Automating Data Independence

P. J. Broadfoot[1], G. Lowe[2], and A. W. Roscoe[1⋆]

[1] Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD, UK, {Philippa.Broadfoot, Bill.Roscoe}@comlab.ox.ac.uk.
[2] Department of Mathematics and Computer Science, University of Leicester, University Road, Leicester, LE1 7RH, UK, gavin.lowe@mcs.le.ac.uk.

Abstract. In this paper, we generalise and fully automate the use of data independence techniques in the analysis of security protocols, developed in [16, 17]. In [17], we successfully applied these techniques to a series of case studies; however, our scripts were carefully crafted by hand to suit each case study, a rather time-consuming and error-prone task. We have fully automated the data independence techniques by incorporating them into Casper, thus abstracting away from the user the complexity of the techniques, making them much more accessible.

Keywords: security protocols; data independence; automatic verification; model checking; Casper; CSP; FDR.

1 Introduction

Model checkers have been extremely effective in finding attacks on security protocols: see, for example, [9, 12–14]. However, until a few years ago, their use in *proving* protocols had 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.

In [16], Roscoe developed techniques borrowed from data independence and related fields to simulate, using the process algebra CSP [15], a system where agents can call upon an infinite supply of different nonces, keys, etc., even though the actual types remain finite. It was thus possible to create models of protocols in which agents could perform an unbounded number of sequential runs—although with a fixed, finite number of *concurrent* runs—and therefore claim that a finite-state check on a model checker, such as FDR [2], proved that a given protocol was free from attacks—subject to the same limit on the number of concurrent runs. These methods made it possible to prove far more complete results on model checkers than hitherto.

Our data independence techniques were further developed and successfully applied to a series of case studies in [17]; however, our scripts were carefully

⋆ The authors' received support from DERA Malvern and the US Office of Naval Research for the work reported in this paper.

crafted by hand to suit each case study. This process proved to be time-consuming, error-prone and required a substantial knowledge of the CSP language and a good understanding of the theory underlying the data independence techniques.

In this paper we present the automation of our data independence techniques, by integrating them into Casper. Casper [10] is a compiler which takes a more abstract description of a security protocol (similar to that in the literature) and a specification of the properties to be verified; it automatically generates the CSP description, which can then be loaded directly into FDR.

Further, we present some generalisations of our methods, which allow a broader class of protocols to be modelled.

This work has already proved to be very useful for generating our CSP protocol models, since it only takes Casper a couple of seconds to perform the task, which in the past took us several weeks per protocol model.

2 Background

2.1 Modelling Security Protocols Using CSP and FDR

Security protocols are traditionally described in the literature as a series of messages between the various legitimate participants. The main example we will be using throughout this paper is a version of the Yahalom protocol closely based on the one suggested in [1]. Its five messages are:

$$
\begin{array}{lll}
\text{Message 1.} & A \to B & : N_a \\
\text{Message 2.} & B \to S & : N_b, \{A, N_a\}_{SKey(B)} \\
\text{Message 3.} & S \to A & : N_b, \{B, K_{ab}, N_a\}_{SKey(A)} \\
\text{Message 4.} & S \to B & : N_b, \{A, K_{ab}, N_b\}_{SKey(B)} \\
\text{Message 5.} & A \to B & : \{N_b\}_{K_{ab}}
\end{array}
$$

The only difference from the version in [1] is the way in which the server communicates message 4 directly to B rather than using A as a messenger. This type of re-direction has been shown in [3] to have no effect on security.

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. This particular version has a well known attack found by Syverson [19]. However, we will use it as our example throughout the paper to illustrate and clarify the techniques introduced.

We analyse security protocols using the process algebra CSP [15] and its model checker FDR [2]. We outline the traditional approach—without the application of data independence techniques—below. Further details can be found in [9, 12].

Each honest participant of the protocol is modelled as a CSP process. These processes are relatively straightforward to derive from standard description of protocols presented in the literature.

The intruder is also modelled as a CSP process. This intruder can: overhear all messages that pass between the honest agents; prevent a message sent by one agent from reaching the intended recipient; generate new messages from those messages held initially or subsequently overheard, subject only to derivation rules that reflect the properties of the crypto-system in use; and send such messages to any agent, purporting to be from any other. We only allow the intruder to generate messages of size bounded by the message structure of the protocol, so we do not consider messages that do not correspond to part of a genuine protocol message. This is a standard assumption and one which can be justified in many cases, but it should be borne in mind that as with various points of our modelling, all our results are relative to it.

