \cap secret(history) \ \neq \ \{\}$) only if that message was previously sent and the intruder is replaying it:

INTERNAL *Intrude* $\hat{=}$
  ANY a, b, m WHERE
    $nonces(m) \cap secret(history) \neq \{\} \quad \Rightarrow \quad m \;\; in \;\; history$
  THEN
    $history := history \frown \langle\ (a,\ b,\ m,\ fake)\ \rangle$
  END

Note that the message is recorded as being fake in the history variable. The *Intrude* operation models all possible ways in which the intruder could fake a protocol message. This completes the description of the *NeedhamSchroeder* machine.

Modifying the abstract specification

The way in which we have specified the *Intrude* operation allows for protocol runs that result in a nonce being shared from another agent c to the intruder without agent c having agreed to share it. For example, the following sequence of messages may result in the nonce nc being added to the set $real\_share(c, \mathcal{I})$ without the agent c ever having agreed to share nc with the intruder.

$(\mathcal{I},\ c,\ m1(ni, \mathcal{I}, K_c),\ \_)$
$(c,\ \mathcal{I},\ m2(ni, nc, c, K_{\mathcal{I}}),\ fake)$

For most uses of the protocol, this is not a problem since the only agent the intruder is fooling is itself (unless the protocol was being used for non-repudiation as well). In terms of our model, it means that a $ReadNonce(\mathcal{I}, c)(nc)$ event could happen without the corresponding *ShareNonce* event having happened. Clearly this is not allowed by the abstract machine *NonceExchange*.

Similarly, the definition of the *Intrude* operation allows sharing of a nonce from the intruder to another agent, i.e., it allows a $ReadNonce(c, \mathcal{I})(n)$ event to happen, without a corresponding *ShareNonce* having happened. Again, this is not allowed by the abstract machine *NonceExchange*. However, this is not a serious violation of authentication since the intruder is faking a nonce from itself only.

These observations mean that, with the addition of the *Intrude* operation, the *NeedhamSchroeder* machine is not a refinement of the *NonceExchange* machine. We do not want to remove the *Intrude* operation since we are trying to analyse the effectiveness of the protocol in the presence of the intruder. Instead we modify the abstract specification to reflect the behaviour just identified. The abstract specification, *NonceExchange*, is modified by including the following operation in it:

Intrude $\hat{=}$
  ANY c, n WHERE
    $c \in Agent \wedge n \in Nonce$
  THEN
      $share(\mathcal{I}, c) := share(\mathcal{I}, c) \cup \{n\}$
    $[]$
      $share(c, \mathcal{I}) := share(c, \mathcal{I}) \cup \{n\}$
    $[]$
      skip
  END

This says that the most the intruder can do is fake the sharing of a nonce from itself to any agent c or fake the sharing of a nonce to itself from any agent c. The *skip* case corresponds to faking attempts that fail to complete because the right conditions are not met. Most importantly, this operation does not allow the intruder to fake the sharing of a nonce from agent a to agent b, where $a \neq \mathcal{I}$ and $b \neq \mathcal{I}$, and the modified abstract machine can be shown to satisfy the same trace property as used in Section 4 with the intruder excluded:

$$
\begin{aligned}
& a, b \neq \mathcal{I} \ \wedge \ 1 \leq i \leq \#tr \ \wedge \ tr(i) = ReadNonce(b, a)(n) \\
& \quad \Rightarrow \ ShareNonce(a, b, n) \in tr[1..i-1].
\end{aligned}
$$

The modification to the abstract specification was arrived at through a process of 'reverse engineering'. The refinement proof was attempted using the original abstract specification but it was found that some of the refinement obligations involving agents only worked for honest agent. This lead to the identification of the above cases involving the intruder and to the need to modify the original abstract specification.

8 Verifying the refinement

The correctness of the refinement is based on using an abstraction invariant to relate the abstract variable *share* of the modified *NonceExchange* machine with the variables *real_share*, *believe* and *history* of *NeedhamSchroeder*. The abstraction invariant is presented as a set of conjunctive clauses. It is assumed that the free variables of each clause are universally quantified over and that index i is suitably constrained, i.e., $1 \leq i \leq \#history$. Some of the important clauses are discussed below. Others are listed in Figure 4.

