AVIS'04 Preliminary Version

# On the Application of Counterexample-Guided Abstraction Refinement and Data Independence to the Parameterised Model Checking Problem

Gavin Lowe

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

*Key words:* Parameterised systems, model checking, data independence, counterexample-guided abstraction refinement, security protocols.

## 1 Introduction

Model checkers can be used to explore the state spaces of models of systems, to discover if a property is satisfied. If the property is not satisfied, the model checker returns a counterexample, which can be used for debugging.

In this paper we concentrate on model checking of CSP traces-refinement—i.e. trace set inclusion—using the FDR model checker. However we believe that the techniques we describe would be easily transferable to other formalisms.

Many models of systems are parameterised in some way. For example, many systems are built from an arbitrary number of components belonging to one or more classes. One can therefore think of the model of the system as being parameterised by the numbers of components from the classes, possibly augmented with extra information about those components. In this paper we will illustrate our techniques by analysing a security protocol; the model of the system will be parameterised by the instances of honest agents running the protocol: their identities, the roles they take, and the nonces they use.

Also, many models of systems are parameterised by the underlying types. In our security protocol example, the model of the system will be parameterised by the types of agent identities and nonces. These types are, in practice, unbounded; however, model checking is feasible only for small choices of the types.

*This is a preliminary version. The final version will be published in*
*Electronic Notes in Theoretical Computer Science*
*URL:* www.elsevier.nl/locate/entcs

One can therefore consider a model of the system, $System(Insts, Types)$, that is parameterised by the instances of components, and the types in the model. A common problem is to verify the system for all possible values of such parameters. If the requirements are captured by specification process *Spec*, then this problem can be expressed as a CSP traces-refinement as follows:

$$\forall Insts, Types \bullet Spec \sqsubseteq System(Insts, Types).$$

A model checker can only directly consider single values of the parameters. This problem is commonly known as the *parameterised model checking problem*.

In this paper we will present a general strategy for handling the parameterised model checking problem. Our strategy is to produce an abstraction of the system in question that also refines the specification:

$$(1)\quad \forall Insts, Types \bullet Abstraction(Types) \sqsubseteq System(Insts, Types),$$
$$(2)\quad \forall Types \bullet Spec \sqsubseteq Abstraction(Types).$$

The desired result then follows from transitivity of refinement.

The abstraction is produced using ideas from Clarke et al.'s technique of counter-example guided abstraction refinement [5]:

(i) Guess an abstraction such that equation (1) is true by construction;

(ii) Use the model checker to test whether equation (2) holds for small values of *Types*;

(iii) If not, use the counterexample produced by the model checker to improve the guess, while maintaining equation (1); goto step 2.

Once we have a suitable abstraction such that equation (2) holds for small values of *Types*, we use techniques from the area of data independence to prove that the same result holds for *all* values of *Types*.

Finally, we prove that the (claimed) abstraction really is an abstraction of the system in question, i.e. that it satisfies equation (1), again using techniques from data independence. (Of course, we could verify that each of the claimed abstractions satisfies equation (1), but it seems best to just verify this for the final abstraction, since it is reasonably straightforward to ensure this property is true by construction.)

This iterative approach is illustrated in Figure 1. We are given the specification and the system (*Spec* and *System*), and we have to find an abstraction ($Abs_n$) that lies between them; we do this by considering a sequence of abstractions ($Abs_0, Abs_1, \ldots$) and testing whether each refines the specification, until we find one that does.

This technique is not currently fully automated: the abstractions are constructed manually (although in the case of security protocols, the compiler Casper [15] can be used to construct the initial model); and the proofs are by hand, with some of the proof obligations carried out by the model checker. We believe that there are good prospects of improving the level of automation, particularly in the specific area of security protocols: we discuss this point

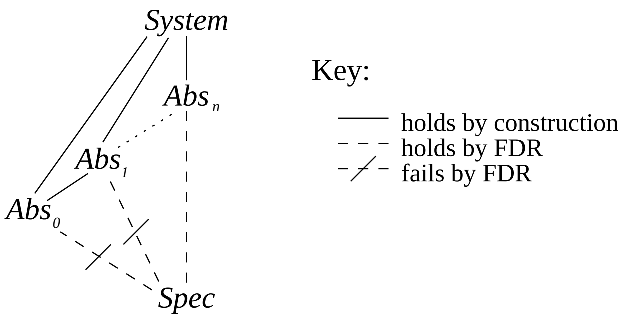


Fig. 1. The sequence of abstractions

further in the conclusions.

To the best of our knowledge, this paper represents the first application of counterexample-guided abstraction refinement and data independence to the parameterised model checking problem.

In this paper we illustrate this technique by using it to verify a well-known security protocol. In Section 2 we summarise the relevant background material about CSP and data independence. In Section 3, we introduce the protocol in question, and describe the basic model of the system and the specification. We find the correct abstraction in Section 4, using the iterative technique outlined above. In Section 5, we verify that equations (1) and (2) hold, using data independent techniques. We sum up in Section 6.

## 2 Background

In this section we briefly review the relevant parts of the syntax and semantics of CSP; see [19,10] for more details. We also describe the data independence techniques we will be using.

An event represents an atomic communication; this might either be between two processes or between a process and the environment. $\Sigma$ represents the set of all events. Channels carry sets of events; for example, $c.5$ is an event of channel $c$.

The process $STOP$ represents a deadlocked process, that can perform no events. The process $a \rightarrow P$ can perform the event $a$, and then act like $P$. The process $c?x \rightarrow P_x$ inputs a value $x$ from channel $c$ and then acts like $P_x$.