A system is created from a parallel combination of honest agents, together with an intruder. The system normally—but not necessarily—contains as many agents as are necessary for a complete run of the protocol (in the case of our example above, this would be two agents and one server).

One then seeks to show that any session that may occur between the honest nodes is secure, no matter what the actions of the intruder 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 (see [11] for further details concerning specifications).

2.2 Casper

Model checkers (in our case, FDR) have proved to be extremely effective in checking for, and finding, attacks on security protocols. However, the process of creating the CSP protocol models is time-consuming, error-prone and requires a substantial knowledge of the CSP language. Casper [10] is a program which takes a more abstract description of a protocol and generates the corresponding CSP description. The CSP output file is such that it can be loaded directly into FDR, and the requested checks upon the protocol automatically tested. The style of the protocol descriptions in a Casper input file is based on that in the literature and is therefore familiar to users who are interested in modelling them. Casper has proved to be an extremely useful tool for generating these scripts and is accessible to a wide audience of users.

Figure 1 presents an example of a simple Casper script, namely the Yahalom protocol example we are using throughout this paper. There are two main parts to a Casper input script: a definition of the way in which the protocol operates (the first four sections); and a definition of the actual system to be verified (the last four sections).

```
#Free Variables

A, B : Agent
S : Server
na, nb : Nonce
kab : SessionKey
SKey : Agent -> ServerKey
InverseKeys = (SKey, SKey), (kab, kab)
```

#Free Variables declares all variables and their corresponding types, used in defining the first part of the Casper script (in this case, agent processes, protocol description and specifications) together with inverse keys.

```
#Processes

INITIATOR(A, na) knows SKey(A)
RESPONDER(B, S, nb) knows SKey(B)
SERVER(S, kab) knows SKey
```

#Processes declares all the agent processes taking part. The first parameter of each process represents their identity.

```
#Protocol description

0.    -> A : B
1.  A -> B : na
2.  B -> S : nb, {A, na}{SKey(B)}
3.  S -> A : nb, {B, kab, na}{SKey(A)}
4.  S -> B : nb, {A, kab, nb}{SKey(B)}
5.  A -> B : {nb}{kab}

# Specification

Agreement(B, A, [])
```

#Specification defines the requirements of the protocol to be verified.

```
#Actual Variables

Alice, Mallory : Agent
Sam : Server
Na, Nb, Nm : Nonce
Kab : SessionKey
InverseKeys = (Kab, Kab)
```

#Actual Variables declares all variables and their corresponding types, used in defining the actual system to be verified.

```
#Functions

symbolic SKey
```

"symbolic" means that Casper will produce its own values to represent the results of the function applications.

```
#System

INITIATOR(Alice, Na)
RESPONDER(Alice, Sam, Nb)
SERVER(Sam, Kab)
```

#System defines the actual system to be verified. In this case, our system is composed of one instance of each agent process running in parallel with each other (where each process can only perform one run).

```
#Intruder Information

Intruder = Mallory
IntruderKnowledge = {Alice, Sam, Mallory, Nm, SKey(Mallory)}
```

Intruder's identity within the system.

Intruder's initial knowledge.

Fig. 1. Example of a Casper script for the Yahalom Protocol

2.3 Data Independence Techniques

A program P is 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, P can input and output members of T, hold them as parameters, compare them for equality, and apply *polymorphic* operations to them such as tupling and list forming. It may not apply other operations such as functions that return members of T, or comparison operators such as $<$, or do things like compute the size of T. For further details and precise definitions, see [4, 5, 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. If we can verify a system for the threshold size of T, then we will have verified it for all larger values of T. This can be done successfully for a wide range of problems, as is shown, for example, in [15, 6].

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 between building a construction such as

```
Sq.<Nb, Encrypt.(SKey(A), <B, Kab, Na>)>
```

(the representation in our data type of a typical message 3) and building a list or tuple over the objects `Nb`, `A`, `B`, `Kab` and `Na`.

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, because our program must at least implicitly carry an unbounded number of values in its state so it knows what to avoid next. The starting point for threshold calculations, in the context of equality tests, is the maximum number of values a process ever has to remember; so such calculations would give an infinite threshold in this case. 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 (presented in [16, 17]) 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 a corresponding system for all parameter values to a finite check.