The set of nonces that b believes came from a is a subset of those really shared by a in the abstract machine:

$$believe(b, a) \subseteq share(a, b). \tag{1}$$

For honest agents a, b, the abstract set of nonces $share(a, b)$ will be the same as $real\_share(a, b)$

$$a, b \neq \mathcal{I} \ \Rightarrow \ share(a, b) = real\_share(a, b). \tag{2}$$

The abstract set of nonces shared by the intruder with another agent c will be the set $real\_share(\mathcal{I}, c)$ as well as those nonces chosen by the intruder in fake messages from or to the intruder as follows:

$$
\begin{aligned}
share(\mathcal{I}, c) \ = \ & real\_share(\mathcal{I}, c) \cup \\
& \{ \, n \mid \ (\mathcal{I}, \ c, \ m1(n, \mathcal{I}, K_c), \ fake) \in history \ \vee \\
& \qquad (\mathcal{I}, \ c, \ m2(\_, n, \mathcal{I}, K_c), \ fake) \in history \, \}
\end{aligned} \tag{3}
$$

Similarly, the abstract set of nonces (purportedly) shared by an agent c with the intruder will be the set $real\_share(c, \mathcal{I})$ as well as those nonces chosen by the intruder in fake messages from or to the intruder as follows:

$$
\begin{aligned}
share(c, \mathcal{I}) \ = \ & real\_share(c, \mathcal{I}) \cup \\
& \{ \, n \mid \ (c, \ \mathcal{I}, \ m1(n, c, K_{\mathcal{I}}), \ fake) \in history \ \vee \\
& \qquad (c, \ \mathcal{I}, \ m2(\_, n, c, K_{\mathcal{I}}), \ fake) \in history \, \}
\end{aligned} \tag{4}
$$

One of the most important clauses says that once a nonce is secret, it remains secret:

$$n \in secret\_nonces(history[1..i]) \ \Rightarrow \ n \in secret\_nonces(history[1..i+1]). \tag{5}$$

According to Rules 3 and 4, we must use the abstraction invariant to show that the following refinements hold:

i). *NonceExchange.ShareNonce* is data-refined by *NeedhamSchroeder.ShareNonce*

ii). *NonceExchange.ReadNonce* is data-refined by *NeedhamSchroeder.ReadNonce*

iii). *NonceExchange.Intrude* is data-refined by *NeedhamSchroeder.Intrude*

iv). Each of the internal operations *Msg*1, *Msg*2, *Msg*3, *BelieveNA*, *BelieveNB* in *NeedhamSchroeder* are data-refinements of *skip*.

$$history(i) = (a, b, m1(na, a', k), real) \quad \Rightarrow \quad a = a' \wedge k = K_b. \tag{6}$$

$$\begin{aligned} &history(i) = (a, b, m1(na, a, k), real) \quad \Rightarrow \\ &\qquad na \in share(a, b) \ \wedge \ na \notin nonces(history[1..(i-1)]). \end{aligned} \tag{7}$$

$$history(i) = (b, a, m2(na, nb, b', k), real) \quad \Rightarrow \quad b = b' \wedge k = K_a. \tag{8}$$

$$\begin{aligned} &history(i) = (b, a, m2(na, nb, b, k), real) \quad \Rightarrow \\ &\qquad nb \in share(b, a) \ \wedge \ nb \notin nonces(history[1..(i-1)]). \end{aligned} \tag{9}$$

$$history(i) = (a, b, m3(nb, k), real) \quad \Rightarrow \quad k = K_b. \tag{10}$$

$$\begin{aligned} &history(i) = (a, b, m3(nb, k), real) \quad \Rightarrow \\ &\quad \exists\, i1, i2, s2, na \ \mid \ i1 < i2 < i3 \cdot \\ &\qquad history(i1) = (a, b, m1(na, a, K_b), real) \ \wedge \\ &\qquad history(i2) = (b, a, m2(na, nb, b, K_a), s2). \end{aligned} \tag{11}$$

$$\begin{aligned} &history(i) = (a, b, m, fake) \ \wedge \ nonces(m) \cap secret(history[1..(i-1)]) \ \neq \ \{\} \\ &\quad \Rightarrow \quad m \ \ in \ \ history[1..(i-1)]. \end{aligned} \tag{12}$$

Figure 4: Further clauses of the abstraction invariant.