The process $P \mathrel{\Box} Q$ represents an external choice between $P$ and $Q$: the environment is given the choice between the initial events of the two processes. The process $P \sqcap Q$ represents a nondeterministic choice between $P$ and $Q$. The processes $\Box\, i : I \bullet P_i$ and $\bigsqcap i : I \bullet P_i$ are replicated external and nondeterministic choices, respectively, indexed over set $I$.

$P \underset{A}{\parallel} Q$ represents the parallel composition of $P$ and $Q$, synchronising on events from $A$; when the synchronisation set is the set of all events, $\Sigma$, we omit it. $P \mathrel{|||} Q$ represents an interleaving of the processes $P$ and $Q$, i.e. parallel composition without any synchronisation; $|||\, i : I \bullet P_i$ is an indexed interleaving. $P \setminus A$ acts like $P$, except all events from the set $A$ are hidden, i.e. made internal, so that they can occur without the cooperation of the environment. $Chaos(A)$ is the most nondeterministic process with alphabet $A$; it can perform any sequence of events from $A$, or deadlock, and can be defined by $Chaos(A) = (\bigsqcap a : A \bullet a \to Chaos(A)) \sqcap STOP$.

A *trace* is a sequence of events that a process can perform. Process $P$ is trace-refined by process $Q$ if all the traces of $Q$ are also traces of $P$:

$$P \sqsubseteq Q \Leftrightarrow traces(P) \supseteq traces(Q).$$

Trace refinement is a pre-congruence with respect to all the CSP operators (i.e. $P \sqsubseteq Q$ implies $F(P) \sqsubseteq F(Q)$ for all contexts $F(\_)$ definable using the CSP operators).

The model checker FDR can be used to test whether a model of a system trace-refines a specification process. Typically, the specification will capture all allowable traces.

Data independence is a tool that can be used to prove results about processes parameterised by a type. A process $P$ is said to be data independent with respect to a type $T$ if the only operations it performs on values of $T$ are to input them, store them, and output them; in particular, $P$ never performs any computation on values of $T$ that constrains what $T$ might be. In effect, $T$ is a polymorphic type parameter for $P$.

Data independence has been studied in the context of CSP by Lazić and Roscoe [12,19]. Precise, automatable, criteria for a CSP process to be data independent are given in [12]. They produce a series of *threshold theorems*, each of which produces a threshold $B$, such that if a refinement holds when a type parameter is instantiated with a type of size $B$, then the refinement also holds for all larger types. Such a theorem is given below.

A process satisfies the property $\mathsf{Norm}$ if, essentially, it contains no nondeterminism the effects of which are not immediately apparent. The property is satisfied by all sequential processes built using only prefixing, input, binary internal and external choices between processes with disjoint initial events, and guarded recursions (more general syntactic conditions are given in [12,19]). Almost all specification processes one ever wants to use satisfy this property, or can be easily transformed to satisfy it.

The following theorem is from [12,19].

**Theorem 2.1** *Suppose Spec and Impl are data independent processes with respect to type parameter $T$, and that Spec satisfies* $\mathsf{Norm}$. *Suppose*

$$B \geq W^{Spec} + W^{Impl} + L^{Impl},$$

*where: $W^{Spec}$ is the maximum number of values of type $T$ that Spec has to store for future use; $W^{Impl}$ is the corresponding value for Impl; and $L^{Impl}$ is the maximum number of values of type $T$ that can be input in a single event of Impl, or nondeterministically chosen in a single nondeterministic choice* *of Impl. Then*[1]

$$Spec(B) \sqsubseteq Impl(B) \Leftrightarrow \forall M \geq B \bullet Spec(M) \sqsubseteq Impl(M).$$

The intuition behind this theorem is that the specification and implementation processes depend only upon the values of at most $B$ variables, and different values for those variables do not fundamentally change their behaviours (because they are data independent), and so if the refinement holds when the type parameter is instantiated with a type of size $B$, then the same refinement will hold for all larger types.

A process satisfies the property $\mathsf{PosConjEqT}'_{\mathsf{C}}$ (for $C$ a set of constants) if all equality tests, other than those involving equality tests with members of $C$, are such that the process leads to $Stop$ when the equality fails. For example:

- The process

$$in?x : T \rightarrow in?y : T \rightarrow \text{if}\, x = y \text{ then } a \rightarrow Stop \text{ else } Stop.$$

  satisfies the condition, because the "else" case is $Stop$.
- For $c \in C$, the process

$$in?x : T \rightarrow \text{if}\, x = c \text{ then } a \rightarrow Stop \text{ else } b \rightarrow Stop.$$

  satisfies the condition, because the condition concerns only equality tests not involving constants.

The following result is from [20]; it allows us to deduce results about a process with type parameter $T$ by considering the same process with type parameter $\phi(T)$ (when we use this theorem, $\phi$ will collapse all types to some fixed small type $T_0$).

**Theorem 2.2** *If a process $P(T)$ satisfies $\mathsf{PosConjEqT}'_{\mathsf{C}}$, and uses no constants other than $C$, and $\phi$ is faithful to the constants of $C$ (i.e. $\forall x : T, c : C \bullet \phi(x) = \phi(c) \Rightarrow x = c$), then*

$$\{\phi(tr) \mid tr \in traces(P(T))\} = traces(P(\phi(T))).$$

The intuition behind this theorem is that $P$ has essentially the same behaviours using the type $T$ as with the potentially smaller type $\phi(T)$. With the smaller type, the "else" branch in constructs of the form "if $x = y$ then $P$ else $Q$" may become unreachable, but the condition $\mathsf{PosConjEqT}'_{\mathsf{C}}$ ensures that we do not lose any traces as a result.

