# Proving security protocols with model checkers by data independence techniques

A.W. Roscoe and P.J. Broadfoot
Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
{Bill.Roscoe, Philippa.Broadfoot}@comlab.ox.ac.uk

July 6, 1999

**Abstract**

Model checkers such as FDR have been extremely effective in checking for, and finding, attacks on cryptographic protocols – see, for example [16, 20] and many of the papers in [7]. Their use in *proving* protocols has, on the other hand, generally been limited to showing that a given small instance, usually restricted by the finiteness of some set of resources such as keys and nonces, is free of attacks. While for specific protocols there are frequently good reasons for supposing that this will find any attack, it leaves a substantial gap in the method. The purpose of this paper is to show how techniques borrowed from data independence and related fields can be used to achieve the illusion that nodes can call upon an infinite supply of different nonces, keys, etc., even though the actual types used for these things remain finite. It is thus possible to create models of protocols in which nodes do not have to stop after a small number of runs, and to claim that a finite-state run on a model checker has proved that a given protocol is free from attacks which could be constructed in the model used. We develop our methods via a series of case studies, discovering a number of methods for restricting the number of states generated in attempted proofs, and using two distinct approaches to protocol specification.

## 1 Introduction

Cryptographic protocols frequently depend on the uniqueness, and often on the unguessability, of some of the data objects they use such as keys and nonces. If we are programming an agent running such a protocol, either for practical use or as part of a program for feeding into a model checker, then each time it creates a new nonce (say), it relies on the fact that it (and usually everyone else) has not used this particular value before. Frequently, of course, there will be no mechanism in place to guarantee this uniqueness, rather the contrary being discounted because of its extreme improbability. On the other hand, the way model checkers work means that one cannot rely on probability in this way, so if using one you have to include some mechanism for *enforcing* no repeats.

The pragmatics of running model checkers mean, unfortunately, that the sizes of types, such as that of nonces, have to be restricted to far smaller sizes than the types they represent in implementations. Usually they have to be kept down to single figures if combinatorial effects are not to make other types unmanageably large. (Objects such as messages contain several members of these simple types, meaning that their numbers grow quickly as the simple types are enlarged.)

The models that the first author and others created therefore allocated a small finite number of these values to each node that has to "invent" them during a run, so that each time a nonce (say) was required a node took one of those remaining from its initial allocation or, if there were none left, simply stopped. This use of agents with the capacity for only a finite number of runs led to several difficulties.

First and foremost, it has meant that while model checkers are rightly regarded as extremely effective tools for finding attacks on protocols, they could only be used to prove that no attack exists on the assumption that each node only engages in a very finite amount of activity. While there are often good intuitive reasons for believing that the limited check would find any attack, these are generally difficult to formalise into a component of a complete proof. Therefore it has been necessary to look to other varieties of tool, such as theorem provers (see, for example, Paulson's work [23]), for proofs once one's model checker has failed to find an attack.

Secondly, it means that questions of no loss of service (in the presence, for example, of an attacker who makes a finite but unbounded number of interventions) are difficult to address, even though the formalisms (such as CSP) used to model the protocols are usually well adapted to test such issues. For it will only take a small number of actions for an attacker to disrupt the limited number of runs that one of these cut down systems can manage. It would be far better to have a way of modelling agents that, when one run is disrupted by enemy action, can reset and start again as often as is required.

Our purpose in this paper is to show how it is frequently possible to achieve the aim of allowing agents to pick a "fresh" nonce (or other object) for each run (without limit on how many they can perform), while only actually having finitely many of them in the type running on the model checker. This is done without curtailing the ability of the intruder to generate attacks, though it may in some circumstances introduce some artificial attacks that are not really possible. Where no attack is found on such a system, it is much easier to argue that the protocol under examination is free from attacks which could be constructed within the model, than it was with the earlier class of limited-run models.

We emphasise that the work in this paper removes some, but not all, of the restrictions of previous modelling work of this sort. In particular, it cannot handle cryptographic issues other than those we allow for symbolically. It will only address issues of type confusion and time related attacks when these are explicitly handled by the protocol models (see the conclusions for further discussions).

Our seemingly impossible goal was attained by using methods derived from the subject of *data independence* (see, for example, Lazić's work [12] and Wolper's work [35]): where a program is parameterised by a data type in the sense that it passes members of the type around, but does not constrain what the type is and may only have its control-flow affected by members of the type in tightly defined ways. The ways in which the CSP models of protocols use types such as keys, nonces and agent identities fall clearly within the scope of this theory. Timestamps are more marginal because of the way in which they are compared, and for that reason are not considered in this paper. The usual result of a data independence analysis is a *threshold*: a finite size of the type which is sufficient to demonstrate correctness for all larger finite or infinite sizes. The particular nature of the protocol models (especially the nature of the intruder or spy process, and the fact that all generated values from the types are assumed to be distinct) meant that for some of these types no finite threshold is derivable using standard results. Nevertheless the same sorts of techniques used to prove data independence theorems can be employed to justify transformations being applied to a data independent type at run time. The result of these transformations is that values from type $T$ are continually shifted around, and carefully identified with each other, to create room for another value to be created that the program will treat exactly as though it were fresh.

The rest of this paper is organised as follows. Firstly we summarise the techniques used to model crypto-protocols in CSP. Next we describe the basics of the theory of data independence and discuss the extent to which the protocol models fit this pattern. In Section 4 we introduce the transformation techniques which allow unbounded runs within finite types, showing both the theory and the CSP implementation methods that have been employed to date. We then examine in turn how our methods apply to a series of well known protocols: Needham-Schroeder Public Key (NSPK), TMN (both corrected), Needham-Schroeder Symmetric Key (NSSK), Otway-Rees and Yahalom. In order to deal successfully with the larger amongst these we develop several new coding strategies for the CSP models. Finally we discuss the implications of this new method and the extent to which one can reasonably claim to have proved a protocol because a specific check has succeeded.

This paper combines a revision of an earlier paper of the same name by the first author [29] with parts of the second author's MSc dissertation [1], which took the techniques and their application further.

## 2 Protocol modelling with CSP/FDR

Protocols are traditionally described in the literature as a series of messages between the various legitimate participants. In the case of the revised NSPK protocol by Lowe [16] (in its abbreviated version without a public key server) these are

1. $A \rightarrow B : \{N_A, A\}_{pk(B)}$
2. $B \rightarrow A : \{N_B, N_A, B\}_{pk(A)}$
3. $A \rightarrow B : \{N_B\}_{pk(B)}$

The revision is the inclusion of $B$'s name in message 2. Without this, there is the now well-known attack discovered by Lowe [16]. We will concentrate on this example protocol while we introduce our techniques.

Such a description implicitly describes the role of each participant in the protocol and carries the implication that whenever an agent has, *from its point of view*, executed all the communications implied by the protocol, then it can take whatever action is enabled by the protocol. The above protocol is intended to *authenticate* $A$ and $B$ to each other and, we will assume, to allow them to enter into a session with each other. It is relatively easy to derive CSP versions of agents running this protocol from such a description: indeed Lowe's tool Casper [18] performs this task essentially automatically. For the protocol above, the code for an agent with identity `id` might be:

```
User(id) = Send(id) [] Resp(id)

Send(id) =
|~| b:diff(agents,{id}) @
  let na=???? within
  comm.id.b.(PK.(pk(b),Sq.<na,id>)) ->
   ([] nb:nonces @
    comm.b.id.(PK.(pk(id),Sq.<nb,na,b>)) ->
     comm.id.b.(PK.(pk(b),nb)) ->
```

```
                    Session(id,b,nb,na))

Resp(id) =
[] a:diff(agents,{id}) @
  [] na:nonces @
     comm.a.id.(PK.(pk(id),Sq.<na,a>)) ->
      (let nb = ???? within
       comm.id.a.(PK.(pk(a),Sq.<nb,na,id>)) ->
        comm.a.id.(PK.(pk(id),nb)) ->
                       Session(id,a,nb,na))
```

Readers not familiar with the details of machine readable CSP can find a description of the examples presented here in the appendix at the end of this paper and a more complete description at the web site quoted in the Conclusions section. As a slight simplification this code omits the option, present in all the models we actually ran in our work on this paper, of a node that is waiting for a message from another agent timing out, terminating the run, and returning to its initial state.

Note the following:

- The messages passed around in this implementation of the protocol are drawn from an abstract, constructed data type that allows us to treat operations such as encryption as symbolic. For example, the first message of the protocol is represented by the construction: `PK.(pk(b),Sq.<na,id>)` which represents the message `{na,id}` encrypted under the public key of agent `b` where `na` represents the nonce of the sender agent `id`. (More details of this type can be found below.)

- The abilities of nodes to encrypt and decrypt messages are implicit in the way they create messages and understand the messages they receive. The fact that various states will only accept messages of a given form, sometimes only when containing a specific nonce, effectively gives a way of coding in the conditions for progress in the protocol. To all intents and purposes, such entries in the protocol represent an equality test (between the nonce expected and that actually received) with progress only being possible when the test succeeds.

- Nodes do not check anything about the nonces they receive but were not originally generated by them (for example, for uniqueness).

- The mechanism for choosing a new nonce has here been left undecided, since that is the essence of this paper. In previous treatments, each node was given a (small) finite supply and had to halt when this was exhausted.

- We have left the actions of a node once it thinks it is in a session as yet unspecified. We must expect, however, that there is the possibility that the session might close and that it might then run the protocol again.

Having done this, it is usual to place as many agents as are necessary for a complete run of the protocol (in this case two; the most frequent difference to this being when some sort of server is required in addition) in parallel in the context of a network/intruder process that can do the following:

- Act as other agents, which may or may not behave in a trustworthy way.

- Overhear all messages that pass between the trustworthy agents.
- Prevent a message sent by one agent from reaching the intended recipient.
- Fake a message to any agent, purporting to be from any other, only subject to what the rules built into the crypto-system in use allow it to create based on its initial knowledge and subsequent listening.

What one usually then seeks to show is that any session that may occur between the trustworthy nodes is secure, no matter what the actions of the network from the range above. This includes showing that if either of the nodes *thinks* it has completed a run of the protocol with the other, then it really has.

The resulting network in our case is one of only three processes, though the interconnections are quite complex as shown in Figure 1 because of the multiple roles the intruder process can play and because of the different fates and sources of communications on the channels apparently between the two trustworthy nodes.

Our intruder model is to all intents and purposes the standard Dolev/Yao [6] one. However, as one can see in Figure 1, instead of having all communication between trustworthy agents go through the spy (where the spy acts as an unreliable medium), communication between trustworthy agents takes place over a different channel (namely, the channel *comm*). The use of this *comm* channel is not essential, since any safety violations found using this model would also be found without it (the intruder could replace each $comm.a.b.m$ by the pair of communications $take.a.b.m$ and $fake.a.b.m$. However, allowing this third communication channel between trustworthy agents has the following advantages:

1. Allowing direct communication between trustworthy agents does not increase the complexity of the model and actually produces simpler attacks. FDR always returns the shortest trace leading to an error. The fact that a direct communication requires only one event will tend to produce attacks with fewer intruder interventions and so conceptually simpler. If an "ordinary" communication between *Alice* and *Bob* required two actions, there would be no incentive to FDR to choose this rather than a pair of real intruder actions.
2. We are currently dealing with safety properties only. However, in the future we would like to extend this work to include liveness properties, for example, no loss of service. In that case, the use of the intruder for benign message transmission creates problems and so it is therefore preferable to have the additional communication channel *comm*.