We do not discuss all the details of the refinement proof, only the more interesting and significant parts.

Conditions i), ii) and iii) are straightforward to prove. Consider condition i), where the abstract and concrete *ShareNonce* operations add a nonce n to $share(a, b)$ and $real\_share(a, b)$ respectively. For the abstract operation to be data-refined by the concrete, it must be the case that, assuming the abstraction invariant holds between *share* and *real_share* before execution of the respective operations, then it must hold afterwards as well. This boils down to showing that

$$\begin{aligned} &share(a, b) = real\_share(a, b) \cup S \quad \Rightarrow \\ &\quad share(a, b) \cup \{n\} = real\_share(a, b) \cup S \cup \{n\}, \end{aligned}$$

which is clearly true. (Here, $S = \{\}$ when $a, b \neq \mathcal{I}$ otherwise S corresponds to the set expressions of (3) and (4).)

In the case of condition ii), the abstract *ReadNonce* operation selects a nonce to output from the set $share(a, b)$, while the concrete operation selects a nonce from the set $believe(b, a)$. The data-refinement condition boils down to showing that

$$y \in believe(b, a) \quad \Rightarrow \quad y \in share(a, b),$$

which follows immediately from (1).

In the case of condition iii), the abstract *Intrude* operation adds nonces, (a), to $share(\mathcal{I}, c)$ or, (b), to $share(c, \mathcal{I})$ or, (c), does nothing (i.e., behaves as *skip*), while the concrete operation issues fake messages. For data refinement to hold between these operations, it must be the case that the issuing of a fake message corresponds to one of the possible abstract behaviours (a), (b) or (c). Clause (3) deals with case (a), that is, sending a fake $m1$ or $m2$ from $\mathcal{I}$ to c corresponds to adding a nonce to $share(\mathcal{I}, c)$ since such messages may result in nonces being added to $believe(c, \mathcal{I})$. Similarly, clause (4) deals with case (b). Sending a fake message between honest agents corresponds to *skip* since such fake messages will not result in nonces being incorrectly added to $believe(a, b)$.

Condition iv) involves showing that each of the internal operations is a refinement of *skip*. It can be shown that $skip \sqsubseteq_{AI} S$ is equivalent to showing that

$$AI \ \Rightarrow \ [S]AI,$$

which means that operation S must preserve invariant AI. The *BelieveNA* operation adds a nonce na to $believe(b, a)$. In order to preserve the invariant clause

$$believe(b, a) \subseteq share(a, b),$$

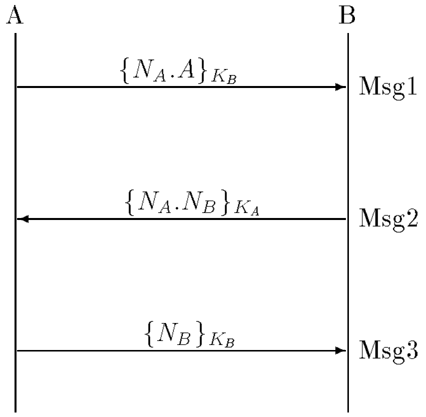

Figure 5: The Original Needham-Schroeder Protocol

it must be the case that na is already in $share(a, b)$. However, $na \in share(a, b)$ is not an explicit guard of the $BelieveNA$ operation. Thus we need to show that the abstraction invariant and the guard of $BelieveNA$ are sufficient to ensure that $na \in share(a, b)$, i.e.,

$$
\begin{array}{l}
Invariant \ \wedge \\
\langle \ (a, \ b, \ m1(na, a, K_b), \ \_), \\
\quad (b, \ a, \ m2(na, nb, b, K_a), \ real), \\
\quad (a, \ b, \ m3(nb, K_b), \ \_) \qquad\qquad \rangle \ \text{subseq} \ history \\
\qquad \Rightarrow \quad na \in share(a, b).
\end{array} \tag{13}
$$