In [16], Roscoe showed 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 sequential runs, and to claim that a finite-state run on a model checker has proved that a given protocol is free from attacks that could be constructed in the model used. One proviso is that these techniques will only detect attacks that can be formed from the given degree of parallelism, and there remains the possibility that attacks could exist with a higher degree of parallelism. These methods were developed further and successfully applied to a number of case studies in [17].

These techniques were based around *manager processes*. In brief, for each data independent type T (for example, nonces or keys), we create a manager process MAN_T that gives the illusion of generating an infinite supply of fresh values, known as *foreground* values, when requested by participants. The manager process monitors the network so that, at all times, it knows which fresh values are stored by which participants. When such a value is no longer stored by any honest participant—when each honest agent that knew the value has terminated or withdrawn from the protocol run—the value can be *recycled*. The manager process is responsible for triggering the recycling mechanism, whereby forgotten foreground values get replaced by *background* values within the intruder's memory; the mapped values can then be reused. It is usually sufficient to include two background values of each type in the system: one that the intruder knows, and one that he doesn't. Roscoe showed in [16] that this recycling mechanism does not lose any attacks.

Our results rely on the fact that the intruder cannot perform any deductions based on the inequality of two objects with the same structures. Therefore, they only apply to intruders over a theory satisfying the *Positive Deductions* condition. What this essentially requires is that the generation of the deductions is symmetric in each data independent type T (i.e., treats all members of T equivalently) and never has an inequality requirement over members of T that appear on the left-hand side of a deduction (see [16] for further details).

3 Generalising Data Independence Techniques

In [17], these techniques were successfully applied to a number of case studies, the implementation being carefully crafted by hand to suit the protocol being modelled, in each case. All the case studies were for protocols obeying the following simple property:

> Each message M in the protocol introduces *at most* one variable V that is of a data independent type, and that is either *freshly* introduced by the sender or *newly acquired* by the receiver of M.

Message M might contain other variables $X_1, \ldots, X_n$ of data independent types in addition to V, as long as $X_1, \ldots, X_n$ are already known to the sender and receiver of M. This property simplifies the modelling strategy, but limits the class of protocols that can be modelled, because many protocols have multiple fresh variables introduced in one message, either of the same or different data independent types.

In this section, we present an overview of our new methodology for the application of these techniques. This makes them applicable to a broader spectrum of protocols, in particular, dropping the assumption above. It is also the key to how we fully automated their application within Casper (presented in Section 5).

3.1 Generalising Manager Processes

Previously, in [17], our manager processes were constructed with the assumption that the protocol models conformed to the property presented above; we refer to such managers as *one-dimensional*. This assumption makes the synchronisations between managers and monitored processes more straightforward, because a manager process does not have to handle multiple values in one synchronisation, and the managers collectively do not have to synchronise with each other.

We generalise the implementation of our techniques and abolish this limiting property above, by introducing *multi-dimensional* managers, as illustrated in the following example.

Example 1. Suppose the agent *Alice* is running some protocol and receives a message M in which she newly acquires two nonces and one key, where nonces and keys are data independent types:

$$... \rightarrow Alice : \{N_1, N_2, K_1\}_{K_2},$$

where key K_2 is assumed to be known to Alice already. Our *one-dimensional* managers could not handle this case. We require *multi-dimensional* managers where each manager process deals with *tuples* of values rather than a *single* value.

In this example, we would have two manager processes, MAN_N for nonces, and MAN_K for keys. MAN_N would have an event $told_N.X$ for every pair X of nonces. This message is subsequently renamed, so as to synchronise with the corresponding network message M, thereby allowing the manager to monitor multiple foreground values in a single synchronisation. Further, the key manager MAN_K will have its own event $told_K.Y$ for each 1-tuple Y of keys. The key manager must synchronise on the network message M, and hence will also synchronise with the nonce manager.

3.2 Stale Knowledge Generation

When applying these techniques, one runs into problems with the size of the state space of the protocol model. One reason for this is the gradual build up within the intruder of knowledge gathered from past protocol runs. All these messages have only background values in the places occupied by the data independent types, because all foreground values have been mapped to background values by the recycling mechanism.