The intruder process is conceptually simple: it can overhear or take out any message at all, and can fake any message that it knows enough to generate (which may well be through having heard the same message, even though it cannot understand the message: in other words a *replay*). In a CSP model one calculates the set of messages that can ever pass around the system, and from that the set of relevant facts and deductions. Deductions are pairs $(X, f)$ where, if the intruder knows every fact in the set $X$, it can also generate $f$. Let us consider, for example, the following data type, which is the one used for the model of the NSPK protocol in this paper.

```
datatype fact = Sq. Seq(fact) |
                PK. (fact, fact) |
                Encrypt. (fact, fact) |
                Agent. AGENT |
                Nonce. NONCE |
```

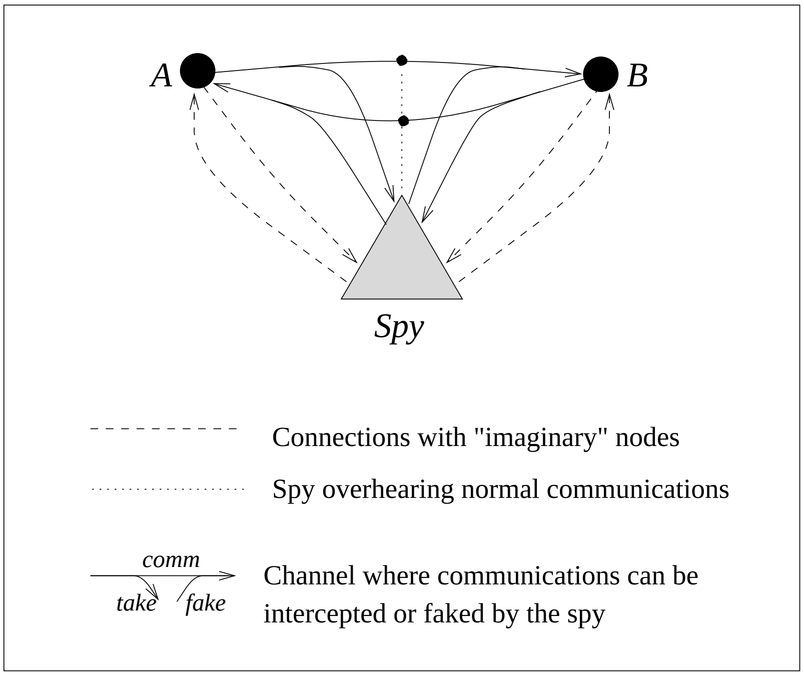


Figure 1: The network to test a simple crypto-protocol

```
          pk. AGENT | sk. AGENT |
          AtoB | BtoA | Cmessage
```

Here, `AGENT` and `NONCE` are whatever types of agent identities and nonces we are using[1], for example `AGENT` could be defined as the set `{Alice,Bob,Cameron}`; `PK` and `Encrypt` are respectively constructors representing public-key and symmetric-key encryption of the second thing by the first, and `pk` and `sk` are constructors creating the public and secret key of each agent. The constant types `AtoB`, `BtoA`, `Cmessage` are used for specification purposes as described towards the end of this section. You would then expect *at least* to have the following inferences in the intruder relevant to any set `Z` of facts:

```
deductions1(Z) = {({Sq . m}, nth(j,m)) ,
     ({nth(i,m) | i <-{0..#m-1}}, Sq . m) |
                  Sq.m <- Z, j<-{0..#m-1}}

deductions2(Z) = {({m, k}, Encrypt.(k,m)) ,
                ({Encrypt.(k,m), k}, m) |
                  Encrypt.(k,m) <- Z}

deductions3(Z) = {({m, k}, PK.(k,m)) ,
                ({PK.(k,m), dual(k)}, m) |
                  PK.(k,m) <- Z}
```

In other words, it can construct and destruct sequences, and can make up and decrypt both public-key and symmetric-key encryptions subject to possessing the right information (note the

[1] This type is somewhat more structured than that given in [28], being parameterised by these two sets, to make the data independence analysis we will see later more transparent.

difference between these last two cases). There could be more, depending on the properties of the particular encryptions in use. These could be, for example

```
deductions4(Z) = {({Encrypt.(k,m), m}, k) |
                    Encrypt.(k,m) <- Z}
```

"from a message and its encryption, deduce the key". An alternative is the following, which is a consequence of a class of known attacks on low exponent RSA under which knowing a pair or encryptions that are in a known linear (as in this case) or more general polynomial relationship with each other can lead to a intruder being able to decrypt (see [3], and [30] for a more complete CSP encoding of the resulting deductions):

```
deductions5(Z) = {({a,b,PK.(k,Sq.<a,f>),PK.(k,Sq.<b,f>)},f) |
                      PK.(k,Sq.<a,f>) <- Z, PK.(k',Sq.<b,f'>) <- Z,
                      f==f', k==k', a!=b}
```

This last one says that if we know two *different* values that the fact `f` is paired with under encryptions using the same public key, then we can deduce `f`. `deductions5` is, for reasons we will see later, particularly interesting in the context of the theory we are developing.

We will, however, in our examples, generally use only the basic set (1–3) above: in other words, we will assume that both the public- and symmetric-key algorithms are free of all but the most basic of deductions.

The intruder process is then created by applying a suitable renaming to one with the following behaviour. It is initialised so that it has whatever information `X` we deem appropriate. This will presumably include all "public" information, whatever information is necessary to allow the intruder to function as the other nodes it can play the role of, and some nonces distinct from those that the trustworthy nodes may invent for themselves.

```
Spy1(X) = learn?x -> Spy1(Close(union(X,{x})))
          [] say?x:X -> Spy1(X)
```

where `Close(X)` is a function that produces all facts derivable from the set `X` under the chosen deductive system.

In order that this intruder process can even potentially be run on FDR, it is necessary that the sets of facts over which it ranges and deductions that it uses are finite. Thus the various data types making up the messages such as agent identities, keys and nonces need to range over finite sets. In practice these have to be kept very small: no more than a few values each, if the sizes of the sets of facts and deductions are not going to grow beyond the range up to (say) 2–3,000 that can be handled on FDR reasonably. In fact, this definition of the intruder cannot be run on present versions of FDR because the compilation regime it uses would insist on computing all its approximately $2^N$ states, where $N$ is the number of facts that the parameter-sets are built from. As discussed in [28, 30], the above definition is replaced in actual runs by a more efficient representation, namely a parallel composition of one two-state process for each learnable fact with deductions being carried out by communication over a special channel. What is important to us, however, is that the resulting process is equivalent to the simple process above.

Having set up a model of the protocol like this, we have to decide what constitutes an attack, or, equivalently, what specification the system is hoped to satisfy. A tremendous amount of literature has been devoted to this point, for example [5, 17, 27]. For reasons we will explain later, for the time being we will concentrate on the extensional specifications used in [28] (so called in [27] because they examine the results that can occur from running the protocol, rather than the protocol messages themselves). In this, nodes enter a session after running a protocol and use one of the nonces generated during the protocol run as a symmetric key. We test (i) that any message Alice or Bob thinks is from the other (within a session) really is and (ii) that messages between Alice and Bob stay secret from the intruder. This is done by using special symbols **AtoB** and **BtoA** for the messages Alice and Bob send each other when they *think* they are connected in a session. We can tell something has gone wrong with *authentication* [2] ((i) above) if either receives anything other than what it ought when in a session with the other. Something has gone wrong with *secrecy* ((ii) above) if the intruder ever learns one of these symbols. Both of these errors are raised via the occurrence of special error events within the system, and so we can test for these conditions by hiding all events other than the relevant error signal:

```
assert STOP [T= System\diff(Events,{|error|})
assert STOP [T= System\diff(Events,{|spyknows|})
```

The very simple form of these two checks will be a considerable help later on. Both of these, and some other forms of correctness condition we will be meeting later on, are all *trace* checks: they do not use more complex behaviours like *failures*. The restriction to the traces model makes the data independence analysis easier, and certainly more comprehensible, and we will therefore largely restrict ourselves to this model from now on.

Our ultimate goal is to prove that our system meets these specifications however many other nodes there are and however large (including infinite) the type of nonces is. Let us denote the system in which the set of agents (including the special names `Alice` and `Bob`) is $A$, the set of all nonces is $N$, and the set of nonces initially known to the intruder (i.e., that the intruder might "invent") is $NS$ by

$$System(A, N, NS)$$

## 3 Data independence methods

A program $P$ can be said to be *data independent* in the type $T$ if it places no constraints on what $T$ is: the latter can be regarded as a *parameter* of $P$. Broadly speaking, it can input and output members of $T$, hold them as parameters, compare them for equality, and apply *polymorphic* operations to them such as tupling, list forming and their inverses. It may not apply other operations such as: functions returning members of $T$ and comparison operators such as $<$, or do things like compute the size of $T$. Data independence has been applied to a variety of notations, originating in [35]. A brief introduction to data independence in CSP can be found in [28]. For further discussion and precise definitions, see [12, 13, 14]. An application to another topic in computer security can be found in [15].

The main objective of data independence analysis is usually to discover a finite *threshold* for a verification problem parameterised by the type $T$: a size of $T$ beyond which the answer (positive or negative) to the problem will not vary. This can be done successfully for a wide range of problems, as is shown for example in [28, 15].

Several of the types used by crypto-protocol models have many of the characteristics of data independence. This is typically true of the type of nonces, types of keys that are not bound to a specific user, and may also be true of the type of agent identities. The main reason for this is that the abstract data type constructions used in the programs are polymorphic: there is no real difference in building a construction such as

[2] As we shall see later, this characterisation of authentication is fairly weak, in the sense that it takes a really disastrous breach to fail it.

```
PK. (pk. B, Sq. <Nonce. N, Agent. A>)
```

(the representation in our data type of a typical message 1) over the objects `A`, `B` and `N` from building a list or tuple.

There are, however, several features of the protocol descriptions that mean the general purpose results for computing thresholds do not give useful results. Firstly, the assumption that each nonce or key generated is distinct means that there can be no hope of finite thresholds from standard results. This is because our program must at least implicitly carry an unbounded number of values in its state so it knows what to avoid next – and the starting point for threshold calculations in the context of equality tests is the maximum number of values a process ever has to remember. Secondly, the nature of the intruder process causes difficulties because it also clearly has the ability to remember an unbounded number of values if they are available.

Since the general-purpose data independence results of the earlier papers had proved to be inapplicable, our approach was to apply some of the methods underlying the proofs of these results directly to the sort of CSP model that a protocol analysis generates. The aim was to take a "full-sized" model of a protocol (one with an unbounded number of other agents, and infinite sets of nonces, etc.) and to use these methods to reduce the problem of proving the correctness of $System(A, N, NS)$ for all parameter values to a finite check.

One of the most important methods for proving data independence theorems is setting up relations or mappings between the behaviours that two different versions of a parameterised system can display. This is an application of the theory of *logical relations* [22, 24, 25, 34]. Specifically, we can ask the question of when, if $T$ and $T'$ are two values for a type parameter, and $\phi : T \rightarrow T'$ is a function, does the function $\phi$ "lift" to map each behaviour of the parameterised process $P(T)$ to one of $P(T')$.

Given that we are concentrating on traces, this can be done with scarcely any restriction if $\phi$ is an injective function (i.e., does not map two distinct values to the same place): we always have

$$traces(P(\phi(T))) = \{\phi(t) \mid t \in traces(P(T))\} \qquad (1)$$

for a data independent program $P$ and function $\phi$ (where, on the left-hand side, $\phi$ has also been applied to the values of any constants of appropriate type within $P$, and on the right-hand side, the application of $\phi$ to a trace means its application to each member of $T$ that occurs within $t$). As soon, however, as we have a non-injective function the situation becomes more difficult, since it can change the results of whatever equality tests occur within the program. In other words, it might map an execution in which two distinct values are input and then compared for equality, into one where two equal values are input and compared for equality. There may be no relationship at all between the subsequent behaviours in these two cases, and the above equivalence becomes the possibly strict inequality

$$traces(P(\phi(T))) \subseteq \{\phi(t) \mid t \in traces(P(T))\} \qquad (2)$$

**Example 1 (Variables)** Consider the following process

$$\begin{aligned} P(T) = in?x : T \rightarrow in?y : T \rightarrow if(x = y)\ & then\ (a \rightarrow STOP) \\ & else\ (b \rightarrow STOP) \end{aligned}$$

Consider the instance of this where $T = \{0, 1\}$ and $\phi$ is a non-injective function over $T$ defined as:

$$\phi(0) = \phi(1) = 0$$

In this example, we have the following sets of traces:

- $traces(P(\phi(T))) = \{\langle\,\rangle, \langle in.0\rangle, \langle in.0, in.0\rangle, \langle in.0, in.0, a\rangle\}$
- $traces(P(T)) = \{\langle\,\rangle, \langle in.1\rangle, \langle in.0\rangle, \langle in.0, in.0\rangle, \langle in.0, in.1\rangle, \langle in.1, in.0\rangle, \langle in.1, in.1\rangle, \langle in.0, in.0, a\rangle, \langle in.1, in.1, a\rangle, \langle in.0, in.1, b\rangle, \langle in.1, in.0, b\rangle\}$
- $\{\phi(t) | t \in traces(P(T))\} = \{\langle\,\rangle, \langle in.0\rangle, \langle in.0, in.0\rangle, \langle in.0, in.0, a\rangle, \langle in.0, in.0, b\rangle\}$

This example illustrates how the equality stated in (1) no longer holds when introducing non-injective functions.

∎

In the case of variables, the inequality (2) will always hold true, because if we pick $T^\dagger \subseteq T$ such that $T^\dagger$ contains exactly one value mapping to each member of $\phi(T)$, it is straightforward to see that $traces(P(T^\dagger)) \subseteq traces(P(T))$ and that, because $\phi$ is injective on $T^\dagger$,

$$traces(P(\phi(T))) = traces(P(\phi(T^\dagger)) = \{\phi(t) \mid t \in traces(P(T^\dagger))\}$$

However, even the above inequality (2) may not hold if $P$ contains distinct constants of type $T$ . If there were two constants $c$ and $c'$ with different values in $T$ but mapping to the same thing under $\phi$, this argument would not work; for this reason, we will exclude the possibility of constants occurring in $P$ until we develop a condition below that handles them properly. What the inequality (2) above says, in essence, is that with a bigger type $T$ we may be able to exercise more of $P$'s behaviour.