Similarly when $BelieveNB$ adds nonce nb to $believe(a, b)$ it must be the case that $nb \in share(b, a)$, i.e.,

$$
\begin{array}{l}
Invariant \ \wedge \\
\langle \ (a, \ b, \ m1(na, a, K_b), \ real), \\
\quad (b, \ a, \ m2(na, nb, b, K_a), \ \_) \ \rangle \ \text{subseq} \ history \\
\qquad \Rightarrow \quad nb \in share(b, a).
\end{array} \tag{14}
$$

Both (13) and (14) follow from the invariant using case analysis (whether an agent is honest or not, whether a message is real or fake). In particular, they require clause (5) of the invariant ("preservation of secrecy") in order to be proven.

Since the internal operations $Msg1$, $Msg2$ and $Msg3$ add real messages to $history$, these operations have the potential to violate clause (5). The $Msg1$ operation adds nonce na to $history$. However this is always fresh and hence could not have been secret previously so clause (5) is trivially preserved.

The $Msg2$ operation adds a message with two nonces, na and nb, to $history$. Nonce nb is fresh but na will have existed previously in $history$. If the transmitted message is encrypted for the intruder, i.e., $key(m) = \mathcal{I}$, then na will be revealed to the intruder. To ensure secrecy is preserved, it should be the case that na was not secret from the intruder whenever the operation $Msg2(\mathcal{I}, b)$ adds $m2(na, nb, b, K_{\mathcal{I}})$ to $history$. This is indeed the case since $Msg2(\mathcal{I}, b)$ is only enabled when $m1(na, \mathcal{I}, K_b)$ is in $history$ and for $m1(na, \mathcal{I}, K_b)$ to be in history, na could not be secret from the intruder.

A similar situation arises in the $Msg3$ operation which adds a message with a non-fresh nonce nb to $history$.

8.1 Failure of the original Needham-Schroeder protocol

Consider the original version of the Needham-Schroeder protocol. This is illustrated in Figure 5. The difference between the two protocols is that, in the fixed version, the identity of agent B is included in the encrypted $msg2$, whereas, in the original protocol, only the nonces are included in the $msg2$.

This protocol does not preserve the secrecy of nonces (Clause 5 of the abstraction invariant) because an honest agent may unwittingly reveal a secret nonce to the intruder. To see this, consider the following sequence of message exchanges involving two interleaved protocol runs identified by $A1 \ldots A3$ and $B1 \ldots B3$ respectively:

$$\begin{array}{ll} A1. & (a,\ \mathcal{I},\ m1(na, a, K_{\mathcal{I}}),\ \mathit{real}) \\ B1. & (a,\ b,\ m1(na, a, K_b),\ \mathit{fake}) \\ B2. & (b,\ a,\ m2(na, nb, K_a),\ \mathit{real}) \\ A2. & (\mathcal{I},\ a,\ m2(na, nb, K_a),\ \mathit{fake}) \\ A3. & (a,\ \mathcal{I},\ m3(nb, K_{\mathcal{I}}),\ \mathit{real}) \\ B3. & (a,\ b,\ m3(nb, K_b),\ \mathit{fake}) \end{array}$$

Here, at step $A1$ an agent a initiates a run with the intruder. The intruder then uses the nonce na at step $B1$ to initiate a fake run of the protocol with agent b pretending to be agent a. In step $A2$ the intruder replays $m2(na, nb, K_a)$ of step $B2$ in its reply to agent a. It can do this because, unlike in the fixed version of the protocol, the $msg2$ does not contain the identity of b. Up to and including step $A2$ the nonce nb is secret since the intruder cannot decrypt $m2(na, nb, K_a)$. However, at step $A3$ the agent a replies to the intruder with $m3(nb, K_{\mathcal{I}})$ thus unwittingly revealing nb to the intruder and violating the preservation-of-secrecy clause. Subsequently at step $B3$, the intruder can fool agent b into believing that na was really shared by a with b. The above sequence is the attack on the original protocol identified by Lowe [10].

