On the Use of Data Refinement in the Development of Secure Communications Systems

Michael Butler
mjb@ecs.soton.ac.uk

Declarative Systems and Software Engineering Group
Technical Report DSSE-TR-2001-1
January 2001

www.dsse.ecs.soton.ac.uk/techreports/

Department of Electronics and Computer Science
University of Southampton
Highfield, Southampton SO17 1BJ, United Kingdom

On the Use of Data Refinement in the Development of Secure Communications Systems

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

January 30, 2001

Abstract

We report on experiences gained from the application of data refinement techniques to the development of examples of secure communications systems. The aim was to the carry the development from initial abstract specification of security services through to detailed designs. The development approach was based on action systems, with B and CSP being used as concrete notations. The security services in question are a confidential communications service and an authenticated transaction service. Refinements include explicit representations of intruder behaviour.

The paper makes several interrelated contributions. It demonstrates the feasibility of applying a refinement approach to this type of problem, including an effective way of combining B and CSP in refinements. It introduces a more systematic approach to the development of abstraction invariants and refinement checking. Finally, it illustrates the limitation, when modelling security protocols, of a formalism that does not deal with probability.

1 Introduction

In this paper, we report on experiences gained from the application of data refinement techniques to the development of some examples of secure communications systems. These are systems that provide secure communications services on a distributed network in which one or more intruders may attempt to subvert the required security by monitoring and faking messages on the network.

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 [6]. These are specialised logics that are used to reason about what agents know and believe about each other. Other approaches involve the use of existing general purpose formal methods such as Hoare's CSP [27, 28] and the theorem-proving system Isabelle/HOL [25]. These existing approaches have concentrated on abstract analysis of security protocols, that is, checking that protocols satisfy certain properties, without carrying the development through to a formally verified detailed design or implementation.

In the work described here, we have used the action system formalism [3], a formal method that supports the development of distributed systems from initial abstract specification of a service to a detailed design. Action systems are an approach to modelling systems which combines a state-based and an event-based view. Action systems can be refined so that more implementation-oriented detail is added and can be decomposed into parallel subsystems. As a concrete notation for describing action systems, we have used a combination of the *abstract machine notation* (AMN) of Abrial's B Method [2] and Hoare's CSP [15]. Tool support is available for AMN and the author has developed a tool that allows AMN and CSP to be combined [10]. The notation is outlined in Section 2. Our approach to refinement is outlined in Section 3. The comprehensive treatment of refinement using a combination of CSP and B given in this paper is in itself novel.

We describe the application of the action system formalism to two case studies, an authenticated transaction service and a confidential communication service. For the most part, the application of the formal method was successful. We were able to specify the services abstractly, then refine these using existing security protocols, taking the intruder behaviour into account, and then perform further refinement to arrive at a detailed architectural design. The initial refinement step which introduces the security protocol and introduces the intruder is no less powerful than the existing approaches to abstract analysis of protocols.

An essential ingredient for checking the correctness of a data refinement step is an abstraction invariant which relates the states of the abstract and concrete systems. Traditionally, one "invents" an invariant and checks all the appropriate proof obligations. Through checking the proof obligations, we may discover some assumptions that are not implied by the invariant; in this case, we strengthen the invariant with the new assumptions and re-check all the proof obligations. Several iterations may be required and all of the proof obligations need to be re-checked each time which can be very tedious. As part of our effort, we discovered a way of iteratively deriving and checking the abstraction invariant without having to re-check all the proof obligations each time the invariant is strengthened. This method does not work in all cases, but it did prove sufficient for our security case studies. The method, which supports a more systematic way of deriving and checking an abstraction invariant, is described in Section 3, and its use is described in later sections.

A usual approach with action systems is to start with a system having a global state that all actions of the system can access directly. This global state is then refined to a state representing the required distributed structure so that the system can be decomposed into parallel subsystems based on a partitioning of the state. However, we encountered a limitation of our approach that prevents us from fully decomposing the system into parallel subsystems. This limitation arises because of the way we treat probability. Let us illustrate this with an example: suppose the set $\mathcal{S}$ represents the set of secret nonces currently in use by all agents. We make the assumption that the probability of the intruder guessing one of these nonces is almost zero so that, when faking a nonce, the probability that a nonce selected by the intruder is in the set $\mathcal{S}$ is close to zero. The formalism we are using does not treat probability so we deal with situations such as these by excluding their possibility with a constraint such as

$$nonce \notin \mathcal{S}.$$

This constraint is necessary to prove the validity of the refinement step which introduces the intruder behaviour. When the system is further refined, whether a nonce is in use by an agent or not, will be part of the local state of that agent so that $\mathcal{S}$ will depend on the local state of all the agents. This means that the above constraint on the behaviour of the intruder cannot be partitioned from the local states of the agents and that a full partitioning of the subsystems cannot be performed within the formalism we are using. We have not attempted to address this limitation in this paper. Note that this is a limitation of all the existing approaches to abstract analysis of security protocols as these do not use formalisms that treat probability either. This limitation does not invalidate the use of existing analysis approaches as they can still be used to discover flaws in security protocols.

This paper does not present any new theoretical framework but instead it describes the use of an existing general purpose formal method in the development of distributed implementations of secure communications services. We believe it makes several useful contributions:

- It illustrates, with examples, a feasible approach to the application of the method to the development of secure communications systems. This includes an explicit model of possible intruder behaviour and deals with development down to a distributed architecture.
- It presents an effective way of combining the notations of CSP and B in a single framework that treats refinement.
- It introduces a more systematic approach to the compositional derivation and checking of abstraction invariants for refinements.
- It illustrates the limitation, when modelling security protocols, of a formalism that does not deal with probability.

For simplicity, we only consider refinement which preserves safety properties in this paper. The intruder can easily disrupt protocol runs which means liveness cannot be guaranteed unless we were to include a complex mechanism that overcomes this using timeouts etc.

2 Formal Notation

An action system consists of the some state variables, representing the state space of the system, an initialisation for the system and a set of actions on the variables of the system. Actions may be guarded by a condition on the state and actions are said to be enabled when their guard is satisfied. Actions may have parameters. An action system proceeds by executing its initialisation and then, continually, an enabled action is selected and executed.

2.1 B AMN

In this paper, the AMN notation of Abrial's B Method is used to specify action systems. In B, a machine is the basic unit of specification and, as with an action system, a B machine consists of some state variables and some operations (actions) on the state variables.

In the AMN operation notation, changes to the state variables are specified using assignment statements ($v := E$). A simultaneous assignment of the form $x := E \parallel y := F$ is equivalent to $x, y := E, F$. A guarded action 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 action is enabled only when G holds. The nondeterministic choice of two statements is written $S \mathbin{[]} T$. Such a statement is enabled when either S or T are enabled. A nondeterministic value selection action is written as:

ANY x WHERE P THEN S 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.

We write $gd(S)$ for the enabling condition of a statement S. The enabling condition of a compound statement may be calculated using the rules shown in Figure 1.

Actions systems are state-based since they use an explicit model of the state of a system. But an event-based view can also be taken of an action system as follows: An action definition has the form

$a(x) \mathrel{\hat=} S$,

where a is the action name, x represents the parameter(s), and S is an AMN statement. An execution of an action a with parameter value e models occurrence of the event $a(e)$.

$$
\begin{aligned}
gd(\text{ SELECT } G \text{ THEN } S \text{ END }) &= G \wedge gd(S) \\
gd(\text{ ANY } x \text{ WHERE } P \text{ THEN } S \text{ END }) &= \exists\, x \cdot (P \wedge gd(S)) \\
gd(\ S \mathbin{[\!]} T\ ) &= gd(S) \vee gd(T) \\
gd(\ x := E\ ) &= \mathit{true} \\
gd(\ \mathit{skip}\ ) &= \mathit{true}.
\end{aligned}
$$

Figure 1: Calculating statement guards.

MACHINE *ConfComms*
SETS *Agent*, *Text*
CONSTANTS $\mathcal{C} \subseteq \mathit{Agent}$
VARIABLES $\mathit{comms} \in (\mathit{Agent} \times \mathit{Agent}) \rightarrow \mathbb{P}\,\mathit{Text}$
INITIALISATION $\mathit{comms} := \lambda\, a, b \cdot \{\}$
OPERATIONS
$\mathit{Send}(a, b, t) \;\hat{=}\; \mathit{comms}(a, b) := \mathit{comms}(a, b) \cup \{t\}$
$\mathit{Receive}(a, b, t) \;\hat{=}\; \mathit{skip}$
$\mathit{Reveal}(a, b, t) \;\hat{=}$
SELECT $(a \in \mathcal{C} \vee b \in \mathcal{C}) \;\wedge\; t \in \mathit{comms}(a, b)$ THEN *skip* END
END

Figure 2: Confidential communication service.

Figure 2 presents an example of an action system modelling a confidential communications service. Occurrence of event $\mathit{Send}(a, b, t)$ models a request by agent a to communicate a text message t to agent b. The effect of this action is to add t to the set $\mathit{comms}(a, b)$, representing all the texts communicated by a to b. The parameterisation of the actions means that a $\mathit{Send}(a, b, t)$ event can occur for any pair of agents $a, b \in \mathit{Agent}$ and any $t \in \mathit{Text}$. Occurrence of event $\mathit{Reveal}(a, b, t)$ models the ability of the intruder to read the text message sent from a to b and represents a violation of confidentiality. The *Reveal* action is guarded by the condition that one of the agents a, b is compromised (i.e., in the set $\mathcal{C}$). Thus texts sent between two non-compromised agents will not be revealed to the intruder so that confidentiality is maintained for non-compromised agents. Later in Section 7, we shall see that a compromised agent is one whose secret key, that it shares with a trusted key server, is known to the intruder.

Occurrence of event $\mathit{Receive}(a, b, t)$ models agent b reading a text that it believes has been communicated to it by agent a. Note that the *Receive* action is always enabled meaning that $\mathit{Receive}(a, b, t)$ might occur without $\mathit{Send}(a, b, t)$ ever having occurred. This would represent a violation of authentication since agent b cannot be sure that t came from a. For this example, we will only be concerned with ensuring confidentiality and not authentication.

2.2 CSP

The csp2B tool [10] can be used to translate a CSP process description into a B machine. Consider the following CSP description of a process:

$$P \;=\; \mathit{Request}?t \rightarrow \mathit{Transact}.t \rightarrow \mathit{Confirm}.t \rightarrow P.$$

Process P performs a *Request* event followed by a *Transact* followed by a *Confirm* and then behaves as P again. This is intended to model a client-server based transaction system. Initially the client is allowed to request that some transaction t take place. In the parlance of CSP, t is an input parameter of the *Request* event so that the system allows the environment (in this case, the

$$
\begin{array}{rcl}
\text{PROCESS } \mathit{AuthTrans} & = & |||\ c, s \in \mathit{Agent} \bullet \mathit{InitAuthTrans}[c, s] \ \text{ WHERE} \\
\mathit{InitAuthTrans}[c, s] & = & \mathit{Request}.c.s?t \rightarrow \\
 & & \mathit{Transact}.c.s.t \rightarrow \\
 & & \mathit{Confirm}.c.s.t \rightarrow \mathit{InitAuthTrans}[c, s] \\
\text{END} & &
\end{array}
$$

Figure 3: Authenticated transaction service in CSP.

client) to choose the value t for the transaction. The server then performs the transaction with the value t chosen by the client. This is modelled by the *Transact*.t event. Finally the *Confirm*.t event occurs, confirming to the client that the transaction really has been performed. Our intention is that the value t might, for example, be a request to transfer a certain sum of money between bank accounts or a request to by some shares or goods, etc.

This description implies authentication in two ways: Firstly, the *Transact*.t event can only occur if *Request*.t has occurred immediately beforehand, i.e., the transaction can only be performed if it really was requested. In a distributed implementation, after the client makes the request, a request message is sent to the server over the network and the server side needs to be sure that this message is authentic. Secondly, the *Confirm*.t event can only occur if a *Transact*.t has occurred immediately beforehand, i.e., the confirmation that the transaction was performed can only occur if the transaction really was performed. Again, in a distributed implementation, a confirmation message is sent to the client over the network, and the client side needs to be sure that this message is authentic.

A more general version of this CSP process is described in Figure 3. Here, the process $\mathit{InitAuthTrans}[c, s]$ represents the repeated occurrence of transaction runs between a client c and a server s and the process *AuthTrans* is defined to be an interleaved set of transaction runs between client-server pairs. For generality, we assume an agent may act both as a client and as a server.

The CSP process *AuthTrans* will be translated into the B machine *AuthTransMch* of Figure 4 by the csp2B tool. Here, the ordering between events defined explicitly in the CSP description is achieved by introducing an abstract program counter, $\mathit{AuthTrans}(c, s)$, for each client-server pair and guarding the actions appropriately. The program counter for each client-server pair is initialised to 0, in which case, only the *Request* action is enabled. Execution of *Request* sets the program counter to 1, in which case only the *Transact* action is enabled, and so on. Execution of *Request* also results in the transaction parameter t being stored in $T(c, s)$ so that it can be referred to by *Transact* and *Confirm*. In general, any input parameter in a CSP process that remains in scope in further events will result in a state variable being introduced in the generated B machine.