## 3 The basic model

In this section we introduce the example of our analysis, and present the basic CSP model. The Adapted Needham Schroeder Public Key Protocol [13,14],

[1] $Spec(B)$ represents $Spec$ with the type parameter instantiated with a type of size $B$.

is as follows:

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

We use standard notation to describe protocols: $a$ and $b$ are agents' identities; $PK(a)$ represents the public key of $a$; $na$ and $nb$ are nonces, that is, large random numbers freshly invented for this run of the protocol; the notation $\{m\}_k$ represents message $m$ encrypted with key $k$. The protocol aims to authenticate $a$ and $b$ to one another, and to establish a pair of shared secrets $na$ and $nb$. The original version of the protocol [18] did not include the identity $b$ in the encrypted component of message 2; this led to an attack on the protocol, where an intruder could imitate the initiator to a responder [13,14].

The compiler Casper [15] produces CSP models of protocols from more concise descriptions. The model described below was produced by editing such a Casper-produced model.

In the CSP model, each message has to be represented by an element from a CSP datatype. The precise representation is not important, but we present the standard representation here, and define shorthands as follows:

$$\begin{aligned} msg1(a, b, na) &= a.b.(Msg1, Encrypt.(PK(b), \langle na, a\rangle)), \\ msg2(b, a, na, nb) &= b.a.(Msg2, Encrypt.(PK(a), \langle na, nb, b\rangle)), \\ msg3(a, b, nb) &= a.b.(Msg3, Encrypt.(PK(b), \langle nb\rangle)). \end{aligned}$$

We can build CSP processes to model the honest agents. The process $Initiator(a, na)$ models an initiator with identity $a$ who uses the nonce $na$; it chooses nondeterministically which responder $b$ to run the protocol with:

$$\begin{aligned} Initiator(a, na) = \textstyle\bigsqcap b : Agent \bullet send.msg1(a, b, na) \rightarrow \\ \Box\, nb : Nonce \bullet receive.msg2(b, a, na, nb) \rightarrow \\ send.msg3(a, b, nb) \rightarrow Stop. \end{aligned}$$

The process $Responder(b, nb)$ models a responder with identity $b$ who uses the nonce $nb$, and is willing to receive a message 1 from an arbitrary initiator $a$; it is defined in a similar way.

We will concentrate on the following question:

> Is the initiator $a$ of the protocol authenticated to the responder $b$ (for $a \neq b$), in the sense that if $b$ receives a valid message 3 from $a$, then $a$ previously sent that message, intended for $b$?

By symmetry, it is enough to consider the following instance of this question:

> Is the initiator $Alice$ of the protocol authenticated to the responder $Bob$, in the sense that if $Bob$ receives a valid message 3 from $Alice$ using the nonce $N$, then $Alice$ previously sent that message, intended for $Bob$?

Thus we concentrate on whether Alice is authenticated to the instance $Responder(Bob, N)$. We'll call this the *principal responder process*. In order to test this, we will check that, once all other events are hidden, the system refines the specification $Spec$, where

$$\begin{aligned} Spec &= send.msg3(Alice, Bob, N) \to Spec', \\ Spec' &= send.msg3(Alice, Bob, N) \to Spec' \\ &\qquad \Box\ receive.msg3(Alice, Bob, N) \to Spec'', \\ Spec'' &= send.msg3(Alice, Bob, N) \to Spec''. \end{aligned}$$

This allows $Bob$ to receive a *single* message 3 from $Alice$ using $N$ after $Alice$ has sent such a message (in state $Spec'$); the specification does not prevent $Alice$ from sending further such messages, although these should not be accepted by $Bob$.

Let $Insts$ capture all instances of initiator and responder instances other than the principal responder. More precisely, $Insts$ is a multiset of values of the form $\textsc{Init}(a, na)$, representing an initiator with identity $a$ and using nonce $na$, and $\textsc{Resp}(b, nb)$, representing a responder with identity $b$ and using nonce $nb$. The nonces are generated freshly, so we will assume that none of the nonces that occurs within $Insts$ is equal to the principal nonce $N$. This is a fairly weak freshness guarantee, as it allows the same nonce to occur more than once within $Insts$; it turns out that, as far as the satisfaction of the property in question is concerned, it doesn't matter if other nonces aren't unique.

Let process $Rest(Insts)$ be the corresponding CSP process, formed as an interleaving of appropriate $Initiator(a, na)$ and $Responder(b, nb)$ processes:

$$\begin{aligned} Rest(Insts) \mathrel{\widehat{=}} & \left( \mathop{|||} \textsc{Init}(a, na) : Insts \bullet Initiator(a, na) \right) \\ & ||| \\ & \left( \mathop{|||} \textsc{Resp}(b, nb) : Insts \bullet Responder(b, nb) \right). \end{aligned}$$

We now sketch how one can build a CSP model of the most general intruder who can interact with the protocol. We model the way the intruder can produce new messages from the messages he already knows as a set $Productions$ of pairs of the form $(f, X)$, representing that $f$ can be produced from the set of messages $X$. For example, for each message $f$ in $fs$, $Productions$ will contain $(f, \{Encrypt.(k, fs), inverse(k)\})$ representing that the intruder can learn new messages by decrypting encrypted messages with known keys.