It is interesting to note that, while the abstract authenticated nonce-sharing specification is not violated until after step $B3$, the abstraction invariant used for the refinement is already violated at step $A3$.

9 Conclusions

Coming up with the appropriate abstraction invariant is perhaps the most challenging part of the analysis. For the Needham-Schroeder protocol, the least obvious clause of the invariant was the preservation-of-secrecy clause:

- once a nonce is used secretly, it remains secret.

The other clauses can easily be derived from the guards of the operations. The preservation-of-secrecy clause arose through trial and error. In hindsight it is an obvious requirement of an authentication protocol and therefore should appear in the invariant, but it was not immediately obvious at the start of the analysis. It is also a subtle property in the sense that, if the protocol is flawed, the property can be violated at an earlier stage than authentication. In Section 8, this was shown to be the case with Lowe's attack on the original Needham-Schroeder protocol.

It is often the case with interesting refinements that parts of the abstraction invariant are not obvious to begin with. Sometimes derivation of the full abstraction invariant is an iterative process with a first obvious attempt being tried. Then, when the proof obligations are being discharged, further clauses will be discovered to be necessary. If these are not derivable from the existing clauses, they should be added to the invariant and the process retried.

The *NeedhamSchroeder* machine is an example of a fault tolerant system since it provides authentication even in the presence of an intruder. We modelled the intruder's behaviour by introducing the *Intrude* operation, a highly nondeterministic operation modelling all possible ways in which, we assume, the intruder may fake a message. We are not obliged to implement the intruder's behaviour, rather it represents our assumption about how the intruder may affect the system.

It may appear counter to the standard view of top-down development that we modified the abstract specification after developing a refinement of it. This arose because our assumptions about how the intruder may behave were at the concrete level of messages rather than at the abstract level of the initial service description. It was only by relating abstract and concrete machines that we could understand how the intruder behaviour manifests itself at the abstract level. Furthermore, we were not in a position to modify the intruder behaviour in order to avoid modifying the abstract specification.

Our Needham-Schroeder development was partly analysed using a commercial tool that supports the B method [14]. All of the specifications were syntax and type-checked. The tool also has theorem-proving support. All the proof obligations for the refinement are automatically generated. However, only about 10% of these were discharged automatically. This is partly because the tool has very few in-built theorems on traces. Most of the proofs for the example covered by this paper were performed on paper.

An important feature of the approach is that refinement preserves event-trace properties, that is, any event-trace property satisfied by a specification machine will also be satisfied by its refinement. It is known, however, that some security properties cannot be modelled in terms of event-trace properties and are not preserved by refinement, e.g., absence of covert channels. Refinement would allow the designer to resolve any nondeterminism such that some covert means of signalling is introduced. One way of avoiding this is to require that the specification be deterministic [17].

Our approach, including use of the preservation-of-secrecy clause in the invariant, was tried on a different protocol, the Otway-Rees key distribution protocol, and was found to be successful. The preservation-of-secrecy clause was again the only non-obvious one and seems to be an important property of authentication protocols.

Comparing our approach with the use of BAN logic, we find that the use of B and refinement results in a more verbose specification and proof than is possible with BAN. This is partly because B is verbose by nature with its use of keywords, and partly because we need to be explicit about everything since we are starting with a very simple modelling framework (i.e., state machines). The refined machine includes an explicit model of the faking of messages by the intruder and of the way in which an honest agent responds to fake messages. The history variable is more knowledge than an individual agent (including the intruder) could ever have, but it is available to us as designers and was essential when constructing the appropriate invariant for the refinement. The model of the system in the refinement provides a rich basis for reasoning about the protocol.

Our model of the protocol resembles that used in Paulson's inductive approach [16] where message transmission, including the faking of messages, is modelled by extending a history trace. Paulson's approach involves proving various properties of all possible traces of a network. His approach is bottom-up in that he first models a protocol and from this model develops various security properties which are then verified. Much of the proof effort seems to be quite similar to ours in that some of the properties that are verified in Paulson's approach correspond to the sorts of clauses that need to be added to the abstraction invariant in our approach. This suggests that the powerful theorem-proving technology used by Paulson, Isabelle/HOL [15], could be used in verifying refinements in our approach.