Clearly the event-oriented behaviour of the B machine *AuthTransMch* is the same as the behaviour described by the CSP process *AuthTrans*. We do not need to prove this as it is guaranteed by the csp2B tool. More precisely, the csp2B tool guarantees that the labelled transition system defined from the CSP process is the same as the labelled transition system defined by the B machine. We choose to use CSP in this case because it provides a more transparent way of describing the order of events in transaction runs.

As with the confidential communications service, we shall assume that some of the agents may be compromised, allowing authentication associated with these agents to be violated. We found it most convenient to specify this using AMN rather than CSP by extending the actions of *AuthTransMch* so that they are also enabled if one of the agents involved is compromised. This extension is defined in Figure 5 where *AuthTransMch* is included in a machine called *CompAuthTrans*. Inclusion in B means that the actions of *AuthTransMch* may be called by actions of *CompAuthTrans* and the variables of *AuthTransMch* may be read by actions of *CompAuthTrans*. The semantics of a call to an included machine is simply the textual replacement of the call by the definition of the called action, with formal parameters being replaced by actual parameters.

MACHINE $AuthTransMch$
VARIABLES
$AuthTrans : (Agent \times Agent) \rightarrow (0..2)$
$T : (Agent \times Agent) \nrightarrow Transaction$
INITIALISATION $AuthTrans := \lambda\, c, s \cdot 0$
OPERATIONS
$Request(c, s, t) \mathrel{\hat{=}}$
SELECT $AuthTrans(c, s) = 0$
THEN $AuthTrans(c, s) := 1 \;||\; T(c, s) := t$ END
$Transact(c, s, t) \mathrel{\hat{=}}$
SELECT $AuthTrans(c, s) = 1 \;\wedge\; t = T(c, s)$
THEN $AuthTrans(c, s) := 2$ END
$Confirm(c, s, t) \mathrel{\hat{=}}$
SELECT $AuthTrans(c, s) = 2 \;\wedge\; t = T(c, s)$
THEN $AuthTrans(c, s) := 0$ END
END

Figure 4: B representation of authenticated transaction service generated by csp2B.

MACHINE $CompAuthTrans$
INCLUDES $csp.AuthTransMch$
OPERATIONS
$$\begin{array}{lcl} Request(c, s, t) & \hat{=} & csp.Request(c, s, t) \;\;[]\;\; (c \in \mathcal{C} \vee s \in \mathcal{C} \implies skip) \\ Transact(c, s, t) & \hat{=} & csp.Transact(c, s, t) \;\;[]\;\; (c \in \mathcal{C} \vee s \in \mathcal{C} \implies skip) \\ Confirm(c, s, t) & \hat{=} & csp.Confirm(c, s, t) \;\;[]\;\; (c \in \mathcal{C} \vee s \in \mathcal{C} \implies skip) \end{array}$$
END

Figure 5: Compromised authenticated transaction service.

Prefixing of the included machine with '$csp.$' simply prefixes each action name of $AuthTransMch$ with '$csp.$' to avoid name clashes[1]. Note that $G \implies S$ is short for SELECT G THEN S END.

For each action of $CompAuthTrans$, the corresponding action from $AuthTransMch$ is combined using the choice operator with an action that is enabled when either c or s is compromised. The choice of two statements is enabled when either statement is enabled. If c or s is compromised, then $Transact(c, s, t)$ can occur in $CompAuthTrans$ even if $Request(c, s, t)$ has not occurred, thus violating authentication. On the other hand, if both c and s are non-compromised, then $Transact(c, s, t)$ cannot occur in $CompAuthTrans$ without $Request(c, s, t)$ having occurred. Similarly for $Confirm(c, s, t)$.

Note that, for this example, we only need to write the CSP process $AuthTrans$ and the machine $CompAuthTrans$. The machine $AuthTransMch$ is generated by the csp2B tool.

3 Traces and Refinement

3.1 System traces

The semantics of B AMN statements are defined by weakest-precondition formulae: for action S and postcondition P, the formula $[S]P$ (weakest precondition of S with respect to P[2]) characterises

[1]In B, the variable names of the included machine would also be prefixed by '$csp.$', but we ignore that here.
[2]$[S]P$ is equivalent to Dijkstra's $wp(S, P)$ [13].

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

Figure 6: AMN statement semantics.

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 6. Note that this is only a subset of the full language.

The guard of a statement, $gd(S)$, is defined as follows [21][3]:

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

The rules of Figure 1 for calculating guards are easily justified using this definition.

If action system M has an action a of the form $a(x) \hat{=} S$, we write $M.a$ for S and $M.a(e)$ for $x := e;\; S$. Execution of the statement $M.a(e)$ models occurrence of the event $a(e)$ (or α for short). A trace is a finite sequence of such events and we write $M.t$ for the sequential composition of statements corresponding to event trace t. For trace t, $M.t$ is defined as follows:

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

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

Definition 3.1 (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 [15]. 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 [21] though we do not use it here since we will only deal with the safety of the systems under development.

The significance of system traces is that their properties are preserved by refinement so that, if the traces of an abstract system guarantee some safety-oriented security service, then so will its refinements.

3.2 Data Refinement

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 actions in the abstract and refined systems (see, e.g., [8]).

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

$$S \sqsubseteq_{AI} T$$

[3]Because of monotonicity of $[S]$, $[S]\; false$ represents those initial states in which S is guaranteed to establish any postcondition, i.e., behave miraculously. An action is said to be enabled when it cannot behave miraculously, i.e., when $\neg\, [S]\; false$ holds.

for 'S is data-refined by T under abstraction invariant AI'. Data refinement is defined as follows [2]:

Rule 3.1 (Data Refinement) *$S \sqsubseteq_{AI} T$ if the following holds[4]:*

$$AI \quad \Rightarrow \quad [T](\langle S\rangle AI).$$

Here, $\langle S\rangle Q$ is the *conjugate* of $[S]$, representing the weakest precondition under which it is *possible* for S to establish Q (as opposed to the *guarantee* provided by $[S]Q$) [21]. The conjugate weakest precondition defined as follows:

$$\langle S\rangle Q \mathrel{\hat=} \neg\, [S] \neg\, Q\ .$$

Rule 3.2 (System Refinement I) *A system M is refined by system N under abstraction invariant AI if N has an action $N.a$ corresponding to each action $M.a$ 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 action corresponding to each of the abstract actions, a refined system may also have some auxiliary 'internal' actions. These are actions 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 actions should have no effect, that is, they should be data-refinements of the *skip* statement on the abstract state.

Rule 3.3 (System Refinement II) *Each internal action $N.h$ of refined machine N must satisfy:*

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

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

$$\begin{aligned} N.\langle\rangle &\mathrel{\hat=} H^* \\ N.(\langle\alpha\rangle \frown t) &\mathrel{\hat=} H^*\ ;\ N.\alpha\ ;\ N.t \end{aligned}$$

It can be shown [7] that if M is refined by N, then

$traces(M) \supseteq traces(N).$

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

Rule 3.3 assumes that M has no internal actions. However, if M has internal actions and N has corresponding internal actions, as well as extra internal actions, then the refinement rules can still be applied by treating the corresponding internal actions as visible actions. This is valid because the internalisation of actions is monotonic w.r.t. refinement [7].

It is easy to show that Rule 3.1 allows the guard of the refined action to be stronger than its corresponding abstract action. 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 [8], 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 actions not to cause livelock (i.e., infinite internal computation).

Rule 3.2 corresponds to the standard B notion of system refinement. Rule 3.3 is not part of standard B, though recent work by Abrial has extended B refinement providing a similar rule [1].

[4] It is also required that $[S]true$, the *precondition* of S, holds, but we ignore this since we only use statements whose preconditions are *true* in this paper.

3.3 Incremental Refinement

Suppose I is the initial abstraction invariant that we choose for showing data refinement. We would check the data refinement obligation for a pair of corresponding actions S and T by simplifying $[T](\langle S\rangle I)$ and checking that it is implied by I. We might find that I is not strong enough and that the abstraction invariant needs to be strengthened to $I \wedge J$. This means we now have to re-do the proof showing that $I \wedge J \Rightarrow [T](\langle S\rangle I \wedge J)$ for each corresponding pair of operations T and S. If I is large and there are many operations, then this will be tedious. Furthermore, several iterations are typically required to get an invariant that is strong enough for all the operations so that the invariant will get larger and larger.

It would be more convenient if we only had to show that the extra clause J was established rather than all of $I \wedge J$. Unfortunately, this is not possible in general as $\langle S\rangle$ is not conjunctive[5], i.e.,

$$[T](\langle S\rangle I) \;\wedge\; [T](\langle S\rangle J) \quad \not\Rightarrow \quad [T](\langle S\rangle I \wedge J).$$

However we do have the following inference rule for data refinement:

Rule 3.4 (Incremental Refinement)

$$\frac{\begin{array}{c} I \wedge J \Rightarrow [T](\langle S\rangle I) \\ I \wedge J \Rightarrow [T]([S]J) \end{array}}{I \wedge J \Rightarrow [T](\langle S\rangle I \wedge J)}$$

The rule is valid because of monotonicity of $[T]$ and the following lemma:

Lemma 3.1

$$\langle S\rangle I \wedge [S]J \;\Rightarrow\; \langle S\rangle(I \wedge J).$$

Proof (by contradiction):

$$\begin{array}{cll} & \langle S\rangle I \;\wedge\; [S]J \;\wedge\; \neg\,\langle S\rangle(I \wedge J) & \\ = & \langle S\rangle I \;\wedge\; [S]J \;\wedge\; [S]\neg\,(I \wedge J) & \\ = & \langle S\rangle I \;\wedge\; [S](J \;\wedge\; \neg\,(I \wedge J)) & \text{“ } [S] \text{ is conjunctive ”} \\ \Rightarrow & \langle S\rangle I \;\wedge\; [S]\neg\, I & \\ = & \neg\,[S]\neg\, I \;\wedge\; [S]\neg\, I & \\ = & \mathit{false}. & \end{array}$$

Rather than having to re-check $[T](\langle S\rangle I \wedge J)$, Rule 3.4 means that we only have to check that J is 'established', i.e., $I \wedge J \Rightarrow [T]([S]J)$. If J is required to prove refinement between a pair of corresponding actions, then we also need to check that refinement under $I \wedge J$ holds between all the other corresponding pairs of actions. If we have already shown that refinement under I holds for these, then Rule 3.4 may be used to do this incrementally. This incremental refinement checking supports a more structured and systematic way of deriving a valid abstraction invariant and results in a real saving in the proof effort. This is illustrated in later sections.

When S is nondeterministic, showing that $I \wedge J \Rightarrow [T]([S]J)$ may not be possible even when $I \wedge J \Rightarrow [T](\langle S\rangle J)$ can be shown. We discuss this in Section 5.1. The rule is easier to apply to internal actions of the refined system since, in that case, the abstract command S is *skip* and $[\mathit{skip}]J = J = \langle \mathit{skip}\rangle J$.

[5]For example, consider $S \mathrel{\hat{=}} x := 0 \;[]\; x := 1$, $I \mathrel{\hat{=}} x \in \{0, 2\}$, $J \mathrel{\hat{=}} x \in \{1, 2\}$. We have that $\langle S\rangle I \wedge \langle S\rangle J \;=\; \mathit{true}$, but $\langle S\rangle(I \wedge J) \;=\; \mathit{false}$.

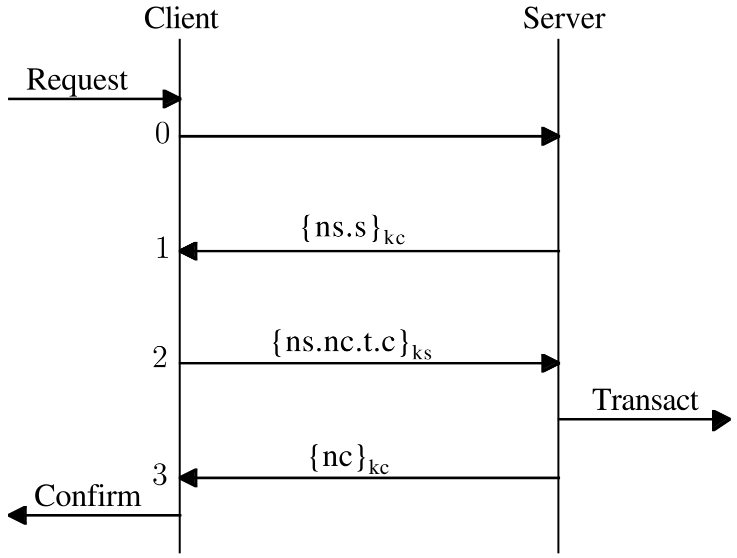

Figure 7: Authentication protocol.