We will need the following definition from [20] later:

**Definition 3.1** A production system is *positive* with respect to a type parameter $T$ if for every function $\phi : T_1 \to T_2$, if $(f, X)$ is a production that the system generates for type $T_1$, then $(\phi(f), \phi(X))$ is a production that the system generates for type $T_2$.

This essentially requires that the production system is symmetric in $T$ (i.e. it treats all members of $T$ equivalently), and never depends upon the inequality of two members of $T$ that appear in the second component of the production. The standard production system has this property.

One can build a CSP process $Intruder(S)$ that models an intruder who knows the set of messages $S$. The process can: (1) overhear messages passed between honest agents, and add those messages to its knowledge; (2) send messages to honest agents, possibly pretending to be from some other agent; (3) produce new messages from the messages already known, according to the production rules. See [14] for details. The intruder's knowledge is initialised to

$$Agent \cup \{n \mid n \in Nonce, n \neq N\} \cup \{PK(a) \mid a \in Agent\} \cup$$
$$\{SK(a) \mid a \in Agent, a \neq Alice \wedge a \neq Bob\}.$$

Note we give the intruder all secret keys other than those of $Alice$ and $Bob$, and all nonces other than $N$; clearly this gives the intruder more abilities than not having those keys and nonces.

In [20], Roscoe and Broadfoot prove that the standard CSP model of an intruder satisfies $\mathsf{PosConjEqT}'_{\mathsf{C}}$ provided: (1) it uses a positive production system (see Definition 3.1); and (2) its initial knowledge set is built using no equality tests with members of $T$ other than the constants $C$. We have given the intruder these extra keys and nonces so that condition (2) is satisfied.

We can therefore define the model of the complete system as follows:

$$System(Agent, Nonce, Insts) =$$
$$((Responder(Bob, N) \mid\mid\mid Rest(Insts)) \parallel Intruder) \setminus (\Sigma - \alpha Spec),$$

where $\alpha Spec$ is the alphabet of $Spec$,

$$\alpha Spec = \{send.msg3(Alice,\ Bob,\ N),\ receive.msg3(Alice, Bob, N)\}.$$

Note that the honest agents cannot communicate directly with one another; instead, all communications go via the intruder, corresponding to the assumption that the intruder has complete control over the network.

We want to prove that

$$\forall\, Agent, Nonce, Insts \bullet Spec \sqsubseteq System(Agent, Nonce, Insts).$$

Below, in the interests of brevity, we leave implicit the parameters of processes such as $System$ where they are clear from the context (indeed, above we left implicit the parameters $Agent$ and $Nonce$ of the components of the system).

## 4 Building the abstraction

In this section we build a suitable abstraction of the system. What we want to do is find a process $Abstraction(Agent, Nonce)$ that is an abstraction of the system, and refines the specification:

(3) $\mathit{Abstraction}(\mathit{Agent}, \mathit{Nonce}) \sqsubseteq \mathit{System}(\mathit{Agent}, \mathit{Nonce}, \mathit{Insts})$,

(4) $\mathit{Spec} \sqsubseteq \mathit{Abstraction}(\mathit{Agent}, \mathit{Nonce})$,

for all types $\mathit{Agent} \supseteq \{\mathit{Alice}, \mathit{Bob}\}$ and $\mathit{Nonce} \supseteq \{N\}$, and for all compatible $\mathit{Insts}$. We will try to ensure that property (3) is true by the way we construct *Abstraction*; we will verify it later. We will use FDR to test whether property (4) is satisfied when we choose small values for the types:

(5) $\mathit{Spec} \sqsubseteq \mathit{Abstraction}(\mathit{Agent}_0, \mathit{Nonce}_0)$,

where

$$\mathit{Agent}_0 = \{\mathit{Alice}, \mathit{Bob}, \mathit{Mallory}\}, \qquad \mathit{Nonce}_0 = \{N, \mathit{Nm}\}.$$

We will then use data independence techniques to extend equation (5) to all values of the types.

We produce solutions to equations (3) and (5) using a technique inspired by Clarke et al.'s technique of counter-example guided abstraction refinement [5]. We will consider a sequence of processes $\mathit{Rest}_i$ for $i = 1 \mathinner{.\,.} 4$, each independent of $\mathit{Insts}$, such that $\mathit{Rest}_i \sqsubseteq \mathit{Rest}(\mathit{Insts})$ for all $\mathit{Insts}$. In fact we will have

$$\mathit{Rest}_1 \sqsubseteq \mathit{Rest}_2 \sqsubseteq \mathit{Rest}_3 \sqsubseteq \mathit{Rest}_4 \sqsubseteq \mathit{Rest}(\mathit{Insts}),$$

so each $\mathit{Rest}_i$ is a better approximation to $\mathit{Rest}(\mathit{Insts})$. For each $i$ define:

$$\begin{array}{l} \mathit{Abstraction}_i(\mathit{Agent}, \mathit{Nonce}) = \\ \qquad ((\mathit{Responder}(\mathit{Bob}, N) \mathbin{|||} \mathit{Rest}_i) \parallel \mathit{Intruder}) \setminus (\Sigma - \alpha \mathit{Spec}). \end{array}$$

Then by monotonicity of the CSP operators with respect to refinement, we have $\mathit{Abstraction}_1 \sqsubseteq \mathit{Abstraction}_2 \sqsubseteq \mathit{Abstraction}_3 \sqsubseteq \mathit{Abstraction}_4 \sqsubseteq \mathit{System}$. In particular, each $\mathit{Abstraction}_i$ will satisfy equation (3) (for *Abstraction*). We will then test whether $\mathit{Spec} \sqsubseteq \mathit{Abstraction}_i$ (initially using FDR to test equation (5)); if this is the case, we can take $\mathit{Abstraction} = \mathit{Abstraction}_i$; if not, we will use the counterexample returned by FDR to help produce the next approximation $\mathit{Rest}_{i+1}$ and corresponding $\mathit{Abstraction}_{i+1}$.