**Example 2 (Constants)** Consider the following process

$$\begin{aligned} P(T, c1, c2) = in?x \to if(x = c1 \; and \; x = c2) \; & then \; (left.x \to STOP) \\ & else \; (right.x \to STOP) \end{aligned}$$

where $c1$ and $c2$ are both constants over the type $T$ of $P(T, c1, c2)$.

Consider the instance of this where $T = \{1, 2\}$, $c1 = 1$, $c2 = 2$ and $\phi$ is a non-injective function over $T$ defined as:

$$\phi(1) = \phi(2) = 0$$

For this example we have the following sets of traces:

- $traces(P(\phi(T), \phi(c1), \phi(c2))) = \{\langle\,\rangle, \langle in.0\rangle, \langle in.0, left.0\rangle\}$
- $traces(P(T)) = \{\langle\,\rangle, \langle in.1\rangle, \langle in.2\rangle, \langle in.1, right.1\rangle, \langle in.2, right.2\rangle\}$
- $\{\phi(t) | t \in traces(P(T))\} = \{\langle\,\rangle, \langle in.0\rangle, \langle in.0, right.0\rangle\}$

This example demonstrates how the inequality (2) does not hold true in the case where non-injective functions are applied to constants since in this case,

$$\langle in.0, left.0\rangle \notin \{\langle\,\rangle, \langle in.0\rangle, \langle in.0, right.0\rangle\}$$

∎

There is, however, an important exception to the rule that non-injective functions lose us the identity (1), namely the case where the result of the conditional when the equality test fails is the process `STOP`: in other words, when we can guarantee that the result of an equality test proving false will never result in the process performing a trace that it could not have performed (subject to appropriate replacements of values of type $T$) were the test to prove true. This is a form of the condition **PosConjEqT** *(Positive Conjunctions of Equality Tests)* of data independence. (It is one of a number of technical conditions derived by Lazić: see [12], for example. The form we quote here is slightly simplified but is valid in the contexts in which we use it, specifically on the right-hand sides of trace refinement checks.) In such programs without constant symbols the equivalence above is regained, and we can regard it as a "collapsing" mechanism since it allows us to compute a great deal about how a program $P(T)$ treats a large $T$ in terms of how it treats a small one.

**Example 3 (PosConjEqT condition and variables)** Consider the following process

$$\begin{aligned} P(T) = in?x : T \to in?y : T \to if(x = y) \;& then\ (a \to STOP) \\ & else\ STOP \end{aligned}$$

Process $P(T)$ obeys the **PosConjEqT** condition, since the result of its equality test proving false results in $STOP$. Consider the instance of this where $T = \{0, 1\}$ and $\phi$ is a non-injective function over $T$ defined as:

$$\phi(0) = \phi(1) = 0$$

For this example we have the following sets of traces:

- $traces(P(\phi(T))) = \{\langle\,\rangle, \langle in.0\rangle, \langle in.0, in.0\rangle, \langle in.0, in.0, a\rangle\}$
- $traces(P(T)) = \{\langle\,\rangle, \langle in.0\rangle, \langle in.1\rangle, \langle in.0, in.0\rangle, \langle in.0, in.1\rangle, \langle in.1, in.0\rangle,$ $\langle in.1, in.1\rangle, \langle in.0, in.0, a\rangle, \langle in.1, in.1, a\rangle\}$
- $\{\phi(t) | t \in traces(P(T))\} = \{\langle\,\rangle, \langle in.0\rangle, \langle in.0, in.0\rangle, \langle in.0, in.0, a\rangle\}$

This example illustrates how introducing the **PosConjEqT** condition for variables regains the equality (1).

∎

If a program with constant symbols satisfies the **PosConjEqT** property, then we need not necessarily get the equality (1), but we can guarantee the following inequality

$$\{\phi(t) \mid t \in traces(P(T))\} \subseteq traces(P(\phi(T))) \qquad (3)$$

Note that this inequality is the opposite to the inequality (2).

**Example 4 (PosConjEqT condition and constants)**

$$\begin{aligned} P(T, c1, c2) = in?x \to if(x = c1\ and\ x = c2) \;& then\ (left.x \to STOP) \\ & else\ STOP \end{aligned}$$

where $c1$ and $c2$ are both constants over the type $T$ of $P(T, c1, c2)$.

Consider the instance of this where $T = \{1, 2\}$, $c1 = 1$, $c2 = 2$ and $\phi$ is a non-injective function over $T$ defined as:

$$\phi(1) = \phi(2) = 0$$

For this example we have the following sets of traces:

- $traces(P(\phi(T), \phi(c1), \phi(c2))) = \{\langle\,\rangle, \langle in.0\rangle, \langle in.0, left.0\rangle\}$
- $traces(P(T)) = \{\langle\,\rangle, \langle in.1\rangle, \langle in.2\rangle\}$
- $\{\phi(t) | t \in traces(P(T))\} = \{\langle\,\rangle, \langle in.0\rangle\}$

This example illustrates how the inequality (3) holds true, but the equality (1) may not for processes with the **PosConjEqT** property where non-injective functions are applied to constants.

∎

We noted in the last section that some aspects of the protocol examples, such as the ways in which agent processes handle keys, nonces and identities, fit into this **PosConjEqT** framework: This is something that will be useful to us later. One exception to this is that we noticed that agent processes may well perform inequality checks with a few constants, such as their own names. Additionally, the presence of a parallel process whose alphabet and communications involve a given constant like this will generally introduce such inequality checks implicitly, as will the special-case of constants belonging to the intruder's initial knowledge. For example, the combination

$$P \underset{\{|comm.Alice|\}}{\|} Q$$

where *Alice* is a constant of the data independent type Agent, requires the equality test $x = Alice$ to determine whether the event $comm.x$ is synchronised or not. Finally, a process involving constants can legitimately depend on their mutual inequality even when it does satisfy **PosConjEqT**, and it would then be equally inappropriate to apply a function $\phi$ that identifies these values.

We thus define a condition $\mathbf{PosConjEqT}'_C$ for $C$ a set of constants: the program satisfies **PosConjEqT** except that it may have non-`STOP` results for equality tests involving at least one member of the set $C$ of constants. What we find here is that the collapsing result above holds provided $\phi$ is faithful to the values of the constants in $C$: for $x \in T$ and $c \in C$, $\phi(x) = \phi(c)$ if and only if $x = c$.

### 3.1 Positive deductive systems

The agent processes in a protocol model are fairly standard-style processes and easy to check for properties like $\mathbf{PosConjEqT}'_C$. The conditions satisfied are always natural consequences of the protocols' structure and are very easy to check visually. In addition, we will shortly have automated tools which will perform these checks. The intruder process is, however, of unusual construction, depending as it does on sets and a deductive system. As discussed briefly above, the role of constants in the initial knowledge of the intruder (such as knowing all secret keys except those of Alice and Bob) has an effect that is easy to see, but more interesting is the role of the deductive system. What turns out to be crucial here is the nature of the preconditions of deductions.

It is frequently the case that a deduction $(X, f)$ will only be able to occur when two objects of a type $T$ (possibly proper subcomponents of members of $X$) are equal, for example the deduction

```
({Encrypt. (k, m), k}, m)
```

carries an implicit equality check between the key in the encryption and the free-standing key. This is entirely within the spirit of **PosConjEqT** and has the crucial property that the identification of two keys by a function will never disable a deduction. It is easy to imagine deductive systems that do not have this property: for example if knowing 3 *distinct* encryptions of agents' names under key $k$ allowed us to deduce $k$; and the low exponent deduction on RSA (as deductions5(X)) described earlier falls into this category since it relies on the distinctness of the objects that are in a linear relationship.

Formally, we will define a deductive system to be *positive* relative to some type parameter $T$ if, for any function between types $\phi : T_1 \to T_2$, whenever $(X, f)$ is an inference the system generates for type $T_1$, then $(\phi(X), \phi(f))$ is one generated for $T_2$. What this essentially requires is that the generation of the deductions is symmetric in $T$ (i.e., treats all members of $T$ equivalently) and never has an inequality requirement over members of $T$ between places they appear on the left-hand side of a deduction.

The standard deductive system described earlier has this property, as do many variants. However, as the counter-example above and `deductions5` show, it is something that one has to be careful of.

The important property we can now state is this: if we build an intruder over a positive deductive system, and its initial knowledge set contains no inequality tests (explicit or implicit) with members of our type $T$ other than the constants $C$, then the resulting process (and hence, subject to obvious conditions on the other parts of the network, the entire protocol model) satisfies $\mathbf{PosConjEqT}'_C$.

### 3.2 A simple application: agent names

Almost all protocol models for model checkers that one sees make the simplification of allowing the intruder only a single identity. In other words, the intruder's role in acting as all the agents in the system other than our trusted Alice and Bob is reduced to giving it a single name to play with in this regard. This is usually justified by the claim that any attack in which the intruder used multiple identities would work equally well if all the other identities were reduced to a single one. What we can now do is prove this assertion as a consequence of the theory we have been developing under a few natural conditions.

**Proposition 1** Suppose $System(A, N, NS)$ represents the CSP model we would obtain if we modelled a given protocol using agent type $A$, in which all the abilities of agents other than those in some proper subset $C$ of $A$ (typically $\{Alice, Bob\}$) are given to the intruder, where $NS$ are nonces known to the intruder including all those created by nodes outside $C$. Suppose further that

- All processes are data independent in the type of agent names.
- The agent processes representing $C$ each satisfy $\mathbf{PosConjEqT}'_C$.
- The intruder process satisfies $\mathbf{PosConjEqT}'_C$ in the sense discussed in the last section (through having a positive deductive system).
- The specification we are trying to prove relates only to nodes' communications involving the names in $C$. In other words it can – as in our example – be decided by looking at the

traces of $System(A, N, NS)\backslash X$ where $X$ includes every event[3] mentioning an agent name outside $C$.

- Further processes such as servers satisfy $\mathbf{PosConjEqT}'_C$.

Then if $System(C \cup \{Cameron\})$ (where $Cameron$ is any name outside $C$) satisfies the specification $S$, then $System(A)$ satisfies $S$ for all $A \supseteq C$.

**Proof** Under these circumstances, we know that for any $A$, we can define a collapsing function $\phi$ that maps every name other than those in $C$ to a fixed name (say *Cameron*) and that this would simply produce the system

$$System(C \cup \{Cameron\}, N, NS)$$

and, since the application of $\phi$ to any trace $t$ produced by hiding the above $X$ is just $t$ (for $\phi$ leaves the only remaining names in $t$, if any, alone) we get

$$\begin{aligned} &traces(System(A, N, NS)\backslash X) \ = \\ &traces(System(C \cup \{Cameron\}, N, NS)\backslash \phi(X)) \end{aligned}$$

so plainly the big system satisfies our specification if and only if the small one does. ∎

What we have done here is to derive criteria under which a standard informal argument can be made rigorous.

Note that the particular specifications we are using in our NSPK example do not mention any members of $A$ or any other data independent types: error conditions are caught within the process itself and flagged via fixed events. This simplifies proofs using collapsing functions, for if $P(T)$ is any process in which all events involving members of $T$ are hidden, then, if the collapsing identity holds for $\phi$,

$$traces(P(T)) = traces(P(\phi(T)))$$

and they obviously satisfy the same specifications. The same is essentially true under the condition given above that the specification only mentions names in $T$ that are the values of the constants that our collapsing function must keep separate, all other values of $P(T)$ being hidden. As soon as one starts to include behaviour involving general members of $T$ explicitly in the specification, life becomes more complex.

Fortunately it seems to be rare for other styles of specifications about sessions between agents $A$ and $B$ to constrain events mentioning other agents, so the above argument applies widely.

## 4 Towards a general proof

In our example (NSPK), we have shown that it is sufficient to deal with the case of two-plus-one agent names, but there is still the problem of the infinite type of nonces that is required. All parts of our program, apart from the mechanism that generates nonces for Alice and Bob (whom we are assuming always use entirely fresh ones), satisfy **PosConjEqT** in the type of nonces, but that mechanism assuredly does not. It is not hard to argue, however, using similar methods to the last section, that we can assume that the intruder initially has only one nonce $n$ in its knowledge, and that $n$ is never generated for the two trustworthy users by our nonce-creation function.