Because we are concerned with preservation of event-trace properties during refinement, our approach is also related to the the use of CSP to analyse security protocols [18, 19]. Schneider uses a proof-theoretic approach to check that CSP processes modelling security protocols satisfy trace properties. These trace properties are similar to the trace property for authentication used in Section 4. Roscoe uses a powerful model-checking tool, FDR, to check that CSP models of protocols satisfy security properties. Since FDR checks refinement between CSP processes, the security properties are themselves encoded as CSP processes. FDR verifies refinements automatically and generates counter-examples in the case of incorrect refinements. This provides a powerful way of discovering flaws in protocols. This was how Lowe discovered a flaw in the original Needham-Schroeder protocol [11].

While model-checking provides insight into why protocols fail when it does discover a flaw, it provides little insight when it verifies that a refinement is correct. A proof-theoretic analysis can provide more insight into why a protocol does work. For example, the discovery of the preservation-of-secrecy clause of the abstraction invariant used in verifying the refinement in this paper provides more insight into why the (fixed) Needham-Schroeder protocol works. This insight should be transferable to other protocols. Model-checking does complement a proof-theoretic approach as it allows developers to quickly experiment with alternative designs.

B is a general purpose formal method and has been used in the development of large systems. This means that it should be possible to integrate our approach to the analysis of security properties with the overall development of a system in which security is only one aspect of functionality.

Acknowledgements

This work was supported in part by the UK Defence Evaluation and Research Agency. Thanks to Jean-Raymond Abrial, Stefan Hallerstede, Colin O'Halloran, Louis Mussat, Ulrich Nitsche and Peter Ryan for their help.

References

[1] J.R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

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

[3] M.J. Butler. Stepwise refinement of communicating systems. *Science of Computer Programming*, 27(2):139–173, September 1996.

[4] M.J. Butler. An approach to the design of distributed systems with B AMN. In J.P. Bowen and M.G. Hinchey, editors, *10th International Conference of Z Users (ZUM'97)*, volume LNCS 1212, pages 223–241. Springer–Verlag, 1997.

[5] B. Dehbonei and F. Mejia. Formal development of safety-critical software systems in railway signalling. In M.G. Hinchey and J P. Bowen, editors, *Applications of Formal Methods*, pages 227–252. Prentice Hall International, 1995.

[6] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[7] J. Draper. The use of the B-Method on an avionics example - the MIST project. In Henri Habrias, editor, *Proceedingsof 1st Conference on the B method*, pages 303–305, 1996.

[8] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice–Hall, 1985.

[9] J.P. Hoare. Application of the B-method to CICS. In M.G. Hinchey and J.P. Bowen, editors, *Applications of Formal Methods*, pages 97–124. Prentice Hall International, 1995.

[10] G. Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–133, 1995.

[11] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. *Software Concepts and Tools*, 17:93–102, 1996.

[12] C.C. Morgan. Of wp and CSP. In W.H.J. Feijen, A.J.M. van Gasteren, D. Gries, and J. Misra, editors, *Beauty is our business: a birthday salute to Edsger W. Dijkstra.* Springer–Verlag, 1990.

[13] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(2):120–126, 1978.

[14] D.S. Nielson and I.H. Sorensen. The B-technologies: a system for computer aided programming. In U.H. Engberg, K.G. Larsen, and P.D. Mosses, editors, *6th Nordic Workshop on Programming Theory.* BRICS, October 1994.

[15] L.C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS.* Springer–Verlag, 1994.

[16] L.C. Paulson. The inductive approach to verifying cryptographic protocols. To appear in *J. Computer Security*, 1998.

[17] A.W. Roscoe. CSP and determinism in security modelling. In *Proceedings of 1995 IEEE Symposium on Security and Privacy.* IEEE Computer Society Press, 1995.

[18] A.W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *Proceedings of 8th IEEE Computer Security Foundations Workshop.* IEEE Computer Society Press, 1995.

[19] S. Schneider. Verifying Authentication Protocols with CSP. In *Proceedings of 10th IEEE Computer Security Foundations Workshop*, 1997.