If there are N such *stale* messages that the intruder might gain, then the effect is to multiply the total state space by as much as 2^N. To eliminate this problem, we observe that eventually the intruder can be expected to gain a complete set of this *stale knowledge* (with all values of data independent types mapped to background values), and can therefore exploit any such stale knowledge when attempting to perform attacks upon the protocol. So we simply calculate, at the start of a run, what the eventual set of stale knowledge ought to be and give this to the intruder as part of his initial knowledge. If done properly, this completely eliminates the state space explosion problem presented above.

In the case studies presented in [17], these sets of stale knowledge were calculated by hand. However, this proved to be error prone.

We have, therefore, automated the calculation of the stale knowledge by performing a series of *pre-run simulations* between combinations of agents, including cases where the intruder took some of the roles, taking care to distinguish those cases where the intruder does or does not know the values of data independent types included within the messages. These simulations do not affect the state space, since we do not perform any actual runs of the protocol. Instead, we make use of the existing computational methods that are used to calculate the intruder's initial knowledge from initially known values specified by the user, closed under the deduction rules. The simulations are performed by taking the set of messages that the intruder would see by observing a typical run, using the mechanism outlined above to calculate the closure under the deduction rules, and then mapping fresh values of data independent types to appropriate background values. Calculating these sets based on the model of the intruder allows us to automatically capture any subtleties within the intruder's specified behaviour.

4 Incorporating Honest Agents Within the Intruder

Together with stale knowledge, a second strategy for reducing the state space of data independent protocol checks was introduced in [17]. This was to incorporate the functionality of some participant of the protocol into the intruder, thereby removing the need for an additional process in the system. An agent modelled this way is referred to as an *internal agent*.

It is only possible to do this easily for processes whose messages (or state of mind measured in some other way) are not directly examined by the specification we are using. Since such specifications tend to concentrate on the honest agents we have, to date, only incorporated server processes into the intruder. Henceforth, we will assume that it is a server that is being treated as internal.

In brief, the functionality of a server is captured within the intruder by means of an additional set of deductions and generations that the intruder can perform. As presented in more detail in [17], we classify internal agents into two categories. The first contains those that do not introduce any fresh variables of a data independent type at any point during the protocol. Internal agents in this first category are straightforward to incorporate into the intruder process and are captured with appropriate deductions (as demonstrated in the case studies presented in [17]). The second category are those internal agents that introduce fresh variables of data independent types at some stage in the protocol runs. Obvious examples of these are servers introducing fresh session keys like the one in the Yahalom example.

Deductions performed by the intruder are usually modelled by pairs of the form (X, f), where X is a finite set of facts and f is a fact that it can construct if it knows the whole of X, for example $(\{\{M\}_K, K\}, M)$. The functionality of the first sort of internal agent can be captured with this type of deduction within the intruder: we get a deduction (X, f) if, after the agent is told the messages in X, it can be expected to emit f (where f will be functionally dependent on X). The example given in [17] was the server in the TMN protocol ([12]) whose function was to receive two messages $M1$ and $M3$ and construct a corresponding third message $M4$, where $M4$ only contains variables in $M1$ and $M3$ (thereby not introducing any fresh variables into the system). The relevant deductions were therefore all valid instantiations of $(\{M1, M3\}, M4)$.

A special sort of "deduction" is required to model situations like that in our example protocol where an internalised server creates fresh values and may then put such values into several related packets (in our example, the second parts of messages 3 and 4). These *generations* have the form (t, X, Y), where t is a non-empty sequence of the fresh objects being created, X is the set of inputs the agent we are modelling requires to trigger it to produce the fresh values, and Y is the set of facts generated, which contain the fresh values that are produced. In our example, X would consist of a message 2, t would be a single fresh key, and Y would contain the second parts of messages 3 and 4:

$$\begin{aligned} t &= \langle K_{ab} \rangle, \\ X &= \{N_b, \{A, N_a\}_{SKey(B)}\}, \\ Y &= \{\{B, K_{ab}, N_a\}_{SKey(A)}, \{A, K_{ab}, N_b\}_{SKey(B)}\}, \end{aligned}$$

where K_{ab} is a fresh foreground value. (The first parts of messages 3 and 4 are not linked to the fresh value, and would be the subject of ordinary deductions.)

The sequence t (though it is rare for it to be more than length 1) synchronises with the appropriate manager(s), and the set Y is functionally dependent on X and t (i.e., if we know what messages went into our server, and what fresh values it generated in response, then we know what messages it will output).