[3]This condition can be relaxed a little in the more general context of data independence arguments, but the extra complications seem unnecessary here.

Noticing that each time a node generates a nonce, it immediately communicates it to another user, it is possible to re-cast the original descriptions of reliable users so that the choice of which nonce they create is delegated to an external nonce manager process $NM$: you can think of this as an artifact of our modelling technique, in much the same way that the intruder is. Every time a process wants to create a nonce with an output communication, we turn the communication into an input of the nonce and force the communication to synchronise with $NM$ (in addition to all the other processes it is synchronising with).

If we then program $NM$ so that it remembers all the nonces it has given out and never issues the same one twice or the one belonging to the intruder at all, we have implemented our assumptions about how nonces are selected. Aside from this process, our example system satisfies **PosConjEqT** on the type $N$ of nonces, but plainly the complete system does not.

As our system runs, the trustworthy nodes hold a small finite number of nonces at any one time (in the case of NSPK[4], at most 4), but the other two processes (intruder and the nonce manager) both hold an unbounded number. $NM$ remembers all the ones seen so far so that it doesn't repeat itself, and the intruder remembers everything it has seen (including objects that together contain an arbitrary number of nonces) so that this can potentially be exploited in future attacks. Notice that what really matters about $NM$ is that it always hands out a nonce unknown to any of the other processes.

It may seem like a strange thing to do, but there is no reason why we should not apply a transformation function $\phi$ to a data independent program in the middle of a run. Suppose $P$ is such a program, satisfying **PosConjEqT**. Any state $P'$ that $P$ may have reached during its execution (i.e., $P'$ is a state reached in its operational semantics, namely a program that represents how $P$ behaves after some sequence of actions) will still be a data independent program, though it could well have acquired some values from $T$ that it holds in its identifiers which were not present in its initial state.[5] For a given assignment to these identifiers, we will always have

$$traces(\phi(P')) \supseteq \{\phi(t) \mid t \in traces(P')\}$$

because of the **PosConjEqT** property, but the inequality may be strict because some values held in identifiers in $P'$ that are distinct may get mapped to the same place by $\phi$, resulting in an equality test in $\phi(P')$ giving the answer *true* where it gave *false* in $P'$. The values variables hold at the point $\phi$ is applied behave like the constants in Example 4 above.

What we are going to do is to apply transformations that identify some of the old nonces, no longer known to Alice and Bob, but remembered by the intruder. This does not directly affect the states of the trustworthy node processes, but can potentially increase the traces of the intruder because of the above inequality, and might therefore lead to the resulting process failing to satisfy a trace specification though it did before. We will see an example of this later. What it cannot do, however, is lead to it satisfying a trace specification not mentioning $T$ that it did before. Hence if the transformed $P'$ meets such a specification, then so did the untransformed one.

These transformations can be carried out whenever we wish during the execution of our program, and in practice have to be programmed into the intruder (whose memory gets transformed each time) and $NM$, which can itself now re-use the values that the intruder has "forgotten" through the transformation. If we make sure that the number of distinct values remembered by the intruder is bounded, by applying transformations whenever it gets too large, then we can get away with having only a finite set of nonces. We have found a way of using the same ones as "fresh" over and over again!

[4]For this bound we are assuming that neither node is active in more than one protocol run at a time: this issue is discussed a little more in the Conclusions.

[5]For example, the process $in?x \to in?y \to P(x, y)$ has acquired values $x$ and $y$ after a trace of length 2.

This provides a "fail-safe" method for attempting to prove traces properties of protocols using finite types. The proof of the following proposition is essentially as argued above.

**Proposition 2** Suppose one of our protocol models has (i) a positive deductive system, (ii) the trustworthy processes satisfy **PosConjEqT** in the type $N$ and (iii) the trace specification $S$ does not explicitly constrain $N$. (So the specification is judged by the check $S \sqsubseteq_T P \backslash X$ where $X$ includes all the communications involving $N$.) Let $P^\dagger$ be a model of the protocol with the addition of events *collapse*.$\phi$ which, when executed, apply $\phi$ to all values of type $N$ held in state. Further suppose that $P^\dagger$ uses a finite type $N$ but is always able to perform some proper collapse when it reaches a state where the whole of $N$ is in use. Then

$$S \sqsubseteq P^\dagger \backslash X \;\Rightarrow\; S \sqsubseteq P \backslash X$$

where $P$ is the same protocol model without collapses but with arbitrary (including infinite) $N$.

∎

Fail-safe here means that it will guarantee to find an attack if there is one, but may still fail even when there is no attack because a transformation has enabled the intruder to perform an inference it could not have performed otherwise. Whether or not such false attacks are thrown up is heavily affected by the strategy that is used for generating the transformations applied during execution: that is what we now discuss.

From an implementation and clarity point of view, we have found it useful to divide the type $N$ into two parts which we might term *foreground* $F$ and *background* $B$. The foreground values are the ones supplied to the trustworthy nodes when they request a fresh value, and the background values contain all those initially known to the intruder and those to which redundant foreground values are mapped. The background values will be accepted by Alice and Bob as valid nonces, but will never be generated by them as their own. When the intruder needs to use a nonce in a message it is creating then it can use any background or foreground value it knows; so as not to restrict its behaviour unreasonably you should make sure that it always knows at least one background nonce (and one member of each other type being given similar treatment).

Provided the manager process is kept accurately up to date with which foreground values are presently known to Alice and Bob, the size of $F$ needs to be at least the largest number of nonces that (a) were generated by one of these two nodes and (b) have continuously been known to at least one of them ever since. In our example protocol, this is evidently no more than 4, and is in fact[6] 3. Using more foreground values than are needed will not given any more stringent a check of the protocol and will usually – though not invariably, owing to the effects of symmetry reductions – slow the check significantly.

The more background values there are, the more flexibility there will be in deciding where to map each redundant member of $F$. In designing the strategy for determining where to map each one, one should be conscious of the desirability of enabling as few spurious deductions as possible. The following principles seem to the authors to be desirable, the first of them indispensable.

(a) If the type under consideration ever gets used as keys, one must never – unless, perhaps examining the effects of key compromise – identify one that the intruder presently does know with one that it does not[7]. For that would give it the immediate "ability" to decrypt all the messages it may hold under the unknown key.

[6] If you do attempt to use a smaller-than-obvious number like this it is important that the model you use generates an error flag if the supply does not, in the event, prove adequate. This is done in our implementations.

[7] In some protocols, especially ones with servers that perform some translation function, one has to be careful in deciding how to discriminate between values that are known or, in some subtle sense, *knowable*, and other ones.

(b) It might cause problems if a redundant member of $F$ is mapped to a member of $B$ that is currently meaningful to a trustworthy node, for this might give the intruder extra messages it can use in its dealings with that node.

In the case of our example, precaution (b) is not necessary: no new attacks are created by ignoring it, though the authors expect that there will be protocols where it is necessary. Precaution (a) is not necessary for the authentication specification, which still succeeds if it is ignored. It is, however, necessary for the secrecy specification, because of the way we have used the nonces generated during runs as session keys: if Alice runs a session with 'Cameron' (the intruder) and then one with Bob, she will actually use separate nonces for these runs. But if our mappings send them both to the same value when they become redundant, then the intruder will become 'able' to decrypt the messages that passed between Alice and Bob during their session because it legitimately knew the nonces that appeared during its own session with Alice.

There is one potential problem with precaution (a), namely that the intruder might learn, through inferences, all of the values in $B$ that are set aside as targets for mapping unknown redundant values. It is wise, therefore, to put an error-trap into one's model to detect this situation, in the same way that we guarded above against running out of members of $F$. In our example, however, this does not occur and it is possible to get both specifications to run successfully with the three members of $F$ already mentioned, and two members of $B$: one initially known to the intruder (and the target of all redundant values that are) and also one that is not (and fortunately never becomes known during the run). This proves, therefore, when combined with the preceding analysis, that $System(A, N, NS)$ satisfies both these specifications however large the three types become.

The authors suspect that, provided the two precautions above are followed in the nonce reduction strategy, it will be very rare to find false attacks against specifications of the type described in this section, but there is no sense in which they are claimed to be complete. We conjecture that in any case where the only factor making the type of facts relevant to the intruder infinite is the type $N$ we are manipulating, then it is possible, for large enough finite $B$, to create a strategy that is guaranteed not to introduce false attacks. Such a result would be interesting theoretically in the sense that it would imply *decidability*, but the size of $B$ generated would probably be so large that the resultant check would be impractical on FDR.

# 5 Implementation considerations

When implementing the very active nonce management regime discussed in the last section it is necessary to include enough communications from Alice and Bob to $NM$ so that the latter knows what nonces are meaningful to them. Such messages are, of course, an artifice of the model with no analogue in the real world. The process $NM$ then issues 'fresh' nonces to Alice and Bob as discussed above, and issues commands to the intruder to map redundant nonces in $F$ to nonces in $B$. To implement precaution (a) above, it is necessary that $NM$ can enquire of the intruder whether or not it already knows a particular nonce, so that it can map it to an appropriate place.

So the state of the process $NM$ has to contain information on how many times each nonce is presently relevant to (each of) Alice and Bob (bearing in mind that they may be persuaded to give a nonce more than one role in a protocol). As necessary, it generates mappings of nonces in $F$ that are not relevant to Alice or Bob, to whatever member of $B$ its strategy dictates.

The intruder process has to be modified so that it can tell $NM$ whether it knows a particular value in $B$, and more significantly has to become able to implement the mappings on its memory requested by $NM$. This is trivial in the case of the abstract `Spy1(X)` process described earlier, but requires a little more ingenuity in the case of the 'lazy' intruder built out of many parallel components. What happens is that, to map nonce $n_1 \in F$ to $n_2 \in B$, components of the lazy intruder that know something involving $n_1$ are commanded to transfer their knowledge to the corresponding component involving $n_2$, before forgetting what they knew.

If $NM$ has several values it can hand out when asked for a fresh nonce, then clearly it does not matter which it gives, because all are treated completely symmetrically by the rest of the network. Therefore (i) there is no point in investigating the effects of handing out different options and (ii) there may be scope for choosing *which* value to hand out with a view to cutting down the overall range of states visited. The concept of symmetry reductions over states spaces is well known to be related to data independence and is discussed, for example, in [8, 9, 11].

The result of all this effort is a model which has roughly the same number of nonces as the protocol models we were accustomed to running previously, though the additional complexity of the intruder, the extra process $NM$, and the fact that the nonces get used in more permutations than had previously been the case, mean that it has both somewhat more and larger (and so slower to run) states than earlier models (see, for example, [16] and [17]). FDR can check the resulting file in a few minutes on a 266MHz Pentium II laptop, though the size and complexity of the states (which incorporate the large intruder processes and the effects of the `chase` compression) mean that this and our other protocol examples are significantly slower (in states per second) and larger (in space per state) than most FDR models: perhaps one order of magnitude in space and two in time, because as well as the increased number of states, checks will take longer due to the increase in complexity of the intruder process.

# 6 General specifications

In the introductory treatment above we deliberately chose a specification style that allows us to hide (in the implementation) all the values that are being manipulated by collapsing functions. If $P$ is such an implementation, with the hiding, but without collapsing and therefore defined over infinite types where an unbounded supply of distinct values is required, and $P^\dagger$ is a version incorporating collapsing, the assumptions and results we have derived so far establish

$$traces(P) \subseteq traces(P^\dagger) \setminus \{|collapse|\}$$

or in other words $P$ is a trace refinement of $P^\dagger$ once the extra events are removed. We have therefore been able to localise the gross effects of the collapsing entirely within the implementation: proving a trace specification of $P^\dagger$ automatically proves the same specification of $P$ (the process we are really interested in but which is unmodellable because it is infinite state).

This has advantages in both theoretical simplicity and (at least relative!) clarity. It does, unfortunately, exclude just about every other style of specification that is used with FDR for security protocols. For other forms of specification tend to constrain the relationships between the protocol messages seen by the two participants in a way that requires us to leave some or all of the nonces, keys, etc. visible.

Two such styles are described below:

- The *canonical intensional specification* of [27] states that no agent ever believes it has completed a protocol run with another (both being trustworthy) unless the sequence of sends and receives implied by the protocol description has actually occurred as a subsequence of the preceding trace,
  - in the right order (i.e., Alice sent message 1, then Bob received it, then he sent message 2, etc.),

- with all the values in these messages being appropriate (e.g., the nonce $N_A$ received by Bob is the one sent by Alice),
- with separate runs being disjoint (e.g., if Bob thinks he has performed two separate runs, then Alice must too),

until the given agent's last message in the protocol.

- The specifications used by Lowe in [17] and Casper take the form of getting agents running a protocol to send a special message saying this, and then a second message when they have finished. These messages carry the data values of the protocol, so we can specify (for example) that if Alice thinks she has completed a protocol run with Bob with values $(N_A, N_B)$, then Bob has said he is running the protocol with Alice with these same values.