4 Refinement of Authenticated Service

In this section we introduce a system called *CompAuthTrans*2 which is a refinement of the system *CompAuthTrans*. The agents are assumed to be connected via a message-passing network and the refined system contains auxiliary internal actions which correspond to the transfer of messages between clients and servers. The protocol used to achieve the service uses public-key encryption and is illustrated in Figure 7. In this diagram, $\{ns.s\}_{kc}$ represents $ns.s$ encrypted with the public key of agent c, $ns.s$ is a composite message consisting of two parts and ns is a nonce chosen by agent s. Only c can decrypt a message $\{m\}_{kc}$ since only it knows its private key. However, if an agent is compromised, its private key is know to the intruder. The protocol used to achieve the service is based on a version of the Needham-Schroeder Public-key Authentication Protocol [23]. Lowe [16] has shown that the original protocol is vulnerable to attack and proposed a fix to prevent this attack. We use the fixed version of the protocol in order to achieve authentication.

The principle of the protocol is that when a client c makes a request for a transaction t with server s, a message is sent over the network to the server. The server responds with a message containing a nonce ns. When the client receives the nonce ns, it in turn responds with a message encrypted for s that includes ns, as well as a fresh nonce nc. When the server receives message 2 from the client, it can be assured that it really came from c rather than an intruder since only c could have decrypted the message 1 and observed ns, thus it can proceed with execution of the transaction. Similarly when c receives message 3 from s it can be assured that it really came from s since only s could have decrypted message 2 and observed nc, thus it accepts the message as confirmation that the transaction took place.

We introduce types representing the four forms of message that are exchanged in the protocol:

$$
\begin{aligned}
Msg ::= \quad & m0 \\
\mid\ & m1(Nonce \times Agent \times Agent) \\
\mid\ & m2(Nonce \times Nonce \times Transaction \times Agent \times Agent) \\
\mid\ & m3(Nonce \times Agent).
\end{aligned}
$$

Here, Msg is a disjoint union where each of the disjoint types is distinguished by the tags $m0$, $m1$, $m2$, and $m3$. In $m1$, $m2$, and $m3$, the last agent component represents the agent whose public key is used to encrypt the message. For message m, $target(m)$ is the agent for whom m is

REFINEMENT $CompAuthTrans2$
REFINES $CompAuthTrans$
INCLUDES $client.ClientMch, server.ServerMch$
VARIABLES
$chan : (Agent \times Agent) \to (bag\ Msg)$
$history : \mathbb{P}\ Msg$
$NS, NC : (Agent \times Agent) \nrightarrow Nonce$

INITIALISATION
$chan := \lambda\, c, s.\{\} \quad || \quad history := \{\}$

Figure 8: Initial part of refined machine.

encrypted:

$$\begin{array}{lcl} target(m1(\_,\_,c)) & = & c \\ target(m2(\_,\_,\_,s)) & = & s \\ target(m3(\_,c)) & = & c. \end{array}$$

The underscore in this definition signifies an arbitrary value.

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

$$\begin{array}{lcl} nonces(m0) & = & \{\} \\ nonces(m1(ns,\_,\_)) & = & \{ns\} \\ nonces(m2(ns,nc,\_,\_)) & = & \{nc, ns\} \\ nonces(m3(nc,\_)) & = & \{nc\}. \end{array}$$

Note that this model of messages makes strong typing assumptions about encrypted messages which means, for example, that

$$\begin{array}{l} m1(ns,s,c) = m1(ns',s',c') \quad \Rightarrow \quad ns = ns' \ \wedge \ s = s' \ \wedge \ c = c' \\ m1(\_,\_,\_) \ \neq \ m2(\_,\_,\_,\_) \end{array}$$

4.1 Refined System

The initial part of the refined machine is shown in Figure 8. The refined machine includes two machines, *ServerMch* and *ClientMch*, generated from the CSP specifications shown in Figure 9 in the manner described previously. These included machines control the ordering of actions for the client and server respectively. Note the client process constrains the value of t in the *SendMsg*2 and *Confirm* actions. The AMN definitions of the actions will contain further constraints on their execution. Messages that are in transit in the network are modelled explicitly by a variable *chan* with $chan(c,s)$ representing the bag (multiset) of messages being exchanged between c and s in both directions (for a protocol run initiated by c). All messages ever sent within the network are recorded in the variable *history*. The most recently selected values for the nonces *ns* and *nc* for each pair of agents c, s are recorded in the variables *NC* and *NS* respectively.

4.2 Actions

The actions of *CompAuthTrans*2 are: *Request*, *Confirm*, *Transact*, *SendMsg*0, *SendMsg*1, *SendMsg*2, *SendMsg*3, and *Fake*.

The *Request* action simply involves a call to the client CSP machine which means it will be enabled precisely when it is enabled by the CSP machine:

$$Request(c,s,t) \ \hat{=} \ client.Request(c,s,t)$$

PROCESS $\mathit{Client} \;=\; ||| \; c, s \cdot \mathit{InitClient}[c, s]$ WHERE

$$
\begin{aligned}
&\mathit{InitClient}[c, s] \;= \\
&\quad \mathit{Request}.c.s?t \;\rightarrow \\
&\quad \mathit{SendMsg}0.c.s \;\rightarrow \\
&\quad \mathit{SendMsg}2.c.s.t \;\rightarrow \\
&\quad \mathit{Confirm}.c.s.t \;\rightarrow\; \mathit{InitClient}[c, s]
\end{aligned}
$$

END

PROCESS $\mathit{Server} \;=\; ||| \; c, s \cdot \mathit{InitServer}[c, s]$ WHERE

$$
\begin{aligned}
&\mathit{InitServer}[c, s] \;= \\
&\quad \mathit{SendMsg}1.c.s \;\rightarrow \\
&\quad \mathit{Transact}.c.s?t \;\rightarrow \\
&\quad \mathit{SendMsg}3.c.s \;\rightarrow\; \mathit{InitServer}[c, s]
\end{aligned}
$$

END

Figure 9: Client and server CSP processes.

When sending a message 0, $m0$ is added to the channel between c and s[6]:

$$
\begin{aligned}
&\mathit{SendMsg}0(c, s) \;\hat{=} \\
&\quad \mathit{client}.\mathit{SendMsg}0(c, s) \;\; || \\
&\quad \mathit{chan}(c, s) := \mathit{chan}(c, s) + \{m0\} \;\; || \\
&\quad \mathit{history} := \mathit{history} \cup \{m0\}
\end{aligned}
$$

Before a message 1 is sent, there should be an $m0$ in $\mathit{chan}(c, s)$:

$$
\begin{aligned}
&\mathit{SendMsg}1(c, s) \;\hat{=} \\
&\quad \text{ANY} \;\; ns \;\; \text{WHERE} \\
&\qquad m0 \in \mathit{chan}(c, s) \;\wedge\; ns \notin \mathit{nonces}(\mathit{history}) \\
&\quad \text{THEN} \\
&\qquad \mathit{server}.\mathit{SendMsg}1(c, s) \;\; || \\
&\qquad \mathit{NS}(c, s) := ns \;\; || \\
&\qquad \mathit{chan}(c, s) := \mathit{chan}(c, s) - \{m0\} + \{m1(ns, s, c)\} \;\; || \\
&\qquad \mathit{history} := \mathit{history} \cup \{m1(ns, s, c)\} \\
&\quad \text{END}
\end{aligned}
$$

Note that the value chosen for nonce ns should be fresh, i.e., should not have been used in any previous messages ($ns \notin \mathit{nonces}(\mathit{history})$). We assume an infinite supply of nonces hence the freshness constraint cannot prevent selection of a nonce.

[6] Here we use parallel assignment $||$ in a more general way. Commands with this more general form of parallel may be rewritten to the basic form using the following rules:

$$
\begin{aligned}
&(\text{SELECT } \; G \text{ THEN } \; S \text{ END}) \; || \; T \;\; = \;\; \text{SELECT } \; G \text{ THEN } \; S \; || \; T \text{ END} \\
&(\text{ANY } \; x \; \text{ WHERE } \; G \text{ THEN } \; S \text{ END}) \; || \; T \;\; = \;\; \text{ANY } \; x \; \text{ WHERE } \; G \text{ THEN } \; S \; || \; T \text{ END} \\
&(S \; [] \; T) \; || \; U \;\; = \;\; (S \; || \; U) \; [] \; (T \; || \; U)
\end{aligned}
$$

In the second equation, we assume that x is not free in T.

If a message 1 has been sent encrypted for c, then a message 2 can be sent to s:

$$
\begin{array}{l}
\mathit{SendMsg2}(c,s) \mathrel{\hat{=}} \\
\quad \text{ANY } \ ns, nc, t \ \text{ WHERE} \\
\qquad m1(ns,s,c) \in \mathit{chan}(c,s) \ \wedge \ nc \notin \mathit{nonces}(\mathit{history}) \\
\quad \text{THEN} \\
\qquad \mathit{client}.\mathit{SendMsg2}(c,s,t) \ \| \\
\qquad \mathit{NC}(c,s) := nc \ \| \\
\qquad \mathit{chan}(c,s) := \mathit{chan}(c,s) - \{m1(ns,s,c)\} + \{m2(ns,nc,t,c,s)\} \ \| \\
\qquad \mathit{history} := \mathit{history} \cup \{m2(ns,nc,t,c,s)\} \\
\quad \text{END}
\end{array}
$$

Note that ns is the same in both $m1$ and $m2$ and nc is fresh. Note also that the value of t will be determined by $\mathit{client}.\mathit{SendMsg2}(c,s,t)$.

If a message 2 has been sent whose first nonce component is the same as the nonce stored in $\mathit{NS}(c,s)$, i.e., the client responds with the correct nonce, then the transaction can take place:

$$
\begin{array}{l}
\mathit{Transact}(c,s,t) \mathrel{\hat{=}} \\
\quad m2(\mathit{NS}(c,s), \_, t, c, s) \in \mathit{chan}(c,s) \quad \Longrightarrow \quad \mathit{server}.\mathit{Transact}(c,s,t)
\end{array}
$$

The parameter t of the transaction must have the same value as that of t in the message. Note that the message is not yet removed from $\mathit{chan}(c,s)$; this is done by the next action which sends a message 3:

$$
\begin{array}{l}
\mathit{SendMsg3}(c,s) \mathrel{\hat{=}} \\
\quad \text{ANY } \ nc, t \ \text{ WHERE} \\
\qquad m2(\mathit{NS}(c,s), nc, t, c, s) \in \mathit{chan}(c,s) \\
\quad \text{THEN} \\
\qquad \mathit{server}.\mathit{SendMsg3}(c,s) \ \| \\
\qquad \mathit{chan}(c,s) := \mathit{chan}(c,s) - \{m2(\mathit{NS}(c,s), nc, t, c, s)\} + \{m3(nc,c)\} \ \| \\
\qquad \mathit{history} := \mathit{history} \cup \{m3(nc,c)\} \\
\quad \text{END}
\end{array}
$$

If a message 3 has been sent whose nonce component is the same as the nonce stored in $\mathit{NC}(c,s)$, i.e., the server responds with the correct nonce, then the transaction is confirmed:

$$
\begin{array}{l}
\mathit{Confirm}(c,s,t) \mathrel{\hat{=}} \\
\quad \text{SELECT} \\
\qquad m3(\mathit{NC}(c,s), c) \in \mathit{chan}(c,s) \\
\quad \text{THEN} \\
\qquad \mathit{client}.\mathit{Confirm}(c,s,t) \ \| \\
\qquad \mathit{chan}(c,s) := \mathit{chan}(c,s) - \{m3(\mathit{NC}(c,s), c)\} \\
\quad \text{END}
\end{array}
$$

The final action to be specified is the *Fake* action representing the masquerading behaviour of an intruder. The nonces visible to an intruder, $\mathit{VISIBLE}$, are those nonces contained in messages destined for compromised agents:

$$
\mathit{VISIBLE} \ \mathrel{\hat{=}} \ \bigcup \{ \ \mathit{nonces}(m) \ \mid \ m \in (\mathit{history} \setminus \{m0\}) \ \wedge \ \mathit{target}(m) \in \mathcal{C} \ \}.
$$

We also introduce the set $\mathit{CRITICAL}$, whose value depends on the variables NS, NC, Server, and Client:

$$
\begin{array}{lcl}
\mathit{CRITICAL} & \mathrel{\hat{=}} & \{ \ \mathit{NS}(c,s) \ \mid \ c \notin \mathcal{C} \ \wedge \ s \notin \mathcal{C} \ \wedge \ \mathit{Server}(c,s) = 1 \ \} \ \cup \\
& & \{ \ \mathit{NC}(c,s) \ \mid \ c \notin \mathcal{C} \ \wedge \ s \notin \mathcal{C} \ \wedge \ \mathit{Client}(c,s) = 3 \ \}
\end{array}
$$

Server(c, s) and *Client*(c, s) model the control states of the respective CSP machines. $Server(c, s) = 1$ when server s has sent a message 1 with the nonce $NS(c, s)$ and is waiting the receive a message 2 containing the same nonce. $Client(c, s) = 3$ when client c has sent a message 2 with the nonce $NC(c, s)$ and is waiting the receive a message 3 containing the same nonce.