Servers that generate fresh values create the most significant theoretical problem in our work. Namely, how can we reasonably limit the intruder's appetite for fresh values obtained from the server (whether internal or external) since it has the capability of requesting any number it wishes? The intruder can do this, for example, by using the same set X of inputs to the server to generate many different Y's, each characterised by a different key. Furthermore, it can often build up a store of these values and later use them one at a time with the honest agents. For essentially this reason, the recycling mechanism used elsewhere cannot be applied to these multiple Y's held within the intruder.

The only way to keep the number of fresh values manageable (or even finite) is to prevent the intruder storing many fresh values for later use. In automatically generated CSP scripts we are severe on the intruder: we stop it acquiring a new set of fresh values until one of the honest agents is ready to receive such a value "from the server". This bounds the state space satisfactorily, but how do we know we have not thrown away some attacks?

We are not aware of any real protocol where this simple approach does lose an attack, but that is not good enough since our objective is to provide a proof. On the assumption that the set Y that comes back from the server does indeed depend only on X and t, it is clearly true that any Y that the intruder could have acquired early for much later use could also be acquired just in time. Therefore, if the only thing the intruder does with the members of Y is to hand them on to honest agents, it is of no advantage to him to possess them early. The complications arise because the intruder might be in a position to make deductions from having many server-generated messages that affect his ability to produce other messages in advance of telling an honest agent about the fresh values.

We sketch below a solution which is guaranteed not to lose any attacks under the following additional assumptions:

- the server creates only one value of one type (so the t of each generation is a sequence of length 1);
- no agent learns any more than one value of this type at a time.

To simplify the presentation, we assume also that there is only one type being generated by the server (for example, there are not some generations of keys and some of nonces).

The solution to this is not unlike the background values used to enable the recycling of foreground values. We add extra, dummy values into the type being generated. These have the special characteristic that they are not accepted as genuine by any honest process (so the latter will never accept any message involving one). The intruder can use these values itself like any others, in particular doing deductions involving them. The trick is that we allow the intruder to perform, at any time, a "generation" based on a valid input set X, but unless there is a space in a honest agent for a value of the given type, the result will always be based on a dummy value. If there is a space, the result may be either a real or dummy value.

We can then argue that any trace of communications that take place between the intruder and agents in a model where there is an unbounded supply of values of the given type (and there is no restriction on how many times generations can occur) can be reproduced (with appropriate recycling mappings that occur in our existing model) in this intruder. For any message M that the intruder can generate when there is an unbounded supply of our type, it is possible to arrange that all the messages from the server that he uses to produce M are obtained after the last protocol message before M ("just in time"). This behaviour can be mapped onto a behaviour of the reduced system by mapping to dummy values all the values generated in these deductions other than the one value v freshly told to a trustworthy agent (if there is one), and making v the fresh value that the manager is still allowed to deliver to the intruder.

This use of dummy values therefore loses no attacks. However, there is the possibility that this technique introduces false attacks. An example would be where the value being introduced is a key K, and one of the messages contains something encrypted under K that the intruder would not otherwise learn; representing K by the dummy value, which the intruder could learn from elsewhere, would allow the intruder to deduce the contents of the message, as a false attack. The solution is to use two dummy values: one that is created in circumstances where we would expect the intruder to learn it legitimately, and one that is created in other cases. However, a single value appears to suffice more frequently than in the analogous case of background values.

There is no need to manage these dummy values via the manager process. Their "generation" can, in fact, be performed using ordinary deductions.

5 Full Automation of Data Independence Techniques

In this section, we begin by presenting an improvement to the CSP protocol models generated by Casper, which reduce the size of the state space; using data independence techniques increases the size of the state space, so these techniques would prove impractical without this balancing reduction. We then discuss the full integration of the data independence techniques within Casper.

5.1 Optimisation of Casper-Generated Scripts

The CSP protocol models generated by Casper make use of *signal events* that reflect the states of mind and beliefs of the honest processes running the protocol; these signal events are then used to specify security properties.

For example, suppose we want to test whether a protocol authenticates an agent Alice to another agent Bob, and whether they agree upon the value of some key k; we do this by testing whether if Bob thinks that he has completed a protocol run with Alice using a particular key k, represented by a signal event `signal.Commit.Bob.Alice.k`, then Alice thinks she was running the protocol with Bob using the same key, represented by an event `signal.Running.Alice.Bob.k`[1]. See [11] for fuller details, for example concerning details of the specifications and the placement of signals.

