# Using a PVS Embedding of CSP to Verify Authentication Protocols

To be presented at TPHOLs'97, Bell Labs, Murray Hill, New Jersey, USA

Bruno Dutertre[1] and Steve Schneider[2]

[1] Department of Computer Science, Queen Mary and Westfield College,
University of London, London E14NS, UK
[2] Department of Computer Science, Royal Holloway,
University of London, Egham, Surrey TW20 0EX, UK

**Abstract.** This paper presents an application of PVS to the verification of security protocols. The objective is to provide mechanical support for a verification method described in [14]. The PVS formalization consists of a semantic embedding of CSP and of a collection of theorems and proof rules for reasoning about authentication properties. We present an application to the Needham-Schroeder public key protocol.

## 1 Introduction

Authentication protocols are used in insecure networks by principals who want to get assurance about their correspondent's identity. Designing such protocols is notoriously error-prone and attacks can often exploit weaknesses or subtle flaws. Validating authentication protocols requires a rigorous analysis and several formal approaches have been advocated for this purpose [3, 16, 12, 9].

In [14], Schneider presents such a method based on CSP [7]. The approach relies on a general network model which includes legitimate protocol participants, the users, and an intruder, the enemy. Both the users and the enemy are specified as CSP processes and authentication properties are expressed as constraints on the sequences of messages the whole network can produce.

The verification strategy uses rank functions, that is, functions which assign an integer value to messages. A key theorem shows that authentication properties can be verified by finding a rank function which satisfies appropriate conditions. These depend on the nature of the encryption mechanism used, on the definition of the users, and on the property to be verified. An important benefit of the technique is to decompose an authentication property – a global property of a network – into local properties of the protocol participants.

This paper shows how the PVS theorem prover [5] can provide effective mechanical support to the above method. Using a semantics embedding of CSP, we specify the general network model in PVS and derive the important theorems about authentication and rank functions. We then define specialised PVS rewrite rules and proof commands to facilitate the verifications. With these rules and commands, the proofs of authentication are very systematic and require only little manual guidance. This allows the user to concentrate on the most important aspects of the analysis: finding rank functions.

In the remainder of this paper, we give a brief introduction to CSP and an overview of the modelling and analysis approach. We then describe the formalization of the network model and of the verification method in PVS. We give a simple example of application to the Needham Schroeder public key protocol[10]. Finally, we discuss and compare our developments with other mechanisations of CSP and with other verification methods for security protocols.

## 2 Authentication Protocols in CSP

### 2.1 CSP Notation

CSP is an abstract language for describing concurrent systems which interact through message passing [7]. Systems are modelled in terms of the events they can perform, each event corresponding to a potential communication between a system and its environment. CSP is a process algebra: systems are constructed from a set of elementary processes which can be combined using operators such as prefixing, choice, or parallel composition. Different semantic models are available; in this paper only the simplest – the so-called trace semantics – is considered.

We assume that a fixed set $\Sigma$ of all possible events is given. A process is characterised by a set of *traces*, that is, finite sequences of elements of $\Sigma$. Each trace represents a possible sequence of communications one can observe on the process interface. The set of traces of a process $P$ is prefix-closed; if one observes a trace $tr$ then all the prefixes of $tr$ have been seen before.

The particular dialect we use includes four primitive notions. The syntax of process expressions is as follows:

$$P \ ::= \ Stop \mid a \rightarrow P \mid \Box_{i \in I} P_i \mid P_1 \Box P_2 \mid P_1 \,|[A]|\, P_2 \mid P_1 \,|||\, P_2,$$

where $a$ is an element of $\Sigma$, $I$ a non-empty set, and $A$ a subset of $\Sigma$. These expressions have the following informal interpretation.

- $Stop$ is the process which cannot engage in any event (deadlock).
- $a \rightarrow P$ is able initially to perform only the event $a$ after which it behaves as $P$.
- $\Box_{i \in I} P_i$ is the choice among an indexed family of processes $P_i$. The resulting process can behave as any one of the $P_i$. When only two processes are involved, choice is denoted by $P_1 \Box P_2$.
- $P_1 \,|[A]|\, P_2$ is the parallel composition of $P_1$ and $P_2$ with synchronization on events in $A$. If one of the processes is willing to engage in an event of $A$ then it has to wait until the other is ready to perform the same event. On events which do not belong to $A$, $P_1$ and $P_2$ do not synchronise; they can perform any such event independently of each other. $P_1 \,|||\, P_2$ is an abbreviation for $P_1 \,|[\emptyset]|\, P_2$.