We allow the intruder to send fake messages, but we assume that the probability of the intruder being able to select nonces in the set *CRITICAL* is so small that the possibility does not exist. We also allow the intruder to select visible nonces and to replay a previously sent message without having to decrypt it. Thus the constraint on the selection of message to be faked is that its nonces are non-critical or visible or that the message is a replay:

$$
\begin{array}{l}
Fake(c, s) \mathrel{\hat{=}} \\
\quad \text{ANY } \ m \ \text{ WHERE} \\
\quad\quad nonces(m) \cap CRITICAL \ = \ \{\} \\
\quad\quad \vee \ \ nonces(m) \ \subseteq \ VISIBLE \\
\quad\quad \vee \ \ m \in history \\
\quad \text{THEN} \\
\quad\quad chan(c, s) := chan(c, s) + \{m\} \ \ || \\
\quad\quad history := history \cup \{m\} \\
\quad \text{END}
\end{array}
$$

The following should be noted about this definition of *Fake*:

- It can be viewed as modelling several agents acting as intruders, possibly in unison, with the set C being the set of intruding agent identifiers.
- It allows for a used nonce to become known to an intruder through, for example, cryptanalysis after a certain time, i.e., once the nonce is no longer critical.

4.3 Verifying the Refinement

To verify the correctness of the refinement step, Rule 3.2 gives us the following refinement obligations (we write CAT for *CompAuthTrans*):

$$
\begin{array}{rcll}
CAT.Request & \sqsubseteq_I & CAT2.Request & (1) \\
CAT.Transact & \sqsubseteq_I & CAT2.Transact & (2) \\
CAT.Confirm & \sqsubseteq_I & CAT2.Confirm & (3)
\end{array}
$$

Rule 3.3 gives us the following refinement obligations on the auxiliary actions introduced in the refinement:

$$
\begin{array}{rcll}
skip & \sqsubseteq_I & CAT2.SendMsg0 & (4) \\
skip & \sqsubseteq_I & CAT2.SendMsg1 & (5) \\
skip & \sqsubseteq_I & CAT2.SendMsg2 & (6) \\
skip & \sqsubseteq_I & CAT2.SendMsg3 & (7) \\
skip & \sqsubseteq_I & CAT2.Fake & (8)
\end{array}
$$

We derive an invariant I that will satisfy each of these conditions in an incremental fashion. The invariant will link the abstract and concrete variables listed as follows:

Abstract *AuthTrans*, T

Concrete *chan*, *history*, *NS*, *NC*, *Client*, *Server*, T_C

Here, T_C is part of *ClientMch* used to store the value for t chosen by the *Request* event.

Our first version of the invariant is shown in Figure 10. Here, c and s are assumed to be universally quantified over elements of *Agent*. Clause 1.1 represents the link between T and T_C

1.1	$c \notin \mathcal{C} \;\wedge\; s \notin \mathcal{C} \;\Rightarrow\; T(c,s) = T_C(c,s)$
1.2	$\mathit{chan}(c,s) \subseteq \mathit{history}$
1.3	$\mathit{CRITICAL} \subseteq \mathit{nonces}(\mathit{history})$

Figure 10: Invariant 1

that we expect to hold, and clauses 1.2 and 1.3 represent an 'obvious' invariant on the concrete variables.

To give a flavour of the proof style, we use Rule 3.1 to prove:

$$\mathit{CAT.Request} \;\sqsubseteq_I\; \mathit{CAT2.Request}.$$

We use $1.1 \wedge 1.2 \wedge 1.3$ as the initial invariant $I1$. We assume $I1$ and proceed as follows (we write Cl for Client and AT for $\mathit{AuthTrans}$)[7]:

$$\begin{array}{cl}
 & [\mathit{CAT2.Request}]\ (\langle \mathit{CAT.Request}\rangle\ 1.1 \wedge 1.2 \wedge 1.3) \\
= & [\mathit{Cl}(c,s) = 0 \implies T_c(c,s) := t \,||\, \mathit{Cl}(c,s) := 1] \\
 & \quad (\langle AT(c,s) = 0 \implies T(c,s) := t \,||\, AT(c,s) := 1\rangle\ 1.1 \wedge 1.2 \wedge 1.3) \\
= & \qquad \text{“SELECT”} \\
 & \mathit{Cl}(c,s) = 0 \;\Rightarrow\; [T_c(c,s) := t \,||\, \mathit{Cl}(c,s) := 1] \\
 & \quad (\langle AT(c,s) = 0 \implies T(c,s) := t \,||\, AT(c,s) := 1\rangle\ 1.1 \wedge 1.2 \wedge 1.3) \\
= & \qquad \text{“conjugate of SELECT”} \\
 & \mathit{Cl}(c,s) = 0 \;\Rightarrow\; [T_c(c,s) := t \,||\, \mathit{Cl}(c,s) := 1] \\
 & \quad (AT(c,s) = 0 \wedge \langle T(c,s) := t \,||\, AT(c,s) := 1\rangle\ 1.1 \wedge 1.2 \wedge 1.3) \\
= & \qquad \text{“}\langle x := E\rangle = [x := E]\text{”} \\
 & \mathit{Cl}(c,s) = 0 \;\Rightarrow\; [T_c(c,s) := t \,||\, \mathit{Cl}(c,s) := 1] \\
 & \quad (AT(c,s) = 0 \wedge\ ([T(c,s) := t]1.1) \wedge 1.2 \wedge 1.3) \\
= & \mathit{Cl}(c,s) = 0 \;\Rightarrow \\
 & \quad AT(c,s) = 0 \wedge\ ([T_c(c,s) := t][T(c,s) := t]1.1) \wedge 1.2 \wedge 1.3 \\
= & \qquad \text{“substitution, assumption”} \\
 & \mathit{Cl}(c,s) = 0 \;\Rightarrow\; AT(c,s) = 0 \wedge\ t = t \\
= & \mathit{Cl}(c,s) = 0 \;\Rightarrow\; AT(c,s) = 0
\end{array}$$

Thus we see that the initial invariant is not strong enough prove that

$$\mathit{CAT.Request} \;\sqsubseteq_I\; \mathit{CAT2.Request}$$

and that in order to be able to prove it we need to strengthen the invariant by conjoining the clause

$$\mathit{Client}(c,s) = 0 \;\Rightarrow\; \mathit{AuthTrans}(c,s) = 0$$

When we try the same for the other refinement obligations $(2)\ldots(8)$, using $I1$ as the gluing invariant, we find that a total of two extra clauses are required. Each of these is shown in Figure 11. In this figure, the first column is a label, the second is the clause, and the third is the action (or

[7] In these proofs, we make use of the fact that

$$\begin{array}{l}
[f(c,s) := E](\forall\, c,s \cdot P(f(c,s))) \quad = \\
\quad (\forall\, c',s' \mid c' \neq c \vee s' \neq s \cdot P(f(c',s'))) \;\wedge\; [f(c,s) := E]P(f(c,s)).
\end{array}$$

$(\forall\, c',s' \mid c' \neq c \vee s' \neq s \cdot P(f(c',s')))$ will follow from the invariant which means that we only need to work with $[f(c,s) := E]P(f(c,s))$.

		Required by:
2.1	$Client(c,s) = 0 \;\Rightarrow\; AuthTrans(c,s) = 0$	*Request*
2.2	$Server(c,s) = 1 \;\wedge\; m2(NS(c,s), \_, t, c, s) \in history \;\Rightarrow$ $AuthTrans(c,s) = 1 \;\wedge\; t = T(c,s)$	*Transact*
2.3	$Client(c,s) = 3 \;\wedge\; m3(NC(c,s), c) \in history \;\Rightarrow$ $AuthTrans(c,s) = 2$	*Confirm*

where $c \notin \mathcal{C} \;\wedge\; s \notin \mathcal{C}$

Figure 11: Invariant 2

corresponding abstract/concrete pair) that give rise to the obligation. Thus, to prove refinement between the corresponding *Transact* actions, $I1$ needs to be strengthened by clause 2.2. The auxiliary actions are trivially refinements of *skip* under $I1$. Note that here and later c and s are quantified over elements of *Agent* excluding the compromised agents, $\mathcal{C}$.

Of course, the refinement proof is not yet complete since we need to show that each of the refinement conditions (1)...(8) holds under the gluing invariant $I1 \wedge I2$, where $12 \;=\; 2.1 \wedge 2.2 \wedge 2.3$. Rather than re-proving these fully using $I1 \wedge I2$, we use the incremental refinement rule, Rule 3.4.

For example, we know that:

$$I1 \wedge 2.1 \quad\Rightarrow\quad [CAT2.Request]\;(\langle CAT.Request\rangle\; I1)$$

Using Rule 3.4, we need to show that

$$I1 \wedge I2 \quad\Rightarrow\quad [CAT2.Request]\;([CAT.Request]\; J)$$

where J is instantiated with each of 2.1, 2.2, and 2.3 in turn. It is straightforward to show that no new clauses are required to show that these obligations hold for the *Request* actions.

When showing this for some of the other actions, new clauses do arise however. For example, with the *Transact* actions and clause 2.1:

$$\begin{array}{cl}
 & [CAT2.Transact]\;([CAT.Transact]\;2.1) \\
= & [AT(c,s) = 1 \wedge t = T(c,s) \;\Longrightarrow\; AT(c,s) := 2\;] \\
 & \quad ([Se(c,s) = 1 \wedge m2(NS(c,s), _, t, c, s) \in history \;\Longrightarrow\; Se(c,s) := 2]\; 2.1) \\
= & \qquad \text{“SELECT”} \\
 & AT(c,s) = 1 \wedge t = T(c,s) \;\Rightarrow \\
 & \quad (Se(c,s) = 1 \;\wedge\; m2(NS(c,s), _, t, c, s) \in history \;\Rightarrow\; [AT(c,s) := 2][Se(c,s) := 2]2.1) \\
= & \qquad \text{“Substitution”} \\
 & AT(c,s) = 1 \wedge t = T(c,s) \;\Rightarrow \\
 & \quad (Se(c,s) = 1 \;\wedge\; m2(NS(c,s), _, t, c, s) \in history \;\Rightarrow\; Cl(c,s) \neq 0) \\
\Leftarrow & \qquad \text{“Sufficient condition”} \\
 & Se(c,s) = 1 \;\wedge\; m2(NS(c,s), _, t, c, s) \in history \;\Rightarrow\; Cl(c,s) \neq 0.
\end{array}$$

This last step involves generalising the new obligation, i.e., making it stronger. This generalisation simplifies the form of the clause which in turn simplifies further proofs.

Another use of generalisation arose when checking that the *Request* actions establish Clause 2.1. Calculation gives:

$$\begin{array}{cl}
 & [CAT2.Request]\;([CAT.Request]\;2.1) \\
= & AT(c,s) = 0 \;\wedge\; Cl(c,s) = 0 \;\wedge\; Se(c,s) = 1 \;\wedge\; m2(NS(c,s), _, t', c, s) \in history \\
 & \qquad \Rightarrow\; t = t'.
\end{array}$$

		Required by:	To preserve:
3.1	$Server(c,s) = 1 \ \wedge \ m2(NS(c,s),\_,\_,c,s) \in history \ \Rightarrow$ $Client(c,s) \neq 0$	*Request* *Transact*	2.1 2.1
3.2	$Server(c,s) = 1 \ \wedge \ m2(NS(c,s),\_,\_,c,s) \in history \ \wedge$ $Client(c,s) = 3 \ \Rightarrow$ $m3(NC(c,s),c) \notin history$	*Confirm*	2.2
3.3	$Client(c,s) = 2 \ \Rightarrow \ AuthTrans(c,s) = 1$	*SendMsg*2	2.2
3.4	$Server(c,s) = 2 \ \Rightarrow \ AuthTrans(c,s) = 2$	*SendMsg*3	2.3
3.5	$Client(c,s) = 3 \ \wedge \ m2(\_,NC(c,s),\_,c,s') \in history \ \Rightarrow$ $s = s'$	*SendMsg*3	2.3
3.6	$CRITICAL \cap VISIBLE = \{\}$	*Fake*	2.2, 2.3

where $c \notin \mathcal{C} \ \wedge \ s \notin \mathcal{C} \ \wedge \ s' \notin \mathcal{C}$

Figure 12: Invariant 3

The condition $t = t'$ simply cannot follow from the antecedent of the implication so we would need to introduce this whole implication as a new obligation. Before doing so, we generalise it to

$$AT(c,s) = 0 \ \wedge \ Cl(c,s) = 0 \ \wedge \ Se(c,s) = 1 \ \wedge \ m2(NS(c,s),\_,t',c,s) \in history \ \Rightarrow \ false.$$

This is in turn generalised to

$$Se(c,s) = 1 \ \wedge \ m2(NS(c,s),\_,t',c,s) \in history \ \Rightarrow \ Cl(c,s) \neq 0.$$

which is the same as the final clause in the previous derivation. Care needs to be taken not to over-generalise to a clause which is not invariant and this requires intuition about the refinement.

All of the extra clauses that arise from showing that $I1 \wedge I2$ is a gluing invariant are shown in Figure 12. Here the fourth column shows which previous clause the new clause is required for. For example, Clause 3.1 is required when showing that the *Transact* and *Request* actions establish 2.1 (as we have just shown).