In either of these cases, it is seemingly necessary to have values of the types we are collapsing visible in the processes on the right-hand sides of refinement checks, and to constrain how they behave on the left-hand sides. It is clear that the left and right hand sides of a check then have to be coordinated about the effects of collapsing: we can achieve this by including the collapses in the traces of both sides. Even if we do this, it is necessary to be very careful: throughout our work so far we have been lucky that collapses tend to increase trace sets, since on the implementation side this results in false negatives (i.e., false attacks) rather than false positives. However, on the specification side this same effect works against us.

**Example 5** Consider the following pair of processes

$$
\begin{array}{rcl}
P_1 & = & a?x \to a?y \to b!x \to STOP \\
P_2 & = & a?x \to a?y \to b!y \to STOP
\end{array}
$$

Plainly $P_1$ does not, in general, refine $P_2$. If we allowed an on-the-fly collapse of the data type after the two inputs, there is the danger that the two input values might be identified resulting in a false positive answer.

∎

There are two dangers: the first is that through a collapse the traces of the specification itself might be increased (by changing the answer to an equality test). The second, and the one displayed by the example above, is that a collapse might result in equating a value held in an identifier in the specification with one in the implementation. Either could lead to a false positive in a refinement check.

To avoid these twin dangers it seems to be necessary to be rather more draconian – and hands-on – in how we handle a specification under an on-the-fly collapse than was the case with implementations. Suppose we have a specification process $S$ which applies to a system $P$ without collapses, and we want to create, without the danger of false positives, a version $S^\dagger$ which applies to $P^\dagger$ in which each collapse that occurs is displayed in the trace.

Unless we have some knowledge of how collapses occur that allows us to restrict where this is done, we need to augment each state of $S$ with the possibility that any collapse $\phi$ might occur, and decide what state the specification now moves to. So suppose $S_n(X)$ is such a state, where $X$ represents all the parameters acquired by this point in the run. The corresponding state $S_n^\dagger(X)$ of $S^\dagger$ will have all the actions of $S_n(X)$ plus, for each collapse $\phi$, the action $collapse.\phi$ leading to a state $S_m^\dagger(X')$ which must have the following properties:

(a) $traces(S_m^\dagger(X')) \subseteq \phi(traces(S_n^\dagger(X))$

(b) $X'$ contains no value in $T$ which is the image under $\phi$ of more than one thing.

**Proposition 3** Suppose $P$ and $P^\dagger$ are defined as in Proposition 2, and $S$, $S^\dagger$ are defined as above. Then

$$S^\dagger \sqsubseteq_T P^\dagger \;\Rightarrow\; S \sqsubseteq_T P$$

**Proof** This is an induction on the length of the execution sequence (visible and invisible actions) of $P$. At each stage either $P^\dagger$ can perform an analogous action or a collapse, followed by an analogous action, and the effect of the collapse is to make $S^\dagger$ (if anything) more stringent and $P^\dagger$ (if anything) worse. Condition (b) above ensures that any equalities which exist after the collapse between values in the states of $S^\dagger$ and $P^\dagger$ also held before. ∎

Restriction (b) on the construction of $S^\dagger$ is, of course, there to prevent the problem illustrated by $P_1$ and $P_2$ above. It may in many applications be an uncomfortably strong condition, but fortunately it matches the way we are modelling protocols well in most cases. To illustrate this, we consider a pair of specifications that are mid-way in spirit between the two examples quoted above.

Suppose Alice has (apparently) initiated a session of the modified NSPK protocol with Bob. Each (it is reasonable to assume) believes the protocol to have been completed when it has performed its own last action. In the case of Alice, this is sending message 3 and, in the case of Bob, it is receiving (and accepting) the same message. It seems reasonable to specify that

(i) Bob never accepts message 3 apparently from Alice, unless Alice has actually sent the same message to him, and

(ii) Alice never sends message 3 to Bob, unless he has sent her a consistent message 2 (i.e., one with the same $N_B$),

(iii) and furthermore, each of these is on a one-to-one basis (so neither thinks it is in more sessions with the other than can reasonably be justified).

We can specify the two directions of this separately using similar specifications. (i) (with (iii)) above, the authentication of initiator Alice to responder Bob, is captured by the specification `S_BA({})`, where

```
S_BA(X) = take3AB?Nb -> S_BA(union(X,{Nb}))
       [] fake3AB?Nb:X -> S_BA(diff(X,{Nb}))
```

provided the implementation is renamed as follows:

- The actions of the intruder *taking* the message 3 containing some nonce Nb sent by Alice and intended for Bob are mapped to the event `take3AB.Nb`, and
- The actions of the intruder *faking* these intercepted messages containing some nonce `Nb` to Bob from Alice are mapped to the event `fake3AB?Nb:X`, where `X` is the set of nonces which are present in the messages 3 sent by Alice to Bob and taken by the intruder.

This specification simply says that each acceptance of a "fake" of the given type is preceded by a "take" of exactly the same message, so that the intruder has merely acted like a faithful (and non-duplicating) messenger. (We do not have to worry about direct communications of this message from Alice to Bob on channel `comm`, as each message 3 accepted on this has by construction been appropriately sent.)

S_BA does not take account of collapses, and would not be an appropriate specification to use of the protocol implementation that incorporates them. After all, we would not want the "justification" of a given receipt of message 3 to be the fact that one has previously been sent with the same actual value Nb, but where an alphabet transformation involving this value had occurred in between. We can form a version that does take account of collapses by following the above rules. Instead of the event $collapse.\phi$, we have replace.(n1,n2) meaning that n1 gets mapped to n2 and all others to themselves. (This channel replace is what implements collapses in our protocol messages, carrying commands from the nonce manager to the intruder.)

```
SC_BA(X) = take3AB?Nb -> SC_BA(union(X,{Nb}))
        [] fake3AB?Nb:X -> SC_BA(diff(X,{Nb}))
        [] replace?(n1,n2) -> SC_BA(diff(X,{n1,n2}))
```

The set difference in the last line enforces condition (b) in the rules for the creation of $S^\dagger$ and therefore prevents false positive results of the type seen in Example 5.

The protocol model used with the original specification can now be modified by the removal of the Session phase of the agent programs (on completing a protocol run, they simply perform the appropriate close event and start again). It checks successfully against the specification SC_BA({}). With the symmetric specification SC_AB({}) that Bob in the role of responder is authenticated to Alice as initiator (i.e., (ii) above), we have to be a little careful about the mapping strategy of nonces, because of the possibility that Bob might give up on a protocol run between sending a message 2 and Alice receiving it. If this happens, then there is a moment when neither Alice nor Bob knows $N_B$. If a collapse then occurs which identifies $N_B$ with something else, the transformation of the specification analogous to the above creates a false "attack". We avoided this by mapping $N_B$ to a value which is itself mapped elsewhere.

We can thus conclude that the protocol model defined over an infinite set of nonces, and without collapsing, satisfies both of these specifications that examine the final messages of protocol runs, as well as the ones given earlier in terms of the Session phase.

Hopefully the above constructions are clear enough to justify the requirements of the transformed specification $S^\dagger$ on an intuitive basis. An important part of the formal proof was the use of a *testing process*: any trace or failures refinement check $Spec \sqsubseteq Impl$ can be shown equivalent to one of the form

$$T \sqsubseteq (\widehat{Spec} \underset{A}{\|} Impl) \setminus A$$

where $T$ is very simple and independent of $Spec$ and $Impl$ (in the case of traces checks it is $STOP$), $\widehat{Spec}$ is a testing process that runs in parallel with $Impl$ and generates an easily detected error if it does something contrary to $Spec$, and $A$ includes all events of $Impl$ and $Spec$. The virtue from our point of view of this transformation was that once again the specification ($T$) ignored the data independent type. The use of testing processes is explained further in [31]. The requirements which data independence arguments and theories place on implementations are generally much weaker than those placed on the specifications. Thus having the testing process $\widehat{Spec}$ on the "implementation" side of the check makes the analysis of the collapsing significantly easier than leaving it on the left hand side. We have in fact found practical advantages in using this transformed formulation of the correctness condition on FDR: including the testing agent as part of the "implementation" as this does means that it can be incorporated into compressions used to reduce the size of the state space.

In the rest of this paper, we study the use of the two styles of specification seen to date – the Session based ones and this last-message authentication – on a variety of protocols. Before we do this, we will discuss briefly how the other styles of protocol specification mentioned above –

the canonical intensional, and Lowe-style, specifications with signal events – might work within our framework.

As far as we know, neither creates any difficulties in the principles of how to prove a specification via on-the-fly collapsing that we have not already solved. However, in each case we think there may be penalties to pay in terms of running speed of the checks that are more severe than the corresponding runs without the collapsing machinery to attempt a proof.

The sizes of the data types used with the collapsing methods (in the example above, the type of nonces) are somewhat larger than in most earlier FDR protocol models. This presents a particular problem in the first case above, of the canonical intensional specification as presented in [27], because the time taken to normalise these particular specifications (something FDR does to the left-hand side of every refinement check) grows very quickly with the size of the alphabet. It ought to be possible to reduce or eliminate this problem by strengthening the specification in a case-by-case way, but this would seemingly lose the ease in which this specification is generated just about automatically from the protocol description. This is a topic for further work.

In the second case, we may suffer simply because the insertion of extra "signal" messages has a significant effect on the state-spaces of the protocol implementation models. It is comparatively rare for the implementation state-space size to be the limiting feature on protocol checks without the additional features we have developed for proof, but much more common with these features – so the extra states this method generates become less of an affordable luxury. It is the very action of sending or receiving a protocol message that triggers Lowe's agent models to send signal events; treating these causal events as the signals, as the style of specification we have used above does, allows us to create equivalent tests of protocols without the increased number of states.

The exception to this is when the protocol messages do not contain all the information we want to signal. The example above displays this potential problem, as we might reasonably want not only to ensure that our two nodes can authenticate each others' presence in a session with them and agree on the nonce $N_B$ (which the specifications `S_AB` and `S_BA` achieve), but also agree[8] on $N_A$. Since this is not contained in message 3, there is no way that specifications in which instances of message 3 can occur can achieve the latter. If we know that a given event $e$ of the protocol would cause, in Lowe's model, a signal whose information content is not contained in $e$, then it is more efficient to add this information to $e$ *in a way that does not affect how the protocol runs, or how the intruder behaves.* In our protocol this would mean adding $N_A$ as a field that both sender and receiver signal to the environment at the same time they either send or receive message 3, but these extra fields are not synchronised nor noticed by the intruder. In recent work with Lowe, we have modified his tool Casper to adopt this approach, generally gaining substantial efficiency improvements.

# 7 The TMN protocol: handling servers and staleness

Originally proposed as a protocol for use in mobile telecommunications, TMN (Tatebayashi-Matsuzaki-Newman) [33] is a favourite protocol for running protocol analysis tools on because there are so many attacks against it! Obviously there is no point in attempting to prove the original version correct, so we take as our starting point the final version of [20] which is claimed to have reasonable properties. The objective of the protocol is to establish a session key between Alice and Bob; in doing this, they communicate only via a central *server* process. (How they

[8]Agreement on $N_A$ may or may not seem an academic nicety in the case of this protocol, but it would be obviously essential in the case of a data item of importance outside the protocol such as an amount of money being transferred or a session key to be used after the run.

choose to communicate later using the key is no concern of ours.)

For two agents to establish a secure session with each other, the protocol performs the following steps:

MESSAGE 1. $A \rightarrow S$ : $\{B, sec_A, k_A\}_{pk(S)}$
MESSAGE 2. $S \rightarrow B$ : $A$
MESSAGE 3. $B \rightarrow S$ : $\{A, sec_B, k_B\}_{pk(S)}$
MESSAGE 4. $S \rightarrow A$ : $V(k_A, k_B)$

where $A$ is the initiator agent, $B$ is the responder agent, $S$ is the server, $sec_A$ and $sec_B$ are the secrets shared with $S$ belonging to agents $A$ and $B$ respectively, to authenticate their messages to $S$, and $k_A$ and $k_B$ are the symmetric keys created by $A$ and $B$ respectively. Messages 1 and 3, which have essentially the same content, have different formats to disambiguate them.

There are two encryption methods used in this protocol, namely public key encryption and Vernam encryption. All agents know the server's public key $pk(S)$, and it is assumed that the corresponding secret key is known to the server only.

The function of the Vernam encryption in message 4 is to ensure that only the initiator is able to decrypt the responder's key, since it is hoped that, apart from the server, only the initiator will know its own key specified in the Vernam encryption.