In each case, we choose the $\mathit{Rest}_i$ process such that we'll be able to apply data independent techniques.

### *4.1 First attempt*

As a first attempt, take $\mathit{Rest}_1$ to be the process that can perform *any* send or receive of a protocol message.

$$\begin{array}{l} \mathit{Rest}_1 = \mathit{Chaos}(\{\mathit{send}.\mathit{msg}1(a, b, \mathit{na}),\ \mathit{receive}.\mathit{msg}1(a, b, \mathit{na}), \\ \qquad \mathit{send}.\mathit{msg}2(b, a, \mathit{na}, \mathit{nb}),\ \mathit{receive}.\mathit{msg}2(b, a, \mathit{na}, \mathit{nb}), \\ \qquad \mathit{send}.\mathit{msg}3(a, b, \mathit{nb}),\ \mathit{receive}.\mathit{msg}3(a, b, \mathit{nb}) \mid \\ \qquad\quad a \in \mathit{Agent}, b \in \mathit{Agent}, \mathit{na} \in \mathit{Nonce}, \mathit{nb} \in \mathit{Nonce}\}). \end{array}$$

Certainly this is an abstraction of *Rest* because it allows all traces over the alphabet, whereas *Rest* just allows certain traces. Note also that the system built using this component is independent of the value of *Insts*.

However, we can use FDR to discover that $Abstraction_1$ is too coarse an approximation, and so doesn't refine *Spec*, for it allows traces where the intruder learns the nonce $N$ from events of $Rest_1$, and then uses that nonce to fake messages in runs of the protocol with the $Responder(Bob, N)$ process. These runs would not have counterparts in *System*.

### 4.2 Second attempt

We now produce a process $Rest_2$ that avoids the problems of $Rest_1$. What we need to do is restrict the sending of messages involving the nonce $N$, to avoid the intruder learning that nonce in circumstances where he would not learn it when interacting with the process *Rest*.

For convenience, define

$$OtherNonce = Nonce - \{N\}.$$

We start by defining a process that allows all sends or receives of messages that do not use the nonce $N$, as follows:

$$\begin{aligned}
&Rest_not_N = \\
&\quad Chaos(\{send.msg1(a, b, na),\ receive.msg1(a, b, na), \\
&\qquad send.msg2(b, a, na, nb),\ receive.msg2(b, a, na, nb), \\
&\qquad send.msg3(a, b, nb),\ receive.msg3(a, b, nb) \mid \\
&\qquad\quad a \in Agent, b \in Agent, \\
&\qquad\quad na \in OtherNonce, nb \in OtherNonce\}).
\end{aligned}$$

Now consider the *send* messages that make use of $N$. The process *Rest* will never send a message 1 using $N$: each initiator chooses the value of *na* freshly, and so will not choose the same value $N$ as the principal responder; this is reflected by the assumption that none of the nonces within *Insts* is equal to $N$. Similarly, the process *Rest* will never send a message 2 using $N$ to instantiate *nb*, because each responder chooses the value of *nb* freshly. Hence we define $Rest_2$ such that it never sends such messages.

It is possible that *Rest* could send a message 2 in which $N$ instantiates *na*. For this to happen, the corresponding responder component must have received a message 1 with $N$ instantiating *na*. We therefore require a process that, after a receive of such a message 1, allows a send of the corresponding message 2. We actually use a process that allows slightly more events: the process $Rest\_na\_as\_N$ will send *arbitrarily many* such instances of message 2, with *na* as $N$, once it has received a single such message 1, and also allow further receives of such a message 1. (A process that allows precisely those sends for which there has been a receive of the corresponding message 1 would have to keep track of the number of receives, and so would be infinite state. A process that allows arbitrary many sends once there has been a receive of the corresponding message 1 would have to keep track of the set of nonces received in instances of message 1; this is not a data independent construct.)

$$\begin{aligned}
&\mathit{Rest_na_as_N} = \\
&\qquad \Box\, a : \mathit{Agent}, b : \mathit{Agent} \bullet \mathit{receive.msg}1(a, b, N) \rightarrow \\
&\qquad \mathit{Chaos}(\{\mathit{send.msg}2(b', a', N, nb),\ \mathit{receive.msg}1(a', b', N) \mid \\
&\qquad\qquad\qquad a' \in \mathit{Agent}, b' \in \mathit{Agent}, nb \in \mathit{OtherNonce}\}).
\end{aligned}$$

Similarly, it is possible that $\mathit{Rest}$ could send a message 3 in which $N$ instantiates $nb$. For this to happen, the corresponding initiator component must have received a message 2 with $N$ instantiating $nb$. We again use a process that allows more traces than that; the process $\mathit{Rest\_nb\_as\_N}$ will send arbitrarily many such instances of message 3, and also allow further receives of such instances of message 2, once it has received a single such message 2.

$$\begin{aligned}
&\mathit{Rest_nb_as_N} = \\
&\qquad \Box\, b : \mathit{Agent}, a : \mathit{Agent}, na : \mathit{OtherNonce} \bullet \\
&\qquad\qquad \mathit{receive.msg}2(b, a, na, N) \rightarrow \\
&\qquad \mathit{Chaos}(\{\mathit{send.msg}3(a', b', N),\ \mathit{receive.msg}2(b', a', na', N) \mid \\
&\qquad\qquad\qquad a' \in \mathit{Agent}, b' \in \mathit{Agent}, na' \in \mathit{OtherNonce}\}).
\end{aligned}$$

We combine the above components, and define

$$\mathit{Rest}_2 = \mathit{Rest\_not\_N} \mid\mid\mid \mathit{Rest\_na\_as\_N} \mid\mid\mid \mathit{Rest\_nb\_as\_N}.$$

This is an abstraction of $\mathit{Rest}$ because it allows every event when $\mathit{Rest}$ would have allowed the same event.