The introduction of Clause 3.6 is important in ensuring that the *Fake* action does not violate authentication. Consider Clause 2.2 which ensures that the concrete *Transact* is only enabled when the abstract *Transact* is enabled. Because the *Fake* action could add messages to *history*, in particular $m2(NS(c,s),\_,t,c,s)$, it could violate Clause 2.2 by making the antecedent of Clause 2.2 true, when its consequent was false. However, Clause 3.6 prevents this by preventing the intruder from using $NS(c,s)$ in a message when $Server(c,s) = 1$, i.e., when $NS(c,s) \in CRITICAL$. Similarly, Clause 3.6 is required to ensure that *Fake* preserves Clause 2.3.

Showing that the clauses of Figure 12 are established gives rise in turn to the extra clauses shown in Figure 13. Similarly the clause of Figure 13 give rise to Clause 5.1 of Figure 14. Clause 5.1 does not give rise to any new clauses which means we are done. That is, we have arrived at a gluing invariant that works, i.e., $I1 \wedge \ldots \wedge I5$, and, in the process of doing this, we have proven the refinement obligations in an incremental fashion.

5 Observations on Refinement

In this section we make some observations on the applicability of the incremental refinement rule and also on the way in which the refinement just presented would have failed if we had used the original flawed Needham-Schroeder protocol.

		Required by:	To preserve:
4.1	$Client(c,s) = 0 \;\Rightarrow\; Server(c,s) \neq 2$	*Request*	3.4
4.2	$Server(c,s) = 1 \;\wedge\; m2(NS(c,s), \_, \_, c, s) \in history \;\Rightarrow$ $Client(c,s) \neq 2$	*Transact*	3.3
4.3	$Client(c,s) = 3 \;\wedge\; m3(NC(c,s), c) \in history \;\Rightarrow$ $Server(c,s) \neq 2$	*Confirm*	3.4
4.4	$Client(c,s) = 1 \;\Rightarrow\; AuthTrans(c,s) = 1$	*SendMsg*0	3.3
4.5	$m1(ns, s', \_) \in history \;\wedge\; s' \in \mathcal{C} \;\Rightarrow\; ns \notin CRITICAL$	*SendMsg*2	3.6
4.6	$m2(\_, nc, \_, c', s) \in history \;\wedge\; c' \in \mathcal{C} \;\Rightarrow\; nc \notin CRITICAL$	*SendMsg*3	3.6

where $c \notin \mathcal{C} \;\wedge\; s \notin \mathcal{C}$

Figure 13: Invariant 4

		Required by:	To preserve:
5.1	$Server(c,s) = 1 \;\wedge\; m2(NS(c,s), \_, \_, c, s) \in history \;\Rightarrow$ $Client(c,s) \neq 1$	*Transact* *SendMsg*0	4.4 4.2

where $c \notin \mathcal{C} \;\wedge\; s \notin \mathcal{C}$

Figure 14: Invariant 5

5.1 Applicability of Incremental Refinement

It is not always possible to apply the incremental rule of Section 3.3. Recall that the rule was:

$$\frac{\begin{array}{c} I \wedge J \Rightarrow [T](\langle S \rangle I) \\ I \wedge J \Rightarrow [T]([S]J) \end{array}}{I \wedge J \Rightarrow [T](\langle S \rangle I \wedge J)}$$

In the case where S makes nondeterministic assignments to variables mentioned in the new clause J, the second obligation, $I \wedge J \Rightarrow [T]([S]J)$, may be impossible to verify. In that case, it will be necessary to verify $I \wedge J \Rightarrow [T](\langle S \rangle I \wedge J)$ directly without taking advantage of the modularity provided by the rule. For example, suppose b is a bag of type T and s is a sequence of type T and we have:

$$\begin{array}{lcl} S & \hat{=} & \text{ANY } x' \text{ WHERE } x' \in b \text{ THEN } x := x' \;||\; b := b - \{x'\} \text{ END} \\ T & \hat{=} & s \neq \langle\rangle \implies y := hd(s) \;||\; s := tl(s). \end{array}$$

We might want a correspondence between x and y so we could start with $x = y$ as the initial invariant. Some calculation gives:

$$[T](\langle S \rangle x = y) \quad = \quad s \neq \langle\,\rangle \Rightarrow b \neq \{\} \wedge hd(s) \in b.$$

We see that right hand side is entailed by the clause $b = bag(s)$[8] (this is an example of generalisation) and so we introduce this as a new clause J. However, if we use the incremental refinement rule to add this to the invariant, we find that $[T]([S]J)$ does not follow from $I \wedge J$: Calculation yields

$$[T]([S]b = bag(s)) \quad = \quad s \neq \langle\,\rangle \Rightarrow (\forall\, x \in b \cdot b - \{x\} = bag(tl(s))).$$

The right-hand-side does not follow from $b = bag(s)$.

[8] $bag(s)$ is the bag of elements in sequence s.

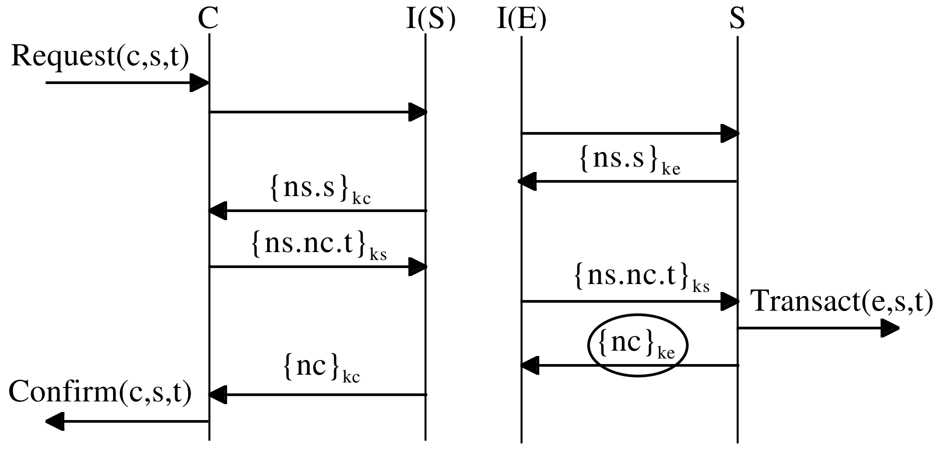

Figure 15: Attack on original Needham-Schroeder protocol.

Many of the new obligations introduced in the refinement of the authenticated transaction service were conditional clauses of the form $G \Rightarrow J$. The condition G arises from the guard of an action that requires J in order to preserve some other clause of the invariant. It may still be possible to apply the incremental refinement rule with such a conditional clause in the case of a nondeterministic abstract action: suppose we wish to verify that abstract action S and corresponding concrete action T together preserve $G \Rightarrow J$; this will be easy in the case that S and T make assignments that make G false; it will also be easy in that case that it is not possible for S and T to be enabled while G holds. These two situations arose several times when verifying the refinement of the authenticated transaction service since many of the actions have disjoint guards and many of the actions disable themselves. More experience with a range of case studies is required in order to have a more complete picture of the applicability of the incremental refinement rule.

5.2 Failure of the original Needham-Schroeder protocol

We will now look at why the refinement would have failed if we had based the protocol on the original Needham-Schroeder Public-key Authentication Protocol which is vulnerable to attack. The difference between the two protocols is that, in the fixed version, the identity of the client c is included in the encrypted message 2 send from the client to the server, whereas, in the original protocol, the clients identity is not included, i.e., the message is of the form $\{ns.nc.t\}_{ks}$ instead of $\{ns.nc.t.c\}_{ks}$.

The attack on the original protocol identified by Lowe [16] involves a single intruder and is represented in Figure 15, where client identities are not included in the message 2 from client to server. Here C and S are non-compromised agents, while E is a compromised agent (possibly the intruder itself). $I(S)$ and $I(E)$ represent the intruder masquerading as S and E respectively. When C initiates a transaction with S, the intruder intervenes and initiates a parallel run of the protocol between E and S. The nonce ns generated by the server for the second protocol run is re-used by the intruder in the fake response from the server in the first protocol run. The response from the client in the first protocol run, $\{ns.nc.t\}_{ks}$, is replayed by the intruder in the second protocol run causing $\mathit{Transact}(e, s, t)$ to occur. The server responds to E with the message $\{nc\}_{ke}$ in the second protocol run, allowing the intruder to generate a fake response, $\{nc\}_{kc}$ in the first protocol run.

The attack of Figure 15 represents a failure of refinement since $\mathit{Confirm}(c, s, t)$ occurs without $\mathit{Transact}(c, s, t)$ having happened. Note that the occurrence of $\mathit{Transact}(e, s, t)$ is not a failure of refinement since E is compromised anyway. The circled message in the second protocol run represents the point at which the abstraction invariant of Section 4.3 is violated: when $\{nc\}_{ke}$ is sent, nc is still in the set $CRITICAL$ from the first protocol run, and now it becomes visible to the intruder as well. This violates Clause 3.6 of the abstraction invariant:

$$CRITICAL \cap VISIBLE = \{\}.$$

It is interesting to note that, while the abstract authenticated transaction specification is not violated until after the $Confirm(c, s, t)$ occurs, the abstraction invariant used for the refinement is already violated when $\{nc\}_{ke}$ is sent.

6 Further Refinement of Authenticated Service

In the refined system presented in the previous section, the receipt of a message is coupled with the action to be taken on its receipt. For example, the $SendMsg1$ action is guarded by $m0 \in chan(c, s)$ and has the effect of removing $m0$ from $chan(c, s)$. In this section we perform a further refinement where message receipt is modelled explicitly by separate actions, one for clients and one for servers. Sending and receipt of messages represents the way in which agents interact directly with the communications network and having separate actions for sending and receipt of messages makes it possible to partition state of the agents from the state network in such a way that interaction is based purely on synchronisation over shared actions. At least this is what we are aiming for in principle. We will see later that the limitations caused by not modelling probability explicitly prevent a full partitioning of the state.

After outlining this refinement step, we outline the partitioning of the system into three parallel subsystems modelling clients, servers, and the network over which they communicate and expose the limitations.

6.1 Separating message receipt

In $CompAuthTrans2$, each $chan(c, s)$ modelled a bi-directional channel between client and server. In $CompAuthTrans3$, each $chan(c, s)$ is replaced by two channels uni-directional channels $cs\_chan(c, s)$, modelling messages being sent from client to server, and $sc\_chan(c, s)$, modelling messages being sent from server to client:

REFINEMENT $CompAuthTrans3$
REFINES $CompAuthTrans2$
INCLUDES $client.Client2Mch, server.Server2Mch$
VARIABLES
$cs\_chan, sc\_chan : (Agent \times Agent) \to bag\ Msg$

INITIALISATION $cs\_chan := \lambda\, c, s.\{\} \quad || \quad sc\_chan := \lambda\, c, s.\{\} \quad || \quad history := \{\}$

$CompAuthTrans3$ includes refined versions of the $Client$ and $Server$ CSP processes. These are shown in Figures 16 and 17. In the $Client2$ CSP process, an event modelling receipt of messages has been added ($CliRecvMsg$). Also, messages and nonces have been added as explicit parameters to events which means that values for messages and nonces are visible in the CSP process and can be constrained by it. For example, the $SendMsg0$ event of Figure 16 dictates the message value should be $m0$.

Making these parameters visible also makes it possible to add explicit error handling to the CSP process. The first $CliRecvMsg$ event of Figures 16 places no constraint on the value of the message m, so it is followed by a conditional statement that allows $SendMsg2$ to occur only if m is a message 1 and otherwise causes a $ClientError$. We do not specify error handling any further here.

Note that the variables NC and NS of $CompAuthTrans2$ do not appear explicitly in $CompAuthTrans3$. Instead they are implied by the CSP processes. For example, nc is an input parameter of the

PROCESS $\mathit{Client2} \ = \ ||| \ c, s \cdot \mathit{InitClient2}[c, s]$ WHERE

$\mathit{InitClient2}[c, s] \ =$
$\quad \mathit{Request}.c.s?t \ \rightarrow$
$\quad \mathit{SendMsg0}.c.s.m0 \ \rightarrow$
$\quad \mathit{CliRecvMsg}.c.s?m \ \rightarrow$
$\quad$ IF $\ m = m1(\_, s, c)$
$\quad$ THEN $\ \mathit{SendMsg2}.c.s\ ?nc\ .m2(\mathit{get\_ns}(m), nc, t, c, s) \ \rightarrow$
$\qquad\qquad \mathit{CliRecvMsg}.c.s?m \ \rightarrow$
$\qquad\qquad$ IF $\ m = m3(nc, c)$
$\qquad\qquad$ THEN $\ \mathit{Response}.c.s.t \ \rightarrow \ \mathit{InitClient2}[c, s]$
$\qquad\qquad$ ELSE $\ \mathit{ClientError}.c.s \ \rightarrow \ \mathit{STOP}$
$\quad$ ELSE $\ \mathit{ClientError}.c.s \ \rightarrow \ \mathit{STOP}$

END

Figure 16: Refined client CSP process

PROCESS $\mathit{Server2} \ = \ ||| \ c, s \cdot \mathit{InitServer2}[c, s]$ WHERE

$\mathit{InitServer2}[c, s] \ =$
$\quad \mathit{SerRecvMsg}.c.s?m \ \rightarrow$
$\quad$ IF $\ m = m0$
$\quad$ THEN $\ \mathit{SendMsg1}.c.s?ns.m1(ns, s, c) \ \rightarrow$
$\qquad\qquad \mathit{SerRecvMsg}.c.s?m \ \rightarrow$
$\qquad\qquad$ IF $\ m = m2(ns, \_, \_, c, s)$
$\qquad\qquad$ THEN $\ \mathit{Transact}.c.s.\mathit{get\_t}(m) \ \rightarrow$
$\qquad\qquad\qquad\qquad \mathit{SendMsg3}.c.s.m3(\mathit{get\_nc}(m), c) \ \rightarrow \ \mathit{InitServer2}[c, s]$
$\qquad\qquad$ ELSE $\ \mathit{ServerError}.c.s \ \rightarrow \ \mathit{STOP}$
$\quad$ ELSE $\ \mathit{ServerError}.c.s \ \rightarrow \ \mathit{STOP}$

END

Figure 17: Refined server CSP process

*SendMsg*2 event in Figure 16 and remains in scope from that point on. As described in Section 2.2, this means an explicit variable has to be introduced to store it, which we refer to as NC_{C2}, in the B machine generated from the CSP process. Similarly, a variable to store the value of the input message of *CliReccvMsg*, which we refer to as M_{C2}, is introduced in the generated machine. Likewise, NS_{S2} and M_{S2} are introduced in the B machine generated from the *Server*3 CSP process.