### 2.2 A Model for Authentication Protocols

In the analysis of authentication protocols, we consider the general network architecture shown in Fig. 1. The network consists of a set of user processes and of an enemy which has full control over the communication medium. The enemy can block, re-address, duplicate, corrupt, or fake messages but we assume that it cannot decrypt or encrypt messages without the appropriate keys.

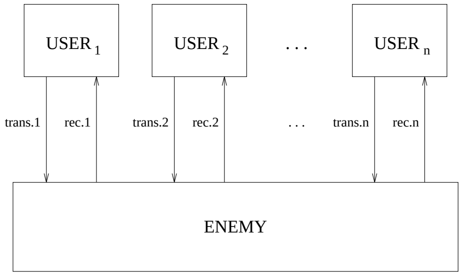

**Fig. 1.** The network

Each user has a unique identity. Its interface with the medium consists of two channels, one for transmission and one for reception. Accordingly, communications are modelled by two types of events. A transmission event is of the form $trans.i.j.m$ and is interpreted as "user $i$ sends a message $m$ destined for user $j$". A reception event is of the form $rec.i.j.m$ and means "$i$ receives a message $m$, apparently from user $j$". The communication channels are private; no user other than $i$ can produce events of the form $trans.i.j.m$ or $rec.i.j.m$.

In order to model the capabilities of the enemy, we use a relation $\vdash$ which specifies when new messages may be generated from existing ones: $S \vdash m$ means that knowledge of all the messages of the set $S$ is enough to produce the message $m$. The relation depends on the particular encryption mechanism used but we can assume that certain natural conditions are satisfied, such as,

$$\forall S, m \;\; m \in S \Rightarrow S \vdash m,$$
$$\forall S, S', m \;\; S \subseteq S' \wedge S \vdash m \Rightarrow S' \vdash m.$$

With the preceding notations, the enemy is specified as follows:

$$\begin{aligned} \mathit{ENEMY}(S) = {} & (\Box_{i,j,m}\ trans.i.j.m \rightarrow \mathit{ENEMY}(S \cup \{m\})) \\ & \Box\ (\Box_{i,j,(m \mid S \vdash m)}\ rec.i.j.m \rightarrow \mathit{ENEMY}(S)). \end{aligned}$$

This describes the behaviour of an enemy which has knowledge of a set of messages $S$. Such a process can either allow a user to transmit a message $m$ after which it behaves as $\mathit{ENEMY}(S \cup \{m\})$ or generate a new message from the set

$S$ and send it to an arbitrary destinator. The enemy's behaviour at the start of a protocol is modelled by the process $\mathit{ENEMY}(\mathit{INIT})$ where $\mathit{INIT}$ represents the information initially available to the enemy.

The user description depends entirely on the protocol being modelled and consists of a family of processes $\mathit{USER}(i)$. The whole network is the following composition of users and enemy:

$$\mathit{NET} = (\ |||_i\ \mathit{USER}(i)\ )\ |[\mathit{trans}, \mathit{rec}]|\ \mathit{ENEMY}(\mathit{INIT}).$$

The users do not communicate directly with each other but the enemy and the composition of users synchronize on all transmission and reception events.

### 2.3 Checking Authentication Properties

The specification of various security properties is discussed in [13]. Authentication involves two disjoint sets of events $T$ and $R$; a process $P$ satisfies the property $T$ `authenticates` $R$ if occurrence of any event of $T$ in a trace of $P$ is preceded by occurrence of some element of $R$. This is denoted by

$$P\ \mathbf{sat}\ T\ \texttt{authenticates}\ R. \tag{1}$$

Examples in [14] illustrate how this relates concretely to authentication. Formally, $T$ `authenticates` $R$ is an abbreviation for the trace predicate

$$tr \restriction R = \langle\rangle \Rightarrow tr \restriction T = \langle\rangle, \tag{2}$$

where $\restriction$ denotes projection[3] and $\langle\rangle$ is the empty trace. The statement (1) is interpreted as "all the traces $tr$ of $P$ satisfy predicate (2)", that is, any trace of $P$ which does not contain events of $R$ does not contain events of $T$ either.

In order to verify authentication properties of a protocol, we have to prove statements of the form $\mathit{NET}$ **sat** $T$ `authenticates` $R$. It can be seen that this condition is equivalent to

$$\mathit{NET}\ |[R]|\ \mathit{Stop}\ \mathbf{sat}\ tr \restriction T = \langle\rangle. \tag{3}$$

This equivalence is the basis of the proof strategy described in [14]. The idea is to assign to every message $m$ an integer value $\rho(m)$ called its rank in such a way that messages occurring in events of $T$ have non-positive rank while only messages of positive rank can be produced by $\mathit{NET}\ |[R]|\ \mathit{Stop}$.