This is still too much of an over-abstraction. FDR detects a false attack where the intruder imitates the initiator Alice in a run of the protocol with the principal responder process: the principal responder sends the message $\mathit{msg}2(\mathit{Bob}, \mathit{Alice}, \mathit{Nm}, N)$; the intruder can then cause the process $\mathit{Rest\_nb\_as\_N}$ to receive this, causing it to perform the event $\mathit{send.msg}3(\mathit{Alice}, \mathit{Mallory}, N)$, from which Mallory can extract $N$, so as to fake the final message 3 to Bob: $\mathit{receive.msg}3(\mathit{Alice}, \mathit{Bob}, N)$.

The above attack is a false attack, in the sense that it is a behaviour not allowed by the $\mathit{Rest}$ process. The problem lies with $\mathit{Rest\_nb\_as\_N}$: it allows events $\mathit{receive.msg}2(b, a, na, N)$ and $\mathit{send.msg}3(a', b', N)$ with $b \neq b'$, whereas the corresponding trace in $\mathit{Rest}$ would have $b = b'$. We correct this below.

### 4.3 Third attempt

We split the process $Rest\_nb\_as\_N$ from the previous section into two processes: the first, $Rest\_nb\_as\_N\_Bob$, dealing with the case $b = Bob$, and ensuring in this case that $b' = Bob$, also; and the second, $Rest\_nb\_as\_N\_other$, dealing with the case $b \neq Bob$, and ensuring in this case that $b' \neq Bob$, also (it turns out not to be necessary to ensure $b = b'$ in this case; to do so would lead to an infinite state or non-data independent process). We omit details because of lack of space.

FDR again finds a false attack on this system. As in the previous section, the intruder imitates the initiator Alice in a run of the protocol with the principal responder process, and the responder sends $msg2(Bob, Alice, Nm, N)$. The intruder causes the $Rest\_nb\_as\_N\_Bob$ process to receive this, which now performs the event $send.msg3(Bob, Bob, N)$ (the first "$Bob$" could be replaced by any identity other than $Alice$ for this false attack to work). The intruder can then use this message to produce $msg3(Alice, Bob, N)$ and send it to the principal responder process.

The problem here is with the process $Rest\_nb\_as\_N\_Bob$: it allows events $receive.msg2(Bob, a, na, N)$ and $send.msg3(a', Bob, N)$ with $a \neq a'$, whereas the corresponding trace in $Rest$ would have $a = a'$. We correct this in the next section. The effect of the correction on the above attack would be that the $send.msg3(Bob, Bob, N)$ event would become $send.msg3(Alice, Bob, N)$, which is the first event performed by the specification process $Spec$; after this event, the specification allows the principal responder to perform the corresponding $receive.msg3(Alice, Bob, N)$ event, so the trace would not constitute an attack.

### 4.4 Fourth attempt

We split the process $Rest\_nb\_as\_N\_Bob$ from the previous section into two processes: one dealing with the case $a = Alice$, and ensuring in this case that $a' = Alice$; and one dealing with the case $a \neq Alice$, and ensuring in this case that $a' \neq Alice$. We again omit the details because of lack of space.

FDR then confirms that the appropriate refinement holds. More precisely, if we take

$$
\begin{aligned}
&Abstraction(Agent, Nonce) = \\
&\qquad ((Responder(Bob, N) \mathbin{|||} Rest_4) \parallel Intruder) \setminus (\Sigma - \alpha Spec),
\end{aligned}
$$

then FDR confirms that the refinement of equation (5) holds when we take $Agent = Agent_0$ and $Nonce = Nonce_0$. We extend this to arbitrary choices of $Agent$ and $Nonce$ in the next section.

## 5 Verifying the refinements

In this section, we begin by verifying that the abstraction refines the specification for *all* values of the types. We then prove that the claimed abstraction really is an abstraction of the system model. In both cases, we rely heavily on data independence techniques.

### 5.1 Generalising the types of agents and nonces

Firstly note that the CSP models we have built are data independent. Also note that the CSP models we have built of honest agents, and their abstractions, satisfy $\mathsf{PosConjEqT}'_{\{\mathsf{Alice},\mathsf{Bob}\}}$ for $Agent$, and $\mathsf{PosConjEqT}'_{\{\mathsf{N}\}}$ for $Nonce$. We have already noted that the intruder process satisfies these conditions.