The actions of *CompAuthTrans*2 are given in Appendix A The incrementally developed invariant used to verify that *CompAuthTrans*2 is refined by *CompAuthTrans*3 is presented in Appendix B.

6.2 Decomposition

To recap, the state variables of *CompAuthTrans*3 are:

- *cs_chan*, *sc_chan*, *history*

- $Client2$, T_{C2}, M_{C2}, NC_{C2}
- $Server2$, M_{S2}, NS_{S2}

The first three were explicitly declared previously, the next four are generated from the *Client*2 CSP process, and the last three are generated from the *Server*2 CSP process.

We decompose *CompAuthTrans*3 into three subsystems based on the above partitioning of state variables: *InsecureNetwork*, *DecoupledClients* and *DecoupledServers* respectively. These machines are shown in Appendix C. *DecoupledClients* represents the interleaved behaviour of all clients while *DecoupledServers* represents the interleaved behaviour of all servers. Clients and servers communicate indirectly via the *InsecureNetwork*. Interaction between agents and the network is based on synchronisation over shared actions. *InsecureNetwork* contains action for sending and receiving of messages by both clients and servers. It also contains the *Fake* action which is why we refer to the network as insecure. The decoupled client and server subsystems include the *Client*2 and *Server*2 CSP processes respectively and the only extra behavioural specification that they provide is the selection of nonces.

The coupling between the subsystems is defined by a machine *CompAuthTrans*4 (Appendix C) which includes *InsecureNetwork*, *DecoupledClients* and *DecoupledServers*. Shared actions are modelled by simultaneous calls to the interacting subsystems. For example, the *SendMsg*0 action of *CompAuthTrans*4 is defined as follows:

$SendMsg0(c, s) \mathrel{\hat{=}}$
    ANY m THEN
        $d\_client.SendMsg0(c, s, m) \;\;||\;\; nw.CliSendMsg(c, s, m)$
    END

Properties of || mean that this composite action is enabled only when both constituent actions are enabled thus modelling synchronisation between *DecoupledClients* and *InsecureNetwork* via the shared actions named *SendMsg*0 and *CliSendMsg* in the respective machines.

In [7, 8], it is shown that this way of composing action systems corresponds to parallel composition in the CSP traces and failures-divergence models. This is true provided the component systems do not share state variables, i.e., the only interaction is via synchronised actions. Our decomposition of *CompAuthTrans*3 into three subsystems *InsecureNetwork*, *DecoupledClients* and *DecoupledServers* violates this proviso because there is sharing of state: The *Fake* operation of *InsecureNetwork* reads from *Client*2, NC_{C2}, *Server*2 and NS_{S2}, which are variables of *DecoupledClients* and *DecoupledServers*. Likewise the *SendMsg*2 action of *DecoupledClients* and the *SendMsg*1 action of *DecoupledServers* both read from the *history* variable of *InsecureNetwork*.

The sharing of variables between machines is an artifact of the way we are modelling probabilistic selection of nonces. Even if we ignore the fact that the decomposition does not correspond to CSP composition, it is still not possible to further refine the subsystems fully within the action system refinement approach. In practice, the selection of a fresh nonces in the *SendMsg*1 action, for example, would be random. But we cannot refine selection of the nonce ns constrained by $ns \notin nonces(history)$ by a more arbitrary choice as this would not be a valid refinement. We could choose to take pragmatic leap and informally take the view that a nondeterministic choice which is constrained to prevent choices with very low probability may be replaced by a probabilistic choice. But such a leap cannot be justified within a formalism that does not treat probability.

7 Refinement of Confidential Communication Service

In this section we outline another example of refinement of an abstract secure service. We refine the confidential communications service *ConfComms* of Figure 2 by a system *ConfComms*2 which uses symmetric keys to encrypt plain text to be communicated over an insecure network. Recall that *ConfComms* has actions for sending and receiving of plain text between parties A and B, as well as a *Reveal* action that allows a spy access to plain text when A or B is compromised.

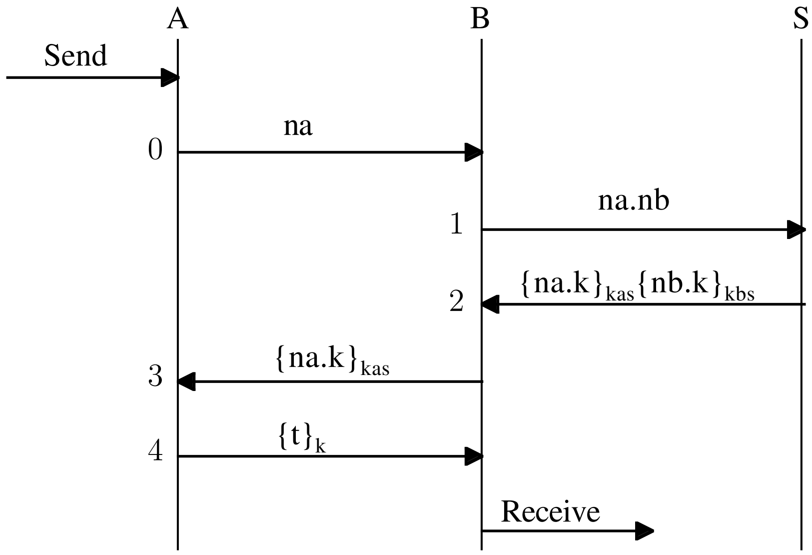

Figure 18: Symmetric session key protocol.

Whenever plain text is to be sent from A to B in the refined system, a fresh session key is requested from a trusted key server using a simplified version of the Otway-Rees protocol [24]. The protocol is shown in Figure 18. In the protocol, A sends a nonce na to B, then B forwards this, along with its own nonce nb, to the server S. S generates a fresh key k and separately encrypts it for both A and B. For A, the key k is encrypted, along with na, using a secret symmetric key kas shared only by A and S, thus only A (and S) may decrypt this part of the message. Similarly for B. The result is that k is known to A and B but not to other parties and k is then used to encrypt the plain text t sent from A to B. In practice, a session might involve several exchanges of plain text in both directions between A and B, but for simplicity we allow a session to have just a single transfer of plain text from A to B.

An agent is said to be compromised, if the key it shares with the server is known to an intruder. Knowing the shared key for either A or B would give an intruder access to the session key k. In the refinement, we also assume that an intruder may try to pretend to be the server and generate the message in step 2 of the protocol. This would mean that A and B could end up using a session key known to the intruder. Unlike in the authentication protocol of Section 4, where the intruder knows all public keys and so could generate a message encrypted for any agent, in the case of the symmetric session key protocol, the intruder can generate a message encrypted for a compromised agent only. The agent can also replay encrypted message components hence the need for the nonces na and nb.

We introduce the following types to model the different messages that are exchanged in the protocol:

$$
\begin{array}{rcl}
NKA & \hat{=} & Nonce \times Key \times Agent \\
Msg & ::= & m0(Nonce) \mid \\
 & & m1(Nonce \times Nonce) \mid \\
 & & m2(NKA \times NKA) \mid \\
 & & m3(NKA) \mid \\
 & & m4(Text \times Key)
\end{array}
$$

A value $(n, k, a) : NKA$ represents message $n.k$ encrypted with shared key kas. We assume that the intruder is able to access such message components from a transmitted message and use them to construct new messages, e.g., if $m2(x, y)$ and $m2(x', y')$ have been transmitted, then the intruder can send a fake $m2(x, y')$. For a set of messages h, $get\_NKA(h)$ gives the set of encrypted message components in h:

$$get\_NKA(h) \;=\; \{\ x \mid m2(x,\_) \in h \ \lor\ m2(\_,x) \in h \ \lor\ m3(x) \in h\ \}$$

Similarly we have functions giving the set of nonces and set of session keys in a set of messages:

$$\begin{aligned} nonces(h) \;&=\; \{\ n \mid m0(n) \in h \ \lor\ m1(n,\_) \in h \ \lor\ m1(\_,n) \in h\ \} \cup \\ &\;\; \{\ n \mid (n,\_,\_) \in get\_NKA(h)\ \} \\ keys(h) \;&=\; \{\ k \mid (\_,k,\_) \in get\_NKA(h)\ \} \end{aligned}$$

The initial part of the refined machine is as follows:

REFINEMENT *ConfComms2*
REFINES *ConfComms*
INCLUDES *aside.AsideMch*, *bside.BsideMch*
VARIABLES
$\quad chan : (Agent \times Agent) \rightarrow (bag\ Msg)$
$\quad history : \mathbb{P}\ Msg$
$\quad NA, NB : (Agent \times Agent) \nrightarrow Nonce$
$\quad K : (Agent \times Agent) \rightarrow Key$

As before, *chan* models messages in transit in the network, *history* represents all messages sent to date, and NA, NB and K model the most recently selected values for na and nb and k respectively for each pair of agents a, b.

The refinement includes two machines *AsideMch* and *BsideMch* generated from the CSP machines of Figure 19 which control the ordering of actions for an agent A and an agent B respectively. The refinement has *Send* and *Receive* actions corresponding to the actions of the abstract machine as well as auxiliary actions modelling the sending of protocol message. These actions are shown in Appendix D. We now present the *Reveal* action, which allows the intruder access to certain plain text, and an auxiliary action, *Fake*, which allows the intruder to send fake messages from the key server.

The used session keys known to the intruder, $KNOWN$, are those session keys that appear in a message component encrypted with the key of a compromised agent ($\mathcal{C}$ is the set of compromised agents):

$$KNOWN \;\hat{=}\; \{\ k \mid \exists\, c \in \mathcal{C} \cdot (\_,k,c) \in get\_NKA(history)\ \}$$

Text encrypted with a known key may be revealed to the intruder through the *Reveal* action:

$Reveal(a,b,t) \;\hat{=}$
$\quad$ SELECT $\ m4(t,k) \in chan(a,b) \ \land\ k \in KNOWN$ THEN $\ skip$ END

The intruder can fake an encrypted message component (n,k,a) if a is compromised or if (n,k,a) has been transmitted previously:

$$CAN\_FAKE(n,k,a) \;\hat{=}\; a \in \mathcal{C} \quad\lor\quad (n,k,a) \in get\_NKA(history)$$

The intruder can send a fake message from the server if it can fake both of the encrypted message components:

$Fake(a,b) \;\hat{=}$
$\quad$ ANY $\ x, y$ WHERE $\ CAN\_FAKE(x) \ \land\ CAN\_FAKE(y)$
$\quad$ THEN
$\qquad chan(a,b) := chan(a,b) + \{m2(x,y)\} \ \ ||$
$\qquad history := history \cup \{m2(x,y)\}$
$\quad$ END

PROCESS $Aside \;=\; ||| \; a, b \cdot InitAside[a, b]$ WHERE

$$
\begin{array}{l}
InitAside[a, b] \;= \\
\quad Send.a.b?t \;\rightarrow \\
\quad SendMsg0.a.b \rightarrow \\
\quad SendMsg4.a.b.t \rightarrow InitAside[a, b]
\end{array}
$$

END

PROCESS $Bside \;=\; ||| \; a, b \cdot InitBside[a, b]$ WHERE

$$
\begin{array}{l}
InitBside[a, b] \;= \\
\quad SendMsg1.a.b \rightarrow \\
\quad SendMsg3.a.b \rightarrow \\
\quad Receive.a.b \rightarrow InitBside[a, b]
\end{array}
$$

END

Figure 19: CSP constraints for symmetric session key system.