Our initial implementation of this protocol followed the methods described in the previous section closely, and concentrated on the extensional, `Session`-based specification. There was once again a single type requiring the use of on-the-fly collapsing, namely that of keys. The main addition to the network is the server process, which evidently can hold several of these keys. Initially we assumed that there was only a single, sequential server even though that was probably not realistic (for otherwise one pair of agents in the middle of a run could hold up all others).

We found it necessary to give the key manager process five foreground keys and the two background ones used under a similar principle to that described for NSPK (one as target for keys the intruder has sufficient information to deduce, and one for others).

Unfortunately, this led to a model for which the attempted check did not terminate. We believe it had many more states than the approximately 5 million we ran it for. This led us to re-examine the check and discover two quite separate strategies for cutting down this explosion.

1. The server is not responsible for generating *fresh* keys or any other new values which are unknown to any other process in the system. All the messages sent by $S$ are built either from information known publicly to all participating processes (for example, agents' names), or from data values received in previous messages from agents. During a complete run of the protocol, the server will receive two keys, namely, a key $k_A$ from the initiator agent, and a key $k_B$ from the responder agent. The server initiates and sends the following 2 messages:

    - Message 2: is simply built up from public knowledge (i.e. agents' names) as a result of a corresponding message 1.
    - Message 4: is built up from the keys: $k_A$ received in message 1 by the initiating agent and $k_B$ received in message 3 by the responder agent.

    Rather than implement the server as a separate process, we can incorporate its functionality into our intruder process. We have already observed that our server generates messages based on information in previous corresponding messages only. We also know that our intruder can create and deduce messages by means of a set of defined deductions. Therefore, we can extend our intruder's functionality to include that of the server, by adding the following appropriate deduction function:

```
deductionsS =
Union({
   {({pkserver(Sq.<a, k, sc(b)>),pkserver(Sq.<k', b, sc(a)>)},
        vernam(k, k')) | k <- keys, k' <- keys, a <- agents,
                         b <- agents}
      })
```

This deduction function can be interpreted as follows: If the intruder knows a message 1 (`pkserver(Sq.<a, k, sc(b)>)`) and a message 3 (`pkserver(Sq.<k', b, sc(a)>)`), then it can deduce a message 4 (`vernam(k, k')`) which corresponds to the message 1 and message 3 already known. Since the intruder is able to overhear all messages, we can safely have the creation of the required message 4 to be dependent on the intruder knowing the message 1 and message 3.

It may seem odd to include a trustworthy party into the intruder in this way. However the powers we have given our intruder to overhear, take and fake messages mean that ordinary nodes' interactions with the server are in any case so much within its control that we might as well regard the server as being its slave. The only thing that this particular server enables the intruder to do that it could not do before is to convert a (message 1, message 3) pair into the corresponding message 4, and it is this that the above additional deduction achieves.

An immediate advantage we gain from this is that the sequentiality of the server process disappears: the above deduction allows the intruder to generate any behaviour it could engineer with an arbitrary number of the original servers operating in parallel.

The idea of removing the server process, and incorporating its functionality into the intruder, not only reduced the number of processes running in the system by one, but just as importantly reduces the minimum number of foreground keys from 5 to 3. The state saving is created by a combination of this effect and the removal of some irrelevant interleavings of actions.

There are arguments for doing the same thing to any party in a protocol such that we are neither specifying anything about its state of mind nor the sequence of messages it performs: both of these things are lost by incorporation into the deductive system of the intruder. One possible application of this idea will be discussed in the Conclusions.

2. We know that the foreground keys get mapped to background keys in the intruder's memory, to provide the illusion that our Key Manager is able to generate an infinite supply of *fresh* keys. This means that for all defined messages in the protocol model, the intruder will eventually know them with the appropriate key transformations applied. By messages, we mean the 4 complete messages' structure defining the TMN protocol (as previously defined). Since the intruder will eventually have all these *stale* messages containing background keys in his knowledge set anyway, we could place them in the intruder's initial set of knowledge. By doing this, we are not allowing the intruder to perform any deductions or attacks which it could not have performed before, since the intruder would have reached a state consisting of this knowledge eventually, and therefore would be able to perform the same actions. However, by giving the intruder this initial knowledge, we are eliminating a number of

possible states which the intruder can be in. More precisely, we are eliminating the states which do not yet know all these stale messages in the intruder's knowledge.

This is a safe transformation in the sense that giving the intruder more knowledge will never remove an attack (as it can only increase the global set of traces). As with other things we have done, the worst thing that can happen is that we might introduce a false attack, but even that should be impossible if the set of stale messages is accurately calculated.

This opens up the possibility of deliberately giving the intruder other extra knowledge that, it is believed, should not create attacks, with the sole purpose of cutting down the state space. This is not something we have tried.

Each of these two transformations individually brought the system down to about the limit of what could be checked on FDR using the machines at our disposal (4–10M states). Combined, they made it very small and quick (about 15,000 states).

*While we developed these strategies for use in the CSP scripts used for data independent proofs of protocols, there is no reason why they could not be used in other CSP models. The server transformation should still have a significant effect on state space (though less spectacular because it would no longer directly enable a cut in type size) and the incorporation of stale information would enable related attacks to be found much faster.*

This implementation was checked for authentication and confidentiality properties by the refinement checks stated in the previous results section. We found that this TMN protocol model does satisfy both the `Session`-based extensional specifications.

These effectively say that by the time a session is up and running, with either party understanding the messages it is receiving using the key generated in the run, then we have the obvious properties of confidentiality and authentication of who we are talking to.

Both [20] and [27] note that this protocol does not have stronger authentication properties of the general type studied in Section 6. Plainly the responder has no assurance, from the protocol run alone, that the initiator was even present. Anyone could have sent him message 2. The initiator would have slightly stronger guarantees were it not for the use of Vernam encryption (bit-wise exclusive or) in message 4. Once she has sent message 1, she cannot tell that any sequence of bits of the appropriate length is not the encryption of some session key under $k_A$. Nevertheless, we conducted a brief exercise to see what our methods would have produced with the stronger notion of authentication. As is frequently the case, the exercise in formulating the specifications made it hard not to spot the problems, and actually running checks revealed a variety of weak attacks.

# 8 Further protocols: generative servers

## 8.1 Needham-Schroeder Symmetric Key

The Needham-Schroeder Symmetric Key (NSSK) protocol is one of the best-known and most studied protocols in the literature. The objective is to establish a session key between two participants using only symmetric key encryption. Like the TMN protocol, it uses a server, but one with completely different functionality. The sequence of messages is given below[9]

[9] The precise form of the message 5 stated here is slightly different from others found in the literature, but has essentially the same meaning. The more standard form of the final message would contain the encrypted message $N_B - 1$, to differentiate between message 4 and 5. Since we are seeking a data independent type, we cannot perform a subtraction of this kind. Therefore we simply use another operation that straightforwardly makes the body of

Message 1. $A \to S : (A, B, N_A)$
Message 2. $S \to A : \{N_A, B, K, \{K, A\}_{Kbs}\}_{Kas}$
Message 3. $A \to B : \{K, A\}_{Kbs}$
Message 4. $B \to A : \{N_B\}_K$
Message 5. $A \to B : \{N_B, N_B\}_K$

where $A$ is the initiator agent, $B$ is the responder agent, $S$ is the server, $N_A$ and $N_B$ are nonces, $A$ and $B$ respectively have long term symmetric keys $Kas$ and $Kbs$ with the server, and $K$ is symmetric key generated by the server for the session between agents $A$ and $B$.

This protocol differed from those we have seen before in two important respects:

- There are now two separate data independent types that we will need to collapse: nonces and session keys. In fact there are arguments for splitting the nonces into two types: those used for messages 1 and 2, and those used for messages 4 and 5.
- The server is now responsible for generating fresh values of one of the types in question. This means that its behaviour cannot simply be embedded into the intruder as an extra sort of deduction.

The first of these problems simply requires that we have more than one manager process present in the network. We had no difficulties in running versions with three or two of these (depending on whether the nonces had been split in two or not).

The increased complexity of this protocol (in terms of the size of messages, which directly relates to the cardinality of alphabet and number of facts relevant to the intruder) made it unattractive to lose the advantage gained by incorporating the server's functionality into the intruder. It is, however, seemingly impossible to do this within the established model of the intruder (something we were able to do in the case of TMN). This is because there is no way that a deductive system of the form used in the intruder can produce a notion of freshness, which requires that when a given value has been used once it may not be used again. After experimenting with several possible solutions to this, we decided that the best and most general solution is the following.

In the existing intruder model we have events of the form `infer.(X,f)` which are communicated when, on the basis of our cryptographic assumptions, a intruder with knowledge `X` (a finite set of facts) could reasonably create `f` (a single fact). We decided to add an analogue of this that would allow the intruder (in its role as a pseudo-server) to synchronise with key/nonce manager processes to create the messages that the server issues containing fresh values. For simplicity, we consider here the case where the server is only issuing a single fresh value at once.

In the real system we are trying to model, the proper agents or the intruder can get the server to produce one or more messages that will depend on this fresh value by sending an appropriate set of requests to it. In the case of NSSK, there is only one message produced (message 2) and one required to get it to do so. We now model this behaviour by a modified version of the inferences above. A *generation* is a triple $(k, X, Y)$, where $k$ is the fresh object being created (in the case of NSSK a session key), $X$ is the set of objects that the intruder has to have in its possession to get the server to act (in our case a single set containing message 1) and $Y$ is the set of messages containing $k$ that the intruder can thus learn (in our case, a singleton set containing the message 2 that corresponds to the other two components). We will see examples later where $X$ and $Y$ are not singleton sets. The event `server.(k,X,Y)` can then occur just when

message 5 from that of message 4 without allowing a replay of one as the other. Plainly this substitution would not be valid if either this changed any potential confusion of messages or either allowed or disallowed any potential deductions. We are confident that with the standard cryptographic model there are no such problems here.

- the (key) manager issues the fresh value $k$ (by synchronising with this action), and
- each fact in $X$ is known by the intruder,

and leads to the state where the intruder knows each of the facts in $Y$. (Assuming the value $k$ really is fresh, we can guarantee that the intruder does not know anything in $Y$. Unless $k$ is a member of $Y$ – which is very unlikely – the intruder does not learn $k$ directly from this action.)

The involvement of the key manager process means that a given generation will only occur once (until a key gets recycled using a collapse). This is why the final component of a generation has to be a set of results – which are all tied to a single key – rather than a single fact.

There is one remaining problem, namely that the intruder can ask the server (whether the server is a separate process or is included in the intruder in the way described above) for as many fresh keys (buried in message 2's) as it likes, given a single message 1. And similar situations will arise in other protocols where the server is generating fresh values: as soon as the intruder can validly ask for one it can ask for any number. Evidently this effect has to be controlled if we are to keep the sizes of our types of fresh values small and finite. The approach we have taken has been to forbid the intruder from requesting a new fresh value while both Alice and Bob are already holding as many as they can, on the grounds that the intruder cannot then exploit the value immediately against Alice or Bob, and it might as well have waited until one of them had forgotten a value before requesting the fresh one. We think this will be a valid argument in just about all practical examples, but in order to justify it in a given case, you must ensure that there is nothing the intruder might be able to get at (either directly or using subsequent inferences) *not* involving a fresh value that the intruder might be able to obtain, and then exploit, as a side-effect.

We were able to prove that this protocol, under standard cryptographic assumptions, satisfies both the `Session`-based specifications and last-message authentication specifications analogous to those described for NSPK in Section 6.

More interesting results were obtained when we assumed that old session keys were compromised. This requires only a trivial modification of our model: when a forgotten key that the intruder does not know is sent to a background value, then it gets mapped to one he does know rather than one he does not. Upon doing this our model (as we expected) immediately yielded an equivalent of the Denning-Sacco attack [4], whereby the responder is susceptible to a replay of a message 3 (which the intruder knows) containing an old session key which he has cracked. He is then able to respond (pretending to be the initiator) to message 4. A side effect of our inclusion, as described earlier, of the data of stale sessions in the intruder's initial knowledge, is that this attack appears on a very short trace. This is because in the classic presentation of the attack, most of the messages are present simply to give the intruder knowledge which it will later exploit as stale, and obviously these messages are not now needed.