Let $\mathcal{M}$ be the message space for a given protocol. A rank function is a function $\rho$ from $\mathcal{M}$ to the integers. Given such a function, we denote by $\rho^+$ the set of messages of positive rank and by $M(tr)$ the set of messages which occur in a trace $tr$. From the definition of $\mathit{NET}$ and $\mathit{ENEMY}$, one can derive the following key theorem [14].

[3] $tr \restriction R$ is the maximal subsequence of $tr$ all of whose elements belong to $R$.

**Theorem 1.** *If the four conditions below are satisfied,*

$$\begin{aligned}
&INIT \subseteq \rho^+,\\
&\forall S, m.\ \ S \subseteq \rho^+ \wedge S \vdash m \Rightarrow \rho(m) > 0,\\
&T \cap \rho^+ = \emptyset,\\
&\forall i.\ \ USER(i) \,|[R]|\, Stop \ \ \mathbf{sat} \ \ M(tr{\restriction}rec) \subseteq \rho^+ \Rightarrow M(tr{\restriction}trans) \subseteq \rho^+,
\end{aligned}$$

*then*

$$NET \ \mathbf{sat}\ T \ \texttt{authenticates}\ R.$$

With this result, one can verify authentication properties by finding an appropriate rank function. Showing that the four conditions are satisfied is simpler than a direct approach because user processes can be considered individually.

## 3 Embedding CSP in PVS

Our mechanization is based on a semantic embedding of CSP: Traces are represented by lists of events, processes are prefix-closed sets of traces, and the CSP operators are functions on processes which preserve the closure condition. Such a formalization is classic and similar to Camilleri's HOL embedding of CSP [4]. The main differences are the representation of events and processes, and the variant of CSP considered. In [4], events are considered as atomic symbols and are represented by strings. Our formalisation is more general and uses parametric types. Given any type `T`, `trace[T]` and `process[T]` represent traces and processes with events of type `T`. The CSP dialect considered by Camilleri is Hoare's original definition of deterministic processes [7]. In this model, a process has two components, a set of traces and an alphabet of events representing the interface. Due to this interface, there are restrictions on certain CSP operators. For our purpose, it is better to follow [14] and use the CSP variant presented previously.

Our definition of processes relies on PVS subtyping; `process[T]` is a subtype of `set[trace[T]]` defined as follows:

```
S: VAR set[trace[T]]
process: TYPE = { S | S(null) AND prefix_closed(S) }.
```

A process is any set of traces which contains the empty trace `null` and which is prefix-closed. All general results about sets or sets of traces apply then immediately to processes.

The CSP primitives are easily defined[6]. It is also convenient to generalise the two parallel composition operators to arbitrary (non-empty) families of processes. This non-standard extension of CSP does not pose any theoretical problem in the trace model and generalises the results presented in Sec. 2. We can consider networks with infinitely many users and all the theorems still hold. Moreover, the PVS statement and proof of these theorems are much simpler if infinite parallel composition is allowed.

Since PVS has a fixed syntax, we cannot use the standard CSP notations. Instead we use existing PVS symbols as indicated in Tab. 1. The operators `Choice` and `Interleave` are polymorphic functions which apply to indexed families of processes. For example, `Choice` is of parametric type `[[U -> process[T]] -> process[T]]`. PVS has a special syntax for denoting the applications of such functions to lambda terms; `Choice(lambda i: P(i))` can be written `Choice! i: P(i)`. More complex expressions are also valid:

```
Choice! i, j: Q(i) // P(i, j)
Choice! i, (j | i < j): i >> (j >> Stop[nat]).
```

In the two expressions above, PVS infers the correct parameter instantiation from the types of the variables and processes. In the first case, `U` is instantiated with a tuple type and in the second case, `U` is instantiated with the dependent type `[i:nat, {j:nat| i < j}]`.

**Table 1.** Syntax of process expressions.

| Operation | CSP | PVS |
|---|---|---|
| Stop | $Stop$ | `Stop` |
| Prefix | $a \rightarrow P$ | `a >> P` |
| Choice | $P_1 \Box P_2$ | `P1 \/ P2` |
| | $\Box_{i \in I} P_i$ | `Choice! i : P(i)` |
| Parallel Composition | $P_1 \vert[A]\vert P_2$ | `Par(A)(P1, P2)` |
| | $P_1 \vert\vert\vert P_2$ | `P1 // P2` |
| | $\vert\vert\vert_{i \in I} P_i$ | `Interleave! i : P(i)` |

Recursive processes are defined as