1	$m \in chan(a, b) \;\Rightarrow\; m \in history$		
		Required by:	To preserve:
2.1	$m4(t, k) \in chan(a, b) \;\wedge\; a \notin \mathcal{C} \;\wedge\; b \notin \mathcal{C}$ $\;\Rightarrow\; k \notin KNOWN$	*Reveal*	1
2.2	$m4(t, \_) \in chan(a, b) \;\Rightarrow\; T_A(a, b) \in comms(a, b)$	*Reveal*	1
3.1	$a \notin \mathcal{C} \;\wedge\; b \notin \mathcal{C} \;\wedge\; Aside(a, b) = 2 \;\wedge$ $(NA(a, b), k, a) \in get\_NKA(history)$ $\;\Rightarrow\; k \notin KNOWN$	*SendMsg*4	2.1
3.2	$Aside(a, b) = 2 \;\Rightarrow\; T_A(a, b) \in comms(a, b)$	*SendMsg*4	2.2
4.1	$Aside(a, b) = 1 \;\Rightarrow\; T_A(a, b) \in comms(a, b)$	*SendMsg*0	3.2

Figure 20: Invariant for Refinement of Communications Service

The incrementally developed invariant used to verify that *ConfComms*2 is a refinement of *ConfComms* is shown in Figure 20. In this figure, $T_A(a, b)$ is part of the *AsideMch* machine generated from the *Aside* CSP process and used to store the value of t chosen by the *Send* action. This invariant is simpler than the one developed in Section 4.3 for the authenticated service, mainly because the encryption is symmetric key rather than public key. A message can only be faked if the key with which it is encrypted is known to the intruder, whereas, with public-key encryption, the public keys used to encrypt messages are known to everyone. This means that the secrecy of nonces does not need to be preserved. Indeed, the nonces are transmitted in plain text in the first two steps of the protocol.

*ConfComms*2 can be further refined, separating message receipt from message sending as in Section 6.1, and then decomposed as in Section 6.2. The limitation caused by not modelling probability also arises with this example. Details of the further refinement are omitted.

8 Related Work

A good overview of formal analysis of security protocols is found in a paper by Clark and Jacob [11].

Several researchers have developed specialised logics for the purpose of analysing security protocols. Usually these take the form of belief logics since they are used to reason about the evolution of what parties to protocols believe about each others state. Perhaps the most well-known of these is the BAN logic of Burrows, Abadi, and Needham [6]. BAN allows the assumptions and goals of a protocol to be stated in abstract terms, as well as an abstract model of the protocol to be described. Special rules of inference are provided for verifying that a protocol meets its goals under the stated assumptions. Implicit in these inference rules is the ability of an intruder to eavesdrop and masquerade. GNY, developed by Gong, Needham and Yahalom [14], extends BAN in several ways, for example, by providing for different levels of trust.

Comparing our approach with the use of BAN logic, we find that the use of action systems and refinement results in a more verbose specification and proof than is possible with BAN. This is partly because the B notation is verbose by nature with its use of keywords, and partly because we need to be much more explicit in our modelling since we are starting with a very general purpose modelling framework. The refined systems include explicit models of the faking of messages by an intruder and the constraints on such faking. We use standard first-order logic and set theory for modelling and proof, whereas BAN is very specialised towards modelling and reasoning about the believes of participants in security protocols.

The NRL Protocol Analyser [19] and the Mitre Interogator [20] are special-purpose tools for analysing protocols based on term rewriting. Both tools attempt to find runs of protocols that lead to insecure states and both have been used to find flaws in security protocols.

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 [27, 28]. Schneider uses a proof-theoretic approach to check that CSP processes modelling security protocols satisfy trace properties, i.e., predicates on the traces of a process. Although not addressed here, it is possible to show that an action system satisfies a trace property by showing that it refines the most general action system satisfying the property [9].

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 in much the same way that we specified the authenticated transaction service. 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 [17]. As with all model-checkers, CSP processes analysed by FDR must be finite state, and effort is usually required to get processes into a form that is model-checkable.

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, Clause 3.6 in the invariant of the first refinement of the authenticated transaction service, which says that critical nonces should not be visible to the intruder, provides insight into why the (fixed) Needham-Schroeder protocol works. This insight should be transferable to other public-key protocols that use nonces to prevent replay attacks. Model-checking does complement a proof-theoretic approach as it allows developers to quickly experiment with alternative designs.

Paulson has developed an inductive approach to analysing security protocols using the Isabelle theorem-proving system [25]. Here 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.

Sinclair uses action systems to analyse information flow in security systems in [29]. Information flow involves the ability to infer information about events of a high security level through the occurrence of events of a lower security level. Such information flow is usually undesirable. Earlier work by Roscoe formulates such information flow in terms of deterministic behaviour in CSP [26]. Sinclair transfers this work to action systems using Morgan's connection [21] between CSP and action systems.

The closest work to our approach is the use of B by Bieber el al [5, 4] and the use of IO-automata by Lynch [18]. Both of these approaches involve the use of state machine-like formalisms for modelling both the desired security properties and the protocols under analysis, and both approaches use refinement as a proof technique. Unlike our approach, Lynch does not model masquerading though she does model eavesdropping. Bieber et al model eavesdropping and masquerading though they do not treat examples involving public-key encryption nor the use of nonces to prevent replays. Neither Lynch nor Bieber provide much insight into the derivation of abstraction invariants for refinements.

Many of the above approaches have proved very valuable in helping to find flaws in existing security protocols and in clarifying various concepts in security. None of these approaches attempt the verification of or refinement to a distributed design, as we do in Section 6, and therefore they do not encounter the problem we encountered when trying to replace a choice constrained by global information by a random choice. Furthermore, like our approach, none of these approaches use any probabilistic reasoning to deal with randomness.

9 Conclusions

The work described in this paper evolved from earlier work by the author on using refinement to analyse the Needham-Schroeder Public Key Authentication Protocol [9]. This present paper represents a major improvement on that earlier work in several ways. Firstly the abstract specification in [9] simply specified the exchange of nonces whereas here we have represented the abstract system as a more realistic end-user system. This makes the the service which the protocol is helping to achieve much clearer and makes it easy to be assured that the specification is a reasonable and valid one.

At a technical level, the refinement in [9] was more simplistic than the one in Section 4. In [9], once a nonce was used by a non-compromised agent it could never be seen by the intruder. Here we see that this is unnecessarily strong and that we only need to restrict its visibility while it is critical, i.e., while the issuer of the nonce is waiting for it to be returned. After that we allow it to be used by the intruder; freshness ensures that this does not lead to a failure of refinement.

The earlier work only achieved one level of refinement which was not sufficient for decomposition to a distributed architecture. Also, the earlier work used action systems only and did not use CSP to specify services and refinements. We found CSP notation to be more convenient for modelling the ordering of events, especially at the refinement level, than direct action system representations of control state. Of course, we still require access to the control state to verify refinements.

The most serious weakness of the earlier work was that the abstraction invariant was derived in an ad-hoc manner. In this paper, we introduced the incremental refinement technique both to derive the abstraction invariant and to verify the refinement in a more structured and incremental manner. Even having fully automated theorem proving would not immediately allow us to verify refinements automatically; a suitable abstraction invariant still has to be derived and these can be quite involved, as some of the examples in this paper show. To some extent, the incremental derivation of the invariant is fairly mechanical, especially the derivation of invariants relating the control variables of the abstract and concrete systems. The link with iterative techniques used in symbolic model checking [12] to find characterisations of properties needs to be investigated. However, we found that the iterative derivation was not completely mechanical. Reasoning is required in places to recognise when new obligations are entailed by the existing invariant. When new obligations are discovered, it is sometimes convenient to generalise them, i.e., make them stronger. For example, this was done in the derivation of Clause 3.1 in Section 4.3. Care needs to be taken not to over-generalise to a clause which is not invariant and this requires intuition about the refinement. In Section 5.1, we pointed out cases where the incremental refinement rule cannot be applied.

The proof obligations for the refinements in this paper correspond to those generated by tools for B such as Steria's *Atelier-B* and B-Core's *B-Toolkit*. A thorough analysis of the performance of these tools on refinements such as those presented here remains to be undertaken. However, the proof support in these B tools does not currently support incremental refinement making the proof effort, especially the derivation of the invariant, much more difficult. Tool support which support the interactive derivation of the refinement invariants, in the manner similar to that used here on paper, would be desirable.

At the end of Section 6.2, we highlighted the problems caused by modelling randomness using global constraints on choice rather than using probability. This prevented us from decomposing the detailed designed into parallel systems with completely independent states within the formalism. Morgan Et Al [22] have developed an approach to refinement of (sequential) probabilistic programs and this may in principal be applicable to action system refinement allowing us to treat probability properly. Our experience with reasoning about probabilistic programs is that it can be difficult to relate probability in abstract programs with probability in concrete programs. An ideal solution in the case of security protocols would be a meta-rule describing when it is allowable to replace a globally constrained choice of nonce by a probabilistic choice. This would allow us to use non-probabilistic refinement for the most part, as used in this paper, and then use the meta-rule to introduce randomness in the distributed refinement. Such a meta-rule should be justified in terms of some probabilistic model.

Acknowledgements

Thanks to Jean-Raymond Abrial, Stefan Hallerstede, Colin O'Halloran, Louis Mussat, Ulrich Ultes-Nitsche and Peter Ryan for their comments and help. Thanks also to the referees of an earlier version of this paper for useful comments which lead to a comprehensive revision of the work.

A Actions for Second Refinement of Authenticated Service

The actions of *CompAuthTrans*2 are: *Request*, *Confirm*, *Transact*, *SendMsg*0, *SendMsg*1, *SendMsg*2, *SendMsg*3, *Fake*, *CliRecvMsg*, *SerRecvMsg*, *ClientError*, and *ServerError*.

Each of the actions of *CompAuthTrans*3 includes a call to either the *Client*2 or *Server*2 CSP processes.

*SendMsg*0 includes a call to the corresponding client CSP action, adds the message to the channel from the client to the server and stores the message in *history*:

$$
\begin{array}{l}
\mathit{SendMsg}0(c,s) \mathrel{\hat{=}} \\
\quad \text{ANY } \; m \; \text{ THEN} \\
\quad\quad \mathit{client.SendMsg}0(c,s,m) \parallel \\
\quad\quad \mathit{cs\_chan}(c,s) := \mathit{cs\_chan}(c,s) + \{m\} \;\parallel\; \mathit{history} := \mathit{history} \cup \{m\} \\
\quad \text{END;}
\end{array}
$$

*SendMsg*1 adds a message on the channel from the server to the client as well as selecting a fresh nonce:

$$
\begin{array}{l}
\mathit{SendMsg}1(c,s) \mathrel{\hat{=}} \\
\quad \text{ANY } \; ns, m \; \text{ WHERE } \; ns \notin \mathit{nonces}(\mathit{history}) \text{ THEN} \\
\quad\quad \mathit{server.SendMsg}1(c,s,ns,m) \;\parallel \\
\quad\quad \mathit{sc\_chan}(c,s) := \mathit{sc\_chan}(c,s) + \{m\} \;\parallel\; \mathit{history} := \mathit{history} \cup \{m\} \\
\quad \text{END;}
\end{array}
$$

$SendMsg2$ and $SendMsg3$ are similar:

$SendMsg2(c, s) \mathrel{\hat{=}}$
ANY nc, m WHERE $nc \notin nonces(history)$ THEN
$server.SendMsg2(c, s, nc, m)$ $||$
$cs\_chan(c, s) := cs\_chan(c, s) + \{m\}$ $||$ $history := history \cup \{m\}$
END;

$SendMsg3(c, s) \mathrel{\hat{=}}$
ANY m THEN
$server.SendMsg3(c, s, m)$ $||$
$sc\_chan(c, s) := sc\_chan(c, s) + \{m\}$ $||$ $history := history \cup \{m\}$
END;

CliRecvMsg selects a messages from the channel from the server to the client and removes the message from the channel as well as including a call to the client CSP machine:

$CliRecvMsg(c, s) \mathrel{\hat{=}}$
ANY m WHERE $m \in sc\_chan(c, s)$ THEN
$client.CliRecvMsg(c, s, m)$ $||$
$sc\_chan4(c, s) := sc\_chan(c, s) - \{m\}$
END;

Similarly for *SerRecvMsg*:

$SerRecvMsg(c, s) \mathrel{\hat{=}}$
ANY m WHERE $m \in cs\_chan(c, s)$ THEN
$server.SerRecvMsg(c, s, m)$ $||$
$cs\_chan(c, s) := cs\_chan(c, s) - \{m\}$
END;

The refined set of critical nonces is defined as follows:

$$\begin{array}{rcl} CRITICAL2 & \hat{=} & \{\ NS_{S2}(c, s) \ \mid\ c \notin \mathcal{C} \ \wedge\ s \notin \mathcal{C} \ \wedge\ Server2(c, s) \in \{2, 3\}\ \} \ \cup \\ & & \{\ NC_{C2}(c, s) \ \mid\ c \notin \mathcal{C} \ \wedge\ s \notin \mathcal{C} \ \wedge\ Client2(c, s) \in \{4, 5\}\ \} \end{array}$$

Here, $Server(c, s) = 2$ immediately after sending a message 1 and $Server(c, s) = 3$ immediately after the subsequent *SerRecvMsg* but before the *Transact* occurs. Similarly for $Client2(c, s) \in \{4, 5\}$.