We now use Theorem 2.2 to show that we can extend the refinement of equation (5) to all values of the types $Agent$ and $Nonce$. First let's deal with agent identities. Let $\phi_1$ be the function that maps all names other than $Alice$ and $Bob$ to $Mallory$; so in particular $\phi_1(Agent) = Agent_0$. Then Theorem 2.2 tells us that

$$\{\phi_1(tr) \mid tr \in traces(Abstraction(Agent, Nonce))\} = \\ traces(Abstraction(Agent_0, Nonce)).$$

However, in $Abstraction$ all events using names other than $Alice$ and $Bob$ are hidden, so the application of $\phi_1$ in the above equation has no effect; hence

$$traces(Abstraction(Agent, Nonce)) = \\ traces(Abstraction(Agent_0, Nonce)).$$

Now let's deal with the nonces. Let $\phi_2$ be the function that maps all nonces other than $N$ to $Nm$. Then, similarly to above, we can show

$$traces(Abstraction(Agent_0, Nonce)) = \\ traces(Abstraction(Agent_0, Nonce_0)).$$

Combining the above steps

$$traces(Abstraction(Agent, Nonce)) = \\ traces(Abstraction(Agent_0, Nonce_0)).$$

We already know that

$$Spec \sqsubseteq Abstraction(Agent_0, Nonce_0).$$

Hence we can deduce that

$$Spec \sqsubseteq Abstraction(Agent, Nonce),$$

for all values of $Agent$ and $Nonce$; thus we have proved equation (4).

### 5.2 Verifying the abstraction

We now seek to prove that equation (3) holds, i.e. that the claimed abstraction really is an abstraction of the system model. Because refinement is a pre-congruence with respect to the CSP operators, it will be enough to prove the corresponding result for the models of the other agents:

$$Rest_4(Agent, Nonce) \sqsubseteq Rest(Insts, Agent, Nonce), \tag{6}$$

for all values of $Agent$ and $Nonce$ and corresponding $Insts$ (we have made the type parameters explicit for convenience). We can reduce this to proving the following, for all $Agent$ and $Nonce$:

$$Rest_4(Agent, Nonce) \sqsubseteq Rest_4(Agent, Nonce) \mid\mid\mid Rest_4(Agent, Nonce), \tag{7}$$

$$Rest_4(Agent, Nonce) \sqsubseteq \bigsqcap a : Agent, na : Nonce \bullet Initiator(a, na), \tag{8}$$

$$Rest_4(Agent, Nonce) \sqsubseteq \bigsqcap b : Agent, nb : Nonce \bullet Responder(b, nb). \tag{9}$$

Then repeated application of equation (7) will show that $Rest_4$ is refined by

$$\Big|\Big|\Big|\, x : Insts \bullet Rest_4(Agent, Nonce),$$

(i.e. $\#(Insts)$ copies of $Rest_4$). Then equations (8) and (9) can be used to refine each $Rest_4$ to the appropriate initiator or responder process, verifying equation (6).

We can verify equations (7)–(9) using Theorem 2.1. Note that $Rest_4$ satisfies the condition Norm (essentially because the components have disjoint alphabets). Hence we can verify the refinements for types of sizes up to the thresholds given by Theorem 2.1, and deduce the result for all larger types [2].

For equation (7), the bound for $Agent$ is 6 ($W^{Spec} = 2$, $W^{Impl} = 2$, $L^{Impl} = 2$), and the bound for $Nonce$ is 4 ($W^{Spec} = 1$, $W^{Impl} = 1$, $L^{Impl} = 2$). In fact, to make the check faster, we test a similar condition for each of the four components of $Rest_4$ (for example, we show $Rest\_not\_N \sqsubseteq Rest\_not\_N \mid\mid\mid Rest\_not\_N$), and then deduce the result from the fact that refinement is a pre-congruence.

For equations (8) and (9), the bound for $Agent$ is 5 ($W^{Spec} = 2$, $W^{Impl} = 2$, $L^{Impl} = 1$) and the bound for $Nonce$ is 4 ($W^{Spec} = 1$, $W^{Impl} = 2$, $L^{Impl} = 1$). In fact, to make the checks faster, we replace the left hand sides of the refinements by just those components of $Rest_4$ that correspond to initiator actions or responder actions, respectively.

## 6 Conclusions

In this paper we have described an approach to the parameterised model checking problem, and illustrated it on a well-known security protocol. We

[2] The processes in question make use of constants, which are not directly dealt with by Theorem 2.1; however, constants can be treated as being input in the initial state of the processes when calculating the thresholds [12].

have produced an abstraction of the system model, and shown that the abstraction refines the specification process, for all instances of honest agents and for all types. The abstraction was produced using a technique based on Clarke et al.'s technique of counterexample-guided abstraction refinement. Proof requirements were carried out using a combination of model checking with FDR, and data independence techniques.

The approach appears very practical: the case study in this paper took about two days' work; I expect that with the benefits of the lessons learnt from this example, subsequent such analyses would be much faster. Most of the FDR checks take a matter of seconds.

A couple of tricks were used during the construction of the abstraction to allow the data independence theorems to be applied. The components of the $Rest_i$ processes allowed more traces than one would have expected, by not distinguishing between nonces other than $N$, and not distinguishing between agent identities other than *Alice* and *Bob*: to do otherwise would have required an interleaving of processes indexed by nonces or identities, which is not a data independent construct. We gave the intruder more knowledge than one might have expected, so that the process satisfies $\mathsf{PosConjEqT}'_{\{\mathsf{Alice},\mathsf{Bob}\}}$ and $\mathsf{PosConjEqT}'_{\{\mathsf{N}\}}$, allowing Theorem 2.2 to be applied.

The parameterised model checking problem has been the subject of several previous works. The problem is undecidable in general [1]. However, various approaches have succeeded in specific cases.