### 8.2 Otway-Rees Protocol

The Otway-Rees protocol consists of the same components described in the Needham-Schroeder Symmetric Key protocol and is defined in [32] as the following sequence of messages:

MESSAGE 1. $A \to B : (I_A, A, B, \{N_A, I_A, A, B\}_{Kas})$
MESSAGE 2. $B \to S : (I_A, A, B, \{N_A, I_A, A, B\}_{Kas}, \{N_B, I_A, A, B\}_{Kbs})$
MESSAGE 3. $S \to B : (I_A, \{N_A, K\}_{Kas}, \{N_B, K\}_{Kbs})$
MESSAGE 4. $B \to A : (I_A, \{N_A, K\}_{Kas})$

where $A$ is the initiator agent, $B$ is the responder agent, $S$ is the server, $N_A$ and $N_B$ are nonces belonging to agents $A$ and $B$ respectively, $I_A$ is the index number generated by the initiator agent $A$ and $K$ is a symmetric key generated by the server for the session between agents $A$ and $B$. We treat $I_A$ in the same way as we treat a nonce. Each run has four values that we might expect to be fresh: three nonces and a session key.

The only encryption method used in this protocol is symmetric key encryption. The server $S$ shares a unique symmetric key with each agent (unknown to the others) which is used to encrypt/decrypt messages between them.

In this protocol, we have an additional data independent type, namely, the index value generated by the initiating agent in message 1. This index value is passed through every message, and clearly becomes known to the intruder as soon as it is generated.

The function of the server here is similar to the NSSK protocol in that it is responsible for the generation of fresh session keys. The first message received is from the responder agent (in the definition above this is agent $B$) consisting of an index number, two agent names, and the two messages from the initiator and the responder agents each encrypted in their respective symmetric keys.

An important feature of the Otway-Rees protocol which differs from previous protocols we have analysed, is the *length* of the individual messages, in particular, message 2 and message 3. These messages contain significantly more data objects than the message constructions of other protocols. The resultant effect of this is that the state space required for the standard checks we perform will be significantly increased, probably beyond what we can handle.

Our aim is to redefine the set of messages such that it still correctly represents the Otway-Rees protocol, but reduces the length (in terms of the number of different data objects) of each message. We define the messages as follows[10]:

MESSAGE 1. $A \to B : (I_A, A, B)$
MESSAGE 2. $A \to S : \{N_A, I_A, A, B\}_{Kas}$
MESSAGE 3. $B \to S : \{N_B, I_A, A, B\}_{Kbs}$
MESSAGE 4. $S \to A : (I_A, \{N_A, K\}_{Kas})$
MESSAGE 5. $S \to B : (I_A, \{N_B, K\}_{Kbs})$

The underlying change has been to divide the old message 2 and message 3 construction into two separate messages each.

The design used for implementing the Otway-Rees protocol model was based on the implementation methods presented for the Needham-Schroeder Symmetric Key protocol above. We need to introduce an Index Manager which has the same style of implementation as the Nonce Manager, except that it is responsible for supplying fresh index values to the initiator agent of a run.

We must also redefine our *Generations* set, which determines what deductions the intruder is able to perform (in order to reflect the server's functionality). In this protocol, we need the intruder to be able to obey the following inference rule:

If the intruder knows a message 2 and a message 3, such that the index values, the

[10]This restructuring of messages defining the Otway-Rees protocol has been used before, for example by Lowe, and it can been shown that any attack on the revised or original protocol, could be transformed from one to the other. The basic argument here is that $B$ is simply acting as a dumb messenger for the components of messages 2 and 3 of the first encoding that he cannot understand; he cannot check their contents and simply passes them on blindly. The intruder cannot do anything different to this message in two hops via $B$ than it could do to the same message in a single hop, and all the transformation does is to transform the two-hop transmissions to direct ones.

initiator agent identities and the responder agent identities in both messages are all equal (respectively taking values $I$, $A$ and $B$), and he is given some key $K$, then he is able to infer the following two messages:

1. One message 4 consisting of the index value $I$, initiator agent $A$ and the nonce $N_A$ present in the corresponding message 2 and the given key $K$, and
2. One message 5 consisting of the index value $I$, responder agent $B$ and the nonce $N_B$ present in the corresponding message 3, and the given key $K$.

Below, we present an example of a valid deduction the intruder could perform:

The intruder has intercepted the following message 2 from *Alice* to the server *Sam*:

$$Alice \rightarrow \{I_A, N_A, Alice, Bob\}_{Kas} \rightarrow Sam,$$

followed by the intruder intercepting the following message 3 from *Bob* to the server:

$$Bob \rightarrow \{I_A, N_B, Alice, Bob\}_{Kbs} \rightarrow Sam$$

From these two messages, the intruder is able to infer the following message 4 and 5 respectively:

$$(I_A, \{N_A, K\}_{Kas}), \quad \text{and} \quad (I_A, \{N_B, K\}_{Kbs})$$

where $K$ is some key which will be supplied by the key manager as in the case of NSSK.

In the NSSK model, the intruder could only ever deduce one message for a given key $K$. In the Otway-Rees protocol, we require the intruder to create two messages for every one key it receives, and we ensure that these two messages are precisely the correct ones by placing a more stringent constraint upon the preconditions of this deduction (i.e. the contents of the two corresponding messages 2 and 3). Note that the *Generations* set thus produced now has two members in the sets in the second and third components of each triple.

The two `Session`-based checks completed successfully, proving that this would be the case without collapsing and with infinite supplies of indexes, nonces and keys.

The final-message authentication check that the initiator is authenticated to the responder (i.e., that $B$ does not accept message 5 unless $A$ has sent messages 1 and 2) failed. This was a surprise to us when it occurred, though subsequently we discovered the same weakness in this protocol described in [2]. It revealed that the intruder can convince $B$ that $A$ wants a session with him by replaying messages 1 and 2 from an old session that $A$ and $B$ have run in the same roles. (With the original version of the protocol the corresponding action would be a replay of an old message 1.) This leads to $B$ getting a key in message 5, after the intruder has stopped message 4 from reaching $A$ (who is not expecting it anyway). This attack does not reveal any secrets to the intruder, but does allow him to convince $B$ that $A$ has recently wanted a session with $B$.

As was the case with the Denning-Sacco attack, this attack appeared after a short trace because of the way it exploits stale data which we include in the intruder's initial knowledge.

### 8.3 Yahalom Protocol

The Yahalom protocol (see [2]) is known to be a particularly subtle protocol. It consists of the same components described in the Otway-Rees protocol and is defined as the following sequence of messages:

MESSAGE 1. $A \to B : (A, N_A)$
MESSAGE 2. $B \to S : (B, \{A, N_A, N_B\}_{Kb})$
MESSAGE 3. $S \to A : (\{B, K, N_A, N_B\}_{Ka}, \{A, K\}_{Kb})$
MESSAGE 4. $A \to B : (\{A, K\}_{Kb}, \{N_B\}_K)$

Here, $A$ is the initiator agent, $B$ is the responder agent, $S$ is the server, $Ka$ and $Kb$ are the symmetric keys belonging to $A$ and $B$ respectively (known only by themselves and the server), and $K$ is the symmetric session key supplied by the server.

Like Otway-Rees and NSSK it uses the server to generate fresh session keys, so we must expect to use a similar structure of CSP model to those seen for them. Like NSSK it uses the new session key to provide a final authentication, and this creates the danger of a Denning-Sacco-like attack in the case where a session key gets compromised. Notice that, as with Otway-Rees, one of the agents ($A$) acts as a dumb messenger for an encrypted packet understandable only to $S$ and $B$. For the same reasons, and with the same justification, we chose to restructure the protocol by making the transmission direct, thus splitting messages 3 and 4:

MESSAGE 1. $A \to B : (A, N_A)$
MESSAGE 2. $B \to S : (B, \{A, N_A, N_B\}_{Kb})$
MESSAGE 3A. $S \to A : \{B, K, N_A, N_B\}_{Ka}$
MESSAGE 3B. $S \to B : \{A, K\}_{Kb}$
MESSAGE 4. $A \to B : \{N_B\}_K$

Given this restructuring, the *Generations* set (again modelling the way the server creates keys) has members whose second components contain just a message 2, and whose third component contains the corresponding messages 3a and 3b.

The analysis of this protocol using our methods presented no new challenges. It proved to satisfy both the `Session`-style and the stronger authentication specifications, even under the assumption that old session keys and old nonces are compromised when the responder is guaranteed to time out of a protocol run before the intruder deduces the session key (possibly from messages the initiator sends under it once she thinks the protocol is complete). Failure to do this would allow the intruder to finish off the protocol with the responder even though the initiator may have closed or otherwise have given up on the session. In other words, the susceptibility exposed, in the case of NSSK, by the Denning-Sacco attack does not apply here provided we make reasonable assumptions and take reasonable care in implementation.

# 9 Conclusions and prospects

We have developed methods by which it is possible to prove far more complete results on model checkers than hitherto. We are confident that any protocol which uses the range of security features seen in the examples in this paper (standard public and symmetric key encryption and nonces) can be addressed using these methods. The use of specific encryption methods with further properties, such as the Vernam encryption used in TMN, do not cause problems provided these can be described symbolically in a positive deductive system.

There are, of course, other features used for security in protocols. Hashing should not present any difficulties to our approach, since it is symbolically no different to asymmetric encryption in which no-one knows the decrypting key. Timestamps and sequence numbers present more of a problem, however, since the operations on them (comparison, adding, etc.) take them outside the range of data independence – further research is required here.

One should always be careful to state the limits of what one has proved in a case like this. Evidently the use of an abstract data type and a specified set of deductive rules over it implies that we are assuming, in our proof, that there are no other subtle properties of the encryption system that an intruder can exploit and, unless it has been built into the deductive system in some way, no way that the intruder can decrypt messages without the possession of the appropriate key. Secondly, we are assuming that a node will impose whatever discipline on accepting messages is implied by the protocol, in particular that it will not interpret a message in one shape as a message in another. Thus we have not allowed in this treatment for attacks based on type confusion, under which, for example, some advantage might be gained if an agent could be persuaded to accept another agent's name as a nonce.

An important limitation on the result one has proved with a check of the form we have described is that it does not normally allow for either of Alice or Bob running more than one session at a time. If it is realistic that they might, then you should really include an appropriate number of copies of each in the network, which would in turn increase the number of foreground values in our types. This would increase the number of states in our example to a prohibitive level even with two copies of each agent[11], and of course we would like to prove appropriate results for an arbitrary number of copies of each – in other words, factoring a further parameter out of the system. This must remain a topic for future research; one possible approach we intend to investigate is to incorporate part of the trustworthy agents into the intruder model, as we have already done to the server. This would give the intruder the ability to use the honest activities of separate instances of a given identity against a fixed one, just as the techniques we have developed here already allow the intruder to use the proper activities of the server for its own ends.

Our analysis has been based on trace specifications, both because the protocol models we are considering do not attempt to satisfy any stronger variety – to do so requires many further implementation details and assumptions about the intruder – and because it simplifies the data independence arguments. It is, nevertheless, possible to apply data independence arguments to stronger styles of specifications, as shown by [28, 12], for example. The fact that we do not now have to restrict how many runs an agent can perform suggests that our new modelling techniques are likely to be extremely useful in liveness and no-loss-of service analyses. Something that would require care in this event would be any use made of the trick of incorporating the server into the intruder's deductive system (if necessary via the set of generations). For we would have to distinguish between general intruder actions – upon which we would not wish to place any reliance – and those in which it acts as a server. For in the latter case it is the *absence* of communications that represent malicious behaviour.

Three separate pieces of work in this area by Gavin Lowe will have a direct bearing on our methods.

(1) He has followed an independent line of research with the same general end of proving protocols via finite check [19]. Instead of using data independence he establishes conditions, by detailed reasoning by case of what could happen, under which any attack on a protocol would appear in a check containing just two agents (no separate name like `Cameron` for the intruder) who are able to run the protocol only once each. The restrictions he works under are fairly severe, both on the nature of the protocol and on the specification – thus far he can consider only secrecy rather than authentication, and none of the protocols we have considered meet his constraints. His result, when it applies, has the advantage that the necessary check is smaller and covers without restriction the case of parallel protocol runs

[11] The most we have been able to run, even for the simpler of the protocol examples, is two copies of one agent and one of the other.

by one agent.

One can imagine using methods such as these in conjunction with ours, in cases where it proves impossible to get such a tight bound as in Lowe's result. This might be to cut down the size of necessary checks by removing the need to search for some types of behaviour, or to encompass parallel runs.

(2) In [10], M. Hui and G. Lowe have formulated ways of simplifying complex protocols in ways that guarantee not to eliminate attacks. This is practically very important work since it means that it becomes possible to address practical protocols from areas like electronic commerce that are considerably more complex than the comparatively tractable examples in this paper, and which in their original forms would not be checkable on FDR etc. The methods we have developed in this paper could be used to prove properties of the simplified protocols, and hence the original ones. The conditions under which the simplification methods apply have much in common with some of the work in this paper: for example they require something very close to our concept of a positive deductive system.

(3) Finally, there would seem to be significant potential advantages in extending Lowe's Casper protocol-to-CSP compiler to analyse the applicability of the techniques in this paper and build the relatively complex CSP programs needed to exercise them. The creation of one of these programs is an essentially mechanical procedure in which a large number of inter-related details have to be remembered, something which makes it all too easy for human beings to make mistakes. We hope to begin work on this shortly.