The *Fake* action adds messages to either channel and the constraint on the selection of nonces is similar to that in *CompAuthTrans2*:

$Fake(c, s) \mathrel{\hat{=}}$
ANY m WHERE
$nonces(m) \subseteq (Nonce \setminus CRITICAL2)$
$\vee \quad m \in history$
THEN
$(cs\_chan(c, s) := sc\_chan(c, s) + \{m\} \ [] \ sc\_chan(c, s) := sc\_chan(c, s) + \{m\})$ $||$
$history := history \cup \{m\}$
END

The remaining actions of *CompAuthTrans3* are simple calls:

$Request(c, s, t) \mathrel{\hat{=}} client.Request(c, s, t)$
$Transact(c, s, t) \mathrel{\hat{=}} server.Transact(c, s, t)$
$Response(c, s, t) \mathrel{\hat{=}} client.Response(c, s, t)$
$ClientError \mathrel{\hat{=}} client.ClientError(c, s)$
$ServerError \mathrel{\hat{=}} server.ServerError(c, s)$

B Invariant for Second Refinement of Authenticated Service

A.1.1	$T_{Client} = T_{Client2}$
A.1.2	$NS_{Server} = NS_{Server2}$
A.1.3	$NC_{Client} = NC_{Client2}$
A.1.4	$history = history2$
A.1.5	$chan(c,s) = cs\_chan(c,s) + sc\_chan(c,s) + \{ M_{Client2}(c,s) \mid Client2(c,s) \in \{3,5,STOP\} \} + \{ M_{Server2}(c,s) \mid Server2(c,s) \in \{1,3,4,STOP\} \}$

The reason that $chan(c,s)$ does not equate simply with $cs\_chan(c,s) + sc\_chan(c,s)$ in A.1.5 is that, in $CompAuthTrans3$, messages are removed from the $cs\_chan(c,s)$ and $sc\_chan(c,s)$ channels (by $CliRecvMsg$ and $SerRecvMsg$) earlier than they are removed from $chan(c,s)$ in $CompAuthTrans2$.

		Required by:	To preserve:
A.2.1	$Client2(c,s) = 0 \Rightarrow Client(c,s) = 0$	*Request*	A.1
A.2.2	$Server2(c,s) = 3 \Rightarrow Server(c,s) = 1$	*Transact*	A.1
A.2.3	$Client2(c,s) = 5 \Rightarrow Client(c,s) = 3$	*Response*	A.1
A.2.4	$Client2(c,s) = 1 \Rightarrow Client(c,s) = 1$	$SendMsg0$	A.1
A.2.5	$Server2(c,s) = 1 \Rightarrow Server(c,s) = 0$	$SendMsg1$	A.1
A.2.6	$Client2(c,s) = 3 \Rightarrow Client(c,s) = 2$	$SendMsg2$	A.1
A.2.7	$Server2(c,s) = 4 \Rightarrow Server(c,s) = 2 \wedge M_{Server2}(c,s) = m2(NS_{Server}(c,s), \_, \_, c, s)$	$SendMsg3$	A.1

		Required by:	To preserve:
A.3.1	$Client2(c,s) = 4 \Rightarrow Client(c,s) = 3$	*CliRecvMsg*	A.2.3
A.3.2	$Client2(c,s) = 2 \Rightarrow Client(c,s) = 2$	*CliRecvMsg*	A.2.6
A.3.3	$Server2(c,s) = 2 \Rightarrow Server(c,s) = 1$	*SerRecvMsg*	A.2.2
A.3.4	$Server2(c,s) = 0 \Rightarrow Server(c,s) = 0$	*SerRecvMsg*	A.2.5

C Decomposition of Authenticated Service

$CompAuthTrans4$ is a construction representing the parallel composition of three subsystems: *InsecureNetwork*, *DecoupledClients*, and *DecoupledServers*.

REFINEMENT $CompAuthTrans4$
REFINES $CompAuthTrans3$
INCLUDES $nw.InsecureNetwork, d\_client.DecoupledClients, d\_server.DecoupledServers$

$Request(c,s,t) \mathrel{\hat=} d\_client.Request(c,s,t)$
$Transact(c,s,t) \mathrel{\hat=} d\_server.Transact(c,s,t)$
$Response(c,s,t) \mathrel{\hat=} d\_client.Response(c,s,t)$
$SendMsg0(c,s) \mathrel{\hat=}$
ANY m THEN $d\_client.SendMsg0(c,s,m) \;||\; nw.CliSendMsg(c,s,m)$ END
$SendMsg1(c,s) \mathrel{\hat=}$
ANY m THEN $d\_server.SendMsg1(c,s,m) \;||\; nw.SerSendMsg(c,s,m)$ END
$SendMsg2(c,s) \mathrel{\hat=}$
ANY m THEN $d\_client.SendMsg2(c,s,m) \;||\; nw.CliSendMsg(c,s,m)$ END
$SendMsg3(c,s) \mathrel{\hat=}$
ANY m THEN $d\_server.SendMsg3(c,s,m) \;||\; nw.SerSendMsg(c,s,m)$ END
$CliRecvMsg(c,s) \mathrel{\hat=}$
ANY m THEN $nw.CliRecvMsg(c,s,m) \;||\; d\_client.CliRecvMsg(c,s,m)$ END
$SerRecvMsg(c,s) \mathrel{\hat=}$
ANY m THEN $nw.SerRecvMsg(c,s,m) \;||\; d\_server.SerRecvMsg(c,s,m)$ END
$ServerError(c,s) \mathrel{\hat=} d\_server.ServerError(c,s)$
$ClientError(c,s) \mathrel{\hat=} d\_client.ClientError(c,s)$
$Fake(c,s) \mathrel{\hat=} nw.Fake(c,s)$

END

MACHINE $InsecureNetwork$
VARIABLES
$cs\_chan, sc\_chan : (Agent \times Agent) \to bag\ Msg$
$history : \mathbb{P}\ Msg$
INITIALISATION
$cs\_chan := \lambda\, c,s.\{\} \quad || \quad sc\_chan := \lambda\, c,s.\{\} \quad || \quad history := \{\}$
OPERATIONS
$CliSendMsg(c,s,m) \mathrel{\hat=}$
$cs\_chan(c,s) := cs\_chan(c,s) + \{m\} \quad || \quad history := history \cup \{m\}$

$CliRecvMsg(c,s,m) \mathrel{\hat=}$
$m \in sc\_chan(c,s) \implies sc\_chan(c,s) := sc\_chan(c,s) - \{m\}$

$SerSendMsg(c,s,m) = \quad sc\_chan(c,s) := sc\_chan(c,s) + \{m\} \quad || \quad history := history \cup \{m\}$

$SerRecvMsg(c,s,m) = \ \hat=$
$m \in cs\_chan(c,s) \implies cs\_chan(c,s) := cs\_chan(c,s) - \{m\}$

$Fake(c,s) \mathrel{\hat=}$
ANY m WHERE
$nonces(m) \subseteq (Nonce \setminus CRITICAL2)$
$\vee \quad m \in history$
THEN
$(cs\_chan(c,s) := cs\_chan(c,s) + \{m\} \;[]\; sc\_chan(c,s) := sc\_chan(c,s) + \{m\}) \;||$
$history := history \cup \{m\}$
END
END

MACHINE *DecoupledClients*
INCLUDES *client.Client2*
OPERATIONS

$Request(c, s, t) \hat{=} client.Request(c, s, t)$
$SendMsg0(c, s, m) \hat{=} client.SendMsg0(c, s, m)$
$CliRecvMsg(c, s, m) \hat{=} client.CliRecvMsg(c, s, m)$

$SendMsg2(c, s, m) \hat{=}$
ANY nc WHERE $nc \notin nonces(history)$ THEN
$client.SendMsg2(c, s, ns, m)$
END

$Response(c, s, t) \hat{=} client.Response(c, s, t)$
$ClientError(c, s) \hat{=} client.ClientError(c, s)$

END

MACHINE *DecoupledServers*
INCLUDES *server.Server2*

$SerRecvMsg(c, s, m) \hat{=} server.SerRecvMsg(c, s, m)$

$SendMsg1(c, s, m) \hat{=}$
ANY ns WHERE $ns \notin nonces(history)$ THEN
$server.SendMsg1(c, s, ns, m)$
END

$Transact(c, s, t) \hat{=} server.Transact(c, s, t)$
$SendMsg3(c, s, m) \hat{=} server.SendMsg3(c, s, m)$
$ServerError(c, s) \hat{=} server.ServerError(c, s)$
END

D Actions for Refinement of Confidential Service

INITIALISATION
$chan := \lambda\, c, s.\{\} \quad || \quad history := \{\}$

OPERATIONS

$Send(a, b, t) \quad \hat{=} \quad aside.Send(a, b, t)$

$SendMsg0(a, b) \quad \hat{=}$
ANY na WHERE $na \notin nonces(history)$ THEN
$aside.SendMsg0(a, b) \quad ||$
$NA(a, b) := na \quad ||$
$chan(a, b) := chan(a, b) + \{m0(na)\} \quad ||$
$history := history + \{m0(na)\}$
END

$SendMsg1(a,b) \ \hat{=}$
ANY na, nb WHERE
$m0(na) \in chan(a,b) \ \wedge \ nb \notin nonces(history)$
THEN
$bside.SendMsg1(a,b) \quad ||$
$NB(a,b) := nb \quad ||$
$chan(a,b) := chan(a,b) - \{m0(na)\} + \{m1(na,nb)\} \quad ||$
$history := history + \{m1(na,nb)\}$
END

$SendMsg2(a,b) \ \hat{=}$
ANY na, nb WHERE
$m1(na,nb) \in chan(a,b) \ \wedge \ k \notin keys(history)$
THEN
$$\begin{aligned} chan(a,b) := \ & chan(a,b) - \{m1(na,nb)\} \\ & + \{m2((na,k,a),(nb,k,b))\} \quad || \end{aligned}$$
$history := history + \{m2((na,k,a),(nb,k,b))\}$
END

$SendMsg3(a,b) \ \hat{=}$
ANY x, k WHERE
$m2(x,(NB(a,b),k,b)) \in chan(a,b)$
THEN
$bside.SendMsg3(a,b) \quad ||$
$K(a,b) := k \quad ||$
$chan(a,b) := chan(a,b) - \{m2(x,(NB(a,b),k,b))\} + \{m3(x)\} \quad ||$
$history := history + \{m3(x)\}$
END

$SendMsg4(a,b) \ \hat{=}$
ANY x, k, t WHERE
$m3(NA(a,b),k,a) \in chan(a,b)$
THEN
$aside.SendMsg4(a,b,t) \quad ||$
$chan(a,b) := chan(a,b) - \{m3(NA(a,b),k,a)\} + \{m4(t,k)\} \quad ||$
$history := history + \{m4(t,k)\}$
END

$Receive(a,b,t) \ \hat{=}$
SELECT
$m4(t,K(a,b)) \in chan(a,b)$
THEN
$bside.Receive(a,b) \quad ||$
$chan(a,b) := chan(a,b) - \{m4(t,K(a,b))\}$
END

References

[1] J.-R. Abrial and L. Mussat. Introducing dynamic constraints in B. In D. Bert, editor, *Second International B Conference*, April 1998.

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

[3] R.J.R. Back and R. Kurki-Suonio. Decentralisation of process nets with centralised control. In *2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, pages 131–142, 1983.

[4] P. Bieber and N. Boulahia-Cuppens. Formal development of authentication protocols. In *6th BCS FACS Refinement Workshop*, 1994.

[5] P. Bieber, N. Boulahia-Cuppens, T. Lehmann, and E. van Wickeren. Abstract machines for communication security. In *6th IEEE Computer Security Foundations Workshop*, 1993.

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

[7] M.J. Butler. *A CSP Approach To Action Systems*. D.Phil. Thesis, Programming Research Group, Oxford University, 1992.

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

[9] M.J. Butler. Using refinement to analyse the safety of an authentication protocol. DSSE-TR-98-8. Available from `http://www.dsse.ecs.soton.ac.uk/techreports/98-8.html`, 1998.

[10] M.J. Butler. csp2B: A practical approach to combining CSP and B. To appear in *Formal Aspects of Computing*, 2000.

[11] J. Clark and J. Jacob. A survey of authentication protocol literature. Available from `http://www-users.cs.york.ac.uk/~jac/papers/drareview.ps.gz`, 1997.

[12] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 1999.

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

[14] L. Gong, R. Needham, and R. Yahalom. Reasoning about belief in cryptographic protocols. In *IEEE Symposium on Research in Security and Privacy*, 1990.

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

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

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

[18] N. Lynch. I/O automaton models and proofs for shared-key communications systems. In *12th IEEE Computer Security Foundations Workshop*, 1999.

[19] C. Meadows. The NRL protocol analyser. *J. of Logic and Programming*, 26(2), 1996.

[20] J. K. Millen, S. C. Clark, and S. B. Freeman. The Interrogator: Protocol security analysis. *IEEE Trans. on software engineering*, SE-13(2):274–288, Feb 1987.

[21] 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.

[22] C.C. Morgan, A.K. McIver, and K. Seidel. Probabilistic predicate transformers. *ACM Transactions on Programming Languages and Systems*, 18(3):325–353, 1996.

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

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

[25] L.C. Paulson. The inductive approach to verifying cryptographic protocols. *J. Computer Security*, 6:85–128, 1998.

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

[27] 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.

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

[29] J. Sinclair. Action systems for security specifications. *J. of Computer Security*, 5:129–154, 1997.