Kurshan and McMillan [11], and Wolper and Lovinfosse [23] show how to verify a parameterised system by induction on the number of components, with both the base case and inductive step verified automatically. Creese and Roscoe [6,7] similarly code up induction as a sequence of CSP refinement checks, and use data independence to widen the range of applicability of such techniques to settings where the behaviour of individual components depends upon the number of components in the system. Clarke et al. [4] use network grammars to define families of systems, regular languages to capture state properties, and abstraction to construct an invariant that simulates all systems in a family. However, in all such approaches considerable ingenuity is required to produce an appropriate inductive hypothesis. Emerson and Namjoshi [9], and Emerson and Kahlon [8] produce results that, in some circumstances, can be used to reduce model checking of systems of arbitrary size to model checking of systems of sizes up to a small cutoff size.

There have been a few previous attempts to address the parameterised model checking problem in the context of security protocols. In [16], I produced conditions on protocols under which model checking of a small system running a protocol is sufficient to decide the security of the protocol in arbitrary systems; however, the conditions were fairly restrictive. Broadfoot and Roscoe [3] model actions of honest agents other than the principal instances within the intruder, and hence argued, using data independent techniques, that in some cases this model captured all attacks that would be possible in arbitrary systems; however, the technique is not fully general, and the correctness arguments can be quite complex. The NRL Protocol Analyzer [17] uses theorem proving techniques to narrow the state space, by proving the unreachability of infinite classes of states, and hence is often able to analyse arbitrary systems running a protocol. Song et al. [21] verify protocols using the Strand Space Model [22]; they use symbolic techniques to reduce the search space to a finite space. Blanchet [2] verifies protocols using a technique based on Prolog rules; in order to produce a finite model, he performs two abstractions, both of which have similarities to some of our abstractions: he allows the same value to be used multiple times for a "fresh" variable if all preceding values in the runs have been identical; and he abstracts away from the notion of run, by allowing protocol steps to be performed multiple times if all preceding steps have occurred.

Alex Hillel, as his final year undergraduate project, is applying these techniques to more example security protocols. Our early impression is that the application of the technique differs little from one example to another. As a result of this, we believe there is considerable scope for improving the level of automation, at least as applied to security protocols.

As noted above, the standard model of security protocols is produced using the CSP compiler Casper [15]. We believe that it would be very feasible to extend Casper to help in the production of the abstractions:

- The first abstraction was simply *Chaos* over the appropriate alphabet, which is straightforward to produce automatically;
- The second abstraction allowed the sending of particular messages only when the previous message had been received; this abstraction is again straightforward to produce from the protocol description;
- Subsequent abstractions were produced by "splitting" on a particular field: for example, the third abstraction was produced by splitting into the cases $b = Bob$ and $b \neq Bob$; it would seem straightforward to allow the user to specify the split, and for Casper to produce the definition; whether it is possible to calculate the split automatically, given the previous counterexample, is less clear at present.

We believe that the data independent arguments, as in Section 5, are effectively the same in all such cases. We believe that it would be possible to generalise those arguments, to produce a theorem that covers all security protocol examples.

We intend to undertake further experiments to assess the generality of this technique, to examples other than security protocols. Can we identify classes of problem for which this technique is easier—or less easy—to apply than existing techniques? Can we extend these techniques from safety properties to liveness properties (we see no major difficulties in this)? Can we develop techniques for automatically, or semi-automatically, generating the abstractions of the system in all cases?

*Acknowledgements*

I would like to thank: Bill Roscoe and Ranko Lazić for helpful discussions about data independence; Alex Hillel for helping to investigate the applicability of these techniques; and Christie Bolton for presentational suggestions and encouragement. This work was partially supported by grants from the UK Engineering and Physical Sciences Research Council, and the Ministry of Defence through QinetiQ.

## References

[1] K. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 15, 1986.

[2] Bruno Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, pages 82–96, 2001.

[3] P. J. Broadfoot and A. W. Roscoe. Capturing parallel attacks within the data independence framework. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop*, 2002.

[4] E. M. Clarke, O. Grumberg, and S. Jha. Verifying parametrized networks using abstraction and regular languages. In *6th International Conference on Concurrency Theory (CONCUR'95)*, 1995.

[5] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification (CAV'00)*, 2000.

[6] S. J. Creese and A. W. Roscoe. Verifying an infinite family of inductions simultaneously using data independence and FDR. In *Formal Methods for Protocol Engineering and Distributed Systems, the proceedings of Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification (FORTE/PSTV'99)*, 1999.

[7] S. J. Creese and A. W. Roscoe. Data independent induction over structured networks. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '00)*, 2000.

[8] E. Allen Emerson and Vineet Kahlon. Reducing model checking of the many to the few. In *Proceedings of the Conference on Automated Deduction (CADE)*, 2000.

[9] E. Allen Emerson and Kedar S. Namjoshi. Reasoning about rings. In *Proceedings of the Symposium on Principles of Programming Languages (POPL '95)*, 1995.

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

[11] R. P. Kurshan and K. McMillan. A structural induction theorem for processes. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*, 1989.

[12] Ranko Lazić. *A Semantic Study of Data Independence with Applications to Model Checking*. D.Phil., Oxford University, 1999.

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

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

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

[16] Gavin Lowe. Towards a completeness result for model checking of security protocols. *Journal of Computer Security*, 7(1,2), 1999.

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

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

[19] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.

[20] A. W. Roscoe and P. J. Broadfoot. Proving security protocols with model checkers by data independence techniques. *Journal of Computer Security*, 7(2, 3):147–190, 1999.

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

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

[23] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Machines*, volume 407 of *LNCS*, 1989.