Previously, the definitions of the honest agents generated by Casper would interleave these signal events with the events representing the messages of the protocol. Unfortunately, these additional signal events greatly increases the number of traces of each process, and so greatly increases the state space. To overcome this problem, we redesigned our models so that instead of interleaving the signals with message events, the signals were introduced at the top most level, via an appropriate renaming of message events of the overall system process.

A problem arises, however, if the message event does not contain all the information needed to construct the corresponding signal event. To overcome this

[1] We are simplifying the structure of the signal events slightly for ease of presentation.

problem, we extended the CSP data structure representing the protocol messages by adding an additional field containing all the information required. (This field has *no* influence on the flow of messages in the protocol runs.) So, in our previous example, the message 5 corresponding to the desired signal event becomes `(Msg5, Encrypt.(kab, <nb>), <na>)`, where the extra field `<na>` completes the set of required variables for our signal event.

This optimisation on Casper has led to a dramatic reduction in the state space sizes of the protocols we model and check using FDR. For example, the number of states explored for the adapted Needham Schroeder Public Key Protocol [9] model (with two instances of the initiator and two of the responder agents running in parallel) using the new version of Casper is 15,050 states as opposed to the old version requiring 425,734 states; the value for the new version incorporates an optimisation inspired by and similar to one of those described in [18], namely that the intruder never acts when a trustworthy can output.

5.2 Incorporating Data Independence Techniques into Casper

Figure 2 presents an example of a Casper script modelling the same Yahalom protocol used in Figure 1, but this time incorporating our data independence techniques. The differences are minimal: we have abstracted away the design and complexity of our data independence techniques from the users of Casper, making our techniques much more accessible.

In brief, the main extensions to our Casper script are as follows. Firstly, the user must indicate which data types are to be treated as data independent. Within the `#Processes` section, variables of such types are indicated using the `generates` keyword: values for such variables will be freshly supplied by the corresponding manager processes. For example, in Figure 2 nonces and session keys are regarded as data independent, so the corresponding variables `na`, `nb` and `kab` are indicated as being generated in this way; by contrast, in Figure 1 these variables were introduced as parameters of the processes, with the parameters being instantiated in the `#System` section.

In the `#Actual Variables` section we declare the actual variables which will be used in our actual system; further, we now need to classify all variables of data independent types as either `Foreground`, `KnownBackground` or `UnknownBackground` values.

In the case of the foreground values, the user needs to estimate how many values the manager will need; if too few values are given, FDR will give a trace error, indicating that the manager process ran out of fresh values, and so was unable to allocate another value; in this case, the user can simply edit the Casper script to include an extra value. It is advisable to declare the minimum necessary number of foreground values for each type, since increasing this number will cause the state space of the system to grow dramatically. Finding methods for calculating the number needed is the subject of current work.

For each data independent type, the user must declare exactly one value as a `KnownBackground` value, and one as a `UnknownBackground` value.

```
#Free Variables

A, B : Agent
S : Server
na, nb : Nonce
kab : SessionKey
SKey : Agent -> ServerKey
InverseKeys = (SKey, SKey), (kab, kab)

#Processes

INITIATOR(A) knows SKey(A) generates na
RESPONDER(B, S) knows SKey(B) generates nb
SERVER(S) knows SKey generates kab
```

Values for na, nb, kab will be freshly supplied by the manager processes.

```
#Protocol description

0.    -> A : B
1.  A -> B : na
2.  B -> S : nb, {A, na}{SKey(B)}
3.  S -> A : nb, {B, kab, na}{SKey(A)}
4.  S -> B : nb, {A, kab, nb}{SKey(B)}
5.  A -> B : {nb}{kab}

# Specification

Agreement(B, A, [])

#Actual Variables

Alice, Mallory : Agent
Sam : Server
N1, N2, N3 : Nonce (Foreground)
NBp : Nonce (KnownBackground)
NBs : Nonce (UnknownBackground)
K1, K2, K3 : SessionKey (Foreground)
KBp : SessionKey (KnownBackground)
KBs : SessionKey (UnknownBackground)
InverseKeys = (K1, K1), (K2, K2), (K3, K3), (KBp, KBp), (KBs, KBs)
```

Background values used for the recycling mechanism.