There seems no reason in principle why our techniques should not be applicable over a wider range of notations and model checkers than CSP and FDR: modelling protocols has become a popular pastime for users of a wide range of tools in the last year or two. The formality of arguments based on other notation would, however, be limited if they did not have a corresponding theory of data independence.

Readers interested in discovering the details of the CSP coding of the protocols with manager processes, etc., described in this paper can obtain a number of examples via URL:

`http://www.comlab.ox.ac.uk/oucl/publications/books/concurrency/examples/security`

## Acknowledgements

This work would have been impossible without the pioneering efforts of Ranko Lazić in developing the theory of data independence upon which the arguments here are based. Equally it would have been impossible without the continued work of the staff of Formal Systems on the FDR tool. The paper has been much improved by comments from anonymous referees and G. Lowe.

The work reported in this paper was supported by DERA Malvern and the U.S. Office of Naval Research.

# Appendix: An Overview of the CSP notation

We will first give a brief overview of the basic machine-readable CSP syntax which is used in this paper. This will be followed by a more detailed description of the CSP notation used in specific examples throughout the paper.

## Fundamental CSP operators

The CSP processes that we use are constructed from the following operators:

- `STOP` is the simplest CSP process, which can do nothing and is therefore equivalent to deadlock.
- `a -> P` is the process which is initially willing to communicate `a` and will wait indefinitely for the `a` to happen. Once event `a` has occurred, it behaves like `P`.
- `P [] Q` is a process which offers the environment the choice of the first events of `P` and of `Q` and then behaves accordingly (`[]` is known as the external choice operator). This means that if the first event chosen is from `P` only, then `P [] Q` behaves like `P`, while if one is chosen from `Q` only, then it behaves like `Q`. If the environment offers the initial events of both `P` and `Q`, then `P [] Q` will non-deterministically select one. Thus the choice of what happens is in the hands of the environment.
- `P |~| Q` is a process which can behave either as process `P` or as `Q`, the choice of which is non-deterministic from the point of view of the environment. This process may reject either the request of the initial event of `P` or that of `Q` by the environment. However, if the environment offers all initial events, then one must take place.
- `P` $\|$ $[X]$ `Q` represents the process where all events in $X$ must be synchronised, and events outside $X$ can proceed independently. This operator is called the *generalised* or *interface* operator.
- `[] x:T @ P(x)` represents the replication of the external choice operator over `P(x)`, where `x` ranges through the set `T`. For example,

  `[] x:{1,2,3} @ P(x) = P(1) [] P(2) [] P(3)`

- `|~| x:T @ P(x)` represents the replication of the internal choice operator over `P(x)`, where `x` ranges through the set `T`. For example,

  `|~| x:{1,2,3} @ P(x) = P(1) |~| P(2) |~| P(3)`

- `P \ X` is the process which behaves like `P` except that the events in `X` have been internalised and therefore become invisible to and uncontrollable by the environment of `P`. This is known as *hiding*.

## Further Detailed CSP Examples

### Agent Processes

A sender agent `id` initiating a run of Lowe's revised Needham Schroeder Public Key protocol [16] is represented by the following CSP process:

```
Send(id) =
|~| b:diff(agents,{id}) @
  let na=???? within
  comm.id.b.(PK.(pk(b),Sq.<na,id>)) ->
   ([] nb:nonces @
    comm.b.id.(PK.(pk(id),Sq.<nb,na,b>)) ->
     comm.id.b.(PK.(pk(b),nb)) ->
                    Session(id,b,nb,na))
```

which is interpreted as follows:

- `|~|b:diff(agents,{id}) @ ...` - represents the replicated internal choice over all agents defined within the system (in the set `agents`) except for itself. This internal choice reflects the initiator agent choosing itself who to start a protocol run with. All proceeding events containing the variable `b` are bound to this internal choice made.
- `let na=???? within ...` - the choice of the nonce used by `Send(id)` is left undecided, since that is the essence of this paper.
- `comm.id.b.(PK.(pk(b),Sq.<na,id>))` - represents the communication of a message 1 from sender agent `id` to responder agent `b` through the channel `comm`. The message `PK.(pk(b),Sq.<na,id>)` represents the encryption of `<na,id>` under the public key of agent `b`, where `na` is the nonce supplied by `id`.
- `[] nb:nonces @ ...` - represents the replicated external choice over all possible nonces defined within the system. This external choice reflects the sender agent accepting any nonce from the responder agent. This is important since the nonce `nb` is supplied by the responder agent and therefore the sender should be willing to accept any nonce existing in the system.
- `comm.b.id.(PK.(pk(id),Sq.<nb,na,b>))` - represents the communication of a message 2 from the responder agent `b` to the sender agent `id` through the channel `comm`. The message `PK.(pk(b),Sq.<nb,na,b>)` represents the encryption of `<nb,na,b>` under the public key of the sender agent `id`, where `nb` is the nonce supplied by `b` and `na` is the nonce supplied by `id`.
- `comm.id.b.(PK.(pk(b),nb))` - represents the communication of a message 3 from the sender agent `id` to the responder agent `b` through channel `comm`. The message `PK.(pk(b),nb)` represents the encryption of `nb` under the public key of the responder agent `b`, where `nb` is the nonce supplied by `b`.

### Abstract Data Types

At the simplest level a data type can be used to define a number of atomic constants, for example:

```
datatype Agent = Alice | Bob | Cameron
```

This means that any variable set to the type `Agent` can either take the value `Alice` or `Bob` or `Cameron`.

A data type can also be defined in terms of a number of values which are associated with tags, for example

```
datatype T = Agent. AGENT | Nonce. NONCE
```

where `AGENT = {Alice, Bob, Cameron}` and `NONCE = {Na,Nb,Nc}`. Examples of possible constructs of type `T` are `Agent. Alice` and `Nonce. Na`.

A data type can also be defined recursively, for example:

```
datatype fact = PK. (fact,fact) |
                pk. AGENT |
                Nonce. NONCE
```

where `NONCE = {Na, Nb, Nc}` and `AGENT = {Alice, Bob,Cameron}`. An example construct of type `fact` is `PK. (pk. Alice, Nonce. Nb)`.

The messages passed round the implementation of a protocol are drawn from an abstract, constructed data type that allows us to treat operations such as encryption as symbolic. For example, for Lowe's revised Needham Schroeder Public Key protocol, we defined the following data type to represent all possible message constructions:

```
datatype fact = Sq. Seq(fact) |
                PK. (fact, fact) |
                Encrypt. (fact, fact) |
                Agent. AGENT |
                Nonce. NONCE |
                pk. AGENT | sk. AGENT |
                AtoB | BtoA | Cmessage
```

where `AGENT` is the set of agents and `NONCE` is the set of nonces defined within the protocol implementation. For example, from this data type, we can construct a symbolic representation of the encryption of the message $\{Na, Alice\}$ (message 1 of this protocol) under the public key of *Bob* as follows:

```
PK. (pk. (Bob), Sq. <Nonce. Na, Agent. Alice>)
```

## References


[1] P.J. Broadfoot, *Application of Data Independence Techniques to Security Protocols*, Oxford University MSc Dissertation, 1998.

[2] M. Burrows, M. Abadi and R.M. Needham, *A logic of authentication*, Proceedings of the Royal Society of London **426**, 233-271, 1989.

[3] D. Coppersmith, M. Franklin, J. Patarin, and M. Reiter, *Low-exponent RSA with related messages*, In Advances in Cryptology – EUROCRYPT '96 (LNCS 1070), 1996.

[4] D.E. Denning and G.M. Sacco, *Timestamps in key distribution protocols*, Communications of the ACM **24**,8, 533-536, 1981.

[5] W. Diffie, P.C. van Oorschot and M.J. Weiner, *Authentication and key exchanges*, Design, Codes and Cryptography **2**, 107-125, 1992.

[6] D. Dolev and A.C. Yao, *On the security of public key protocols*, IEEE Transactions on Information Theory, **29**, 2, 1983.

[7] Proceedings of DIMACS workshop on the design and formal verification of cryptographic protocols, 1997. Published on the world-wide web at URL: `http://dimacs.rutgers.edu/Workshops/Security/program2/program.html`

[8] E.A. Emerson and A.P. Sistla, *Utilizing symmetry when model checking under fairness assumptions: an automata-theoretic approach*, Proceedings of the 7th CAV, Springer LNCS **939**, 309-324, 1995.

[9] E.A. Emerson and A.P. Sistla, *Symmetry and model checking*, Formal Methods in System Design **9**, 105-131, 1996.

[10] M. Hui and G. Lowe, *Safe simplifying transformations for security protocols*, Proceedings of the 12th IEEE Computer Security Foundations Workshop, 1999.

[11] C.N. Ip and D.L. Dill, *Better verification through symmetry*, Formal Methods in System Design **9**, 41-75, 1996.

[12] R.S. Lazić, *A semantic study of data-independence with applications to the mechanical verification of concurrent systems*, Oxford University D.Phil thesis, 1998.

[13] R.S. Lazić and A.W. Roscoe, *Using logical relations for automated verification of data-independent CSP*, Proceedings of the Workshop on Automated Formal Methods (Oxford, U.K.), Electronic Notes in Theoretical Computer Science **5**, 1997.

[14] R.S. Lazić and A.W. Roscoe, *A semantic study of data-independence with applications to model-checking*, Submitted for publication, 1998.

[15] R.S. Lazić and A.W. Roscoe, *Verifying determinism of data-independent systems with labellings, arrays and constants*, Proceedings of INFINITY 1998.

[16] G. Lowe, *Breaking and fixing the Needham-Schroeder public-key protocol using FDR*, Proceedings of TACAS '97, Springer LNCS 1055, 1996.

[17] G. Lowe, *Some new attacks upon security protocols*, Proceedings of 1996 IEEE Computer Security Foundations Workshop, IEEE Computer Society Press, 1996.

[18] G. Lowe, *Casper: a compiler for the analysis of security protocols*, Proceedings of 1997 IEEE Computer Security Foundations Workshop, IEEE Computer Society Press, 1997.

[19] G. Lowe, *Towards a completeness result for model checking of security protocols*, Proceedings of the 11th IEEE Computer Security Foundations Workshop, 1998.

[20] G. Lowe and A.W. Roscoe, *Using CSP to detect errors in the TMN protocol*, IEEE transactions on Software Engineering **23**, 10, 659-669, 1997.

[21] J.D. Guttman, *Honest Ideals on Strand Spaces*, Proceedings of the 11th IEEE Computer Security Foundations Workshop, 1998.

[22] J.J. Mitchell, *Type systems for programming languages*, in 'Handbook of theoretical computer science' (van Leeuwen, *ed*), Elsevier, 1990.

[23] L.C. Paulson, *Mechanized Proofs of Security Protocols: Needham-Schroeder with Public Keys*, Report **413**, Cambridge University Computer Lab, 1997.

[24] G.D. Plotkin, *Lambda-definability in the full type hierarchy*, in 'To H.B. Curry: essays on combinatory logic, lambda calculus and formalism' (Seldin and Hindley, *eds*), Academic Press, 1980.

[25] J.C. Reynolds, *Types, abstraction and parametric polymorphism*, Information Processing **83**, 513-523, North-Holland, 1983.

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

[27] A.W. Roscoe, *Intensional specifications of security protocols*, Proceedings of 1996 IEEE Computer Security Foundations Workshop, IEEE Computer Society Press, 1996.

[28] A.W. Roscoe, *The theory and practice of concurrency*, Prentice Hall, 1998.

[29] A.W. Roscoe, *Proving security protocols with model checkers by data independence techniques*, Proceedings of the 11th IEEE Computer Security Foundations Workshop, 1998.

[30] A.W. Roscoe and M.H. Goldsmith, *The perfect 'spy' for model-checking crypto-protocols*, in [7].

[31] A.W. Roscoe and J.T. Yantchev, *Testing specifications for better compression*, to appear.

[32] B. Schneier, *Applied Cryptography*, John Wiley & Sons, 1996.

[33] M. Tatebayashi, N. Matsuzaki and D.B. Newman, *Key distribution protocol for digital mobile communication systems*, Advances in Cryptology: Proceedings of Crypto '89, Lecture Notes in Computer Science **435**, 324-333, Springer-Verlag, 1990.

[34] P.L. Wadler, *Theorems for free!*, 347-359, Proceedings of the $4^{\text{th}}$ ACM FPLCA, 1989.

[35] P. Wolper, *Expressing interesting properties of programs in propositional temporal logic*, 184-193, Proceedings of the $13^{\text{th}}$ ACM POPL, 1986.