Declared foreground values used by the corresponding manager processes to generate the continous source of fresh values.

```
#Functions

symbolic SKey

#System

INITIATOR(Alice)
RESPONDER(Alice, Sam)
```

WithdrawOption = True / False

Optional: User can choose whether agents can withdraw at any point during a run or not.

```
#Intruder Information

Intruder = Mallory
IntruderKnowledge = {Alice, Sam, Mallory, NBp, KBp, SKey(Mallory)}
```

IntruderProcesses = SERVER

Functionality of SERVER process is to be captured within the intruder process.

Fig. 2. Casper script for the Yahalom protocol, using data independence techniques

The user can specify which roles should be modelled internally to the intruder process, using a line such as `IntruderProcesses = SERVER`, within the `#Intruder Information` section.

A further extension concerns the ability for the agent processes to withdraw *during* a session or not, captured by `WithdrawOption = True / False`. For full generality, one should allow agents to withdraw, but this increase the state space, so we make it an option.

5.3 Example

We now illustrate some of the advantages of our techniques by considering the results obtained when they are applied to the running example. The script in Figure 2 treats nonces and session keys as data independent. The script defines a system where a particular agent Alice can act both as initiator and responder, and can perform an unbounded number of sequential runs. The server is defined to be internal within the intruder process. The property we are interested in verifying is represented by the authentication specification `Agreement(B, A, [])` which, in brief, means that if `A` thinks she has successfully completed a run of the protocol with `B`, then `B` has previously been running the protocol, apparently with `A`, and furthermore, there is a one-to-one relationship between the runs of `A` and the runs of `B`. In our example, we test this authentication property between `Alice` (as responder) and herself (as initiator).

It is well known that this particular version of the Yahalom protocol is flawed. The attack we use for the purpose of illustrating our techniques is essentially the same as the well-known attack by Syverson (in [19]), except it is a self-authentication attack, between Alice and herself. We write $Alice_I$ and $Alice_R$ to differentiate Alice in her roles as initiator and responder, respectively.

- Message 1. $Alice_I \rightarrow Intruder_{Alice_R} : N_1$.
- The intruder performs the functionality of the server using message 1 from the current run, and an old message 2, namely $\{Alice, NBp\}_{SKey(Alice)}$, where NBp is the unknown background nonce, representing some old nonce used in a previous run. This allows the intruder to generate the two corresponding messages $\{Alice, K_1, N_1\}_{SKey(Alice)}$ and $\{Alice, K_1, NBp\}_{SKey(Alice)}$. If the server were implemented as a separate process, then this would be reflected in the following sequence of steps:

$$\begin{array}{lll} \text{Message 2.} & Intruder_{Alice_R} \rightarrow Sam & : N_1, \{Alice, NBp\}_{SKey(Alice)} \\ \text{Message 3.} & Sam \rightarrow Intruder_{Alice_I} & : N_1, \{Alice, K_1, NBp\}_{SKey(Alice)} \\ \text{Message 4.} & Sam \rightarrow Intruder_{Alice_R} & : N_1, \{Alice, K_1, N_1\}_{SKey(Alice)}. \end{array}$$

- Message 3. $Intruder_{Sam} \rightarrow Alice_I : \{Alice, K_1, N_1\}_{SKey(Alice)}$.
- Message 5. $Alice_I \rightarrow Intruder_{Alice_R} : \{N_1\}_{K_1}$.

Alice, as initiator, believes she has completed a run of the protocol with herself, as responder, when in actual fact, Alice did not participate in this latest run as responder.

This attack requires Alice to have run a session with herself previously, so in our previous models, without data independence techniques, we would not have caught this particular attack if we had defined a system where each agent could only perform one run each. When using data independent techniques, we do not need to worry about how many runs each agent has to perform in order to be sure we have captured potential attacks, since our techniques are such that the number of runs each instance can perform is unbounded.

Furthermore, this example nicely illustrates how effective the use of stale knowledge is in terms of state space. In the attack above, where we provided the intruder with stale knowledge, the attack was found in 223 states. However, if we remove the stale knowledge from the intruder's initial knowledge and perform the same check again, then we get the equivalent attack, but FDR now requires 258,967 states to find it. In systems where no attack is found, the difference becomes even larger.

6 Conclusion

In this paper, we presented some generalisations and the full automation of the data independence techniques developed in [17]. The first main generalisation involved dropping the assumption presented in Section 3.1, thereby making our techniques applicable to a broader spectrum of protocols. The second concerned the method used for calculating the stale knowledge sets: to ensure that the maximum set is correctly generated taking all sorts of subtleties within the model (for example, algebraic equivalences) into account, we based our new method of calculating them upon the existing computational methods that are used to calculate the intruder's initial knowledge.

The work on the application of data independence techniques presented thus far has successfully drawn us much closer towards complete correctness proofs automatically generated by Casper and FDR. We are now interested in expanding this work to be able to construct proofs for an arbitrary number of agents running the protocol in parallel with each other. We believe that our methods of incorporating honest agents into the intruder will prove useful for this.

Further planned extensions include the automation of the calculation of the number of foreground values required for each data independent type (as described in Section 5.2), a generalisation of the argument in Section 4 so as to do away with the additional assumptions, the incorporation of time stamps into our techniques, and the continued development of optimisation strategies within our models to reduce the state space size.

Data independence applies to a wide range of notations other than CSP, and we imagine that the same sort of ideas discussed here could profitably be used in other protocol model checkers. However, details will inevitably vary from notation to notation and require care. An encouraging development here is the work of Lazić and Nowak ([7]) which shows how Lazić's CSP data independence results can be transferred to a general setting.

References

1. Burrows, M., Abadi, M., Needham, R.: A Logic of Authentication. Proceedings of the Royal Society of London A, Vol. 426 (1989) 233-271
2. Formal Systems (Europe) Ltd: Failures-Divergences Refinement: FDR2 Manual (1997)
3. Hui, M., Lowe, G.: Fault-Preserving Simplifying Transformations for Security Protocols. Submitted for publication (2000)
4. Lazić, R.S.: A semantic study of data-independence with applications to the mechanical verification of concurrent systems. Oxford University D.Phil thesis (1998)
5. Lazić, R.S., Roscoe, A.W.: 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)
6. Lazić, R.S., Roscoe, A.W.: Verifying determinism of data-independent systems with labellings, arrays and constants. Proceedings of INFINITY (1998)
7. Lazić, R.S., Nowak, D.: A Unifying Approach to Data-independence. Proceedings of the 11th International Conference on Concurrency Theory (2000)
8. Lowe, G.: An Attack on the Needham-Schroeder Public-Key Authentication Protocol. Information Processing Letters, Vol. 56 (1995) 131-133
9. Lowe, G.: Breaking and fixing the Needham-Schroeder public-key protocol using FDR. Proceedings of TACAS '97. Springer LNCS 1055 (1996)
10. Lowe, G.: Casper: a compiler for the analysis of security protocols. Proceedings of 1997 IEEE Computer Security Foundations Workshop. IEEE Computer Society Press (1997)
11. Lowe, G.: A hierarchy of authentication specifications. Proceedings of 1997 IEEE Computer Security Foundations Workshop. IEEE Computer Society Press (1997)
12. Lowe, G., Roscoe, A.W.: Using CSP to detect errors in the TMN protocol. IEEE transactions on Software Engineering **23**, 10 (1997) 659-669
13. Marrero, W., Clarke, E., Jha, S.: A Model Checker for Authentication Protocols. Proceedings of the DIMACS Workshop on Design and Formal Verification of Security Protocols (1997)
14. Mitchell, J.C., Mitchell, M., Stern, U.: Automated Analysis of Cryptographic Protocols Using Murϕ. IEEE Symposium on Security and Privacy (1997) 141-151
15. Roscoe, A.W.: The theory and practice of concurrency. Prentice Hall (1998)
16. Roscoe, A.W.: Proving security protocols with model checkers by data independence techniques. Proceedings of the 11th IEEE Computer Security Foundations Workshop (1998)
17. Roscoe, A.W., Broadfoot, P.J.: Proving security protocols with model checkers by data independence techniques. Journal of Computer Security. Special Issue CSFW11 (1999) 147-190
18. Shmatikov, V., Stern, U.: Efficient Finite-State Analysis for Large Security Protocols. Proceedings of the 11th IEEE Computer Security Foundations Workshop (1998)
19. Syverson, P.: A Taxonomy of Replay Attacks. Proceedings of the 7th IEEE Computer Security Foundations Workshop (1994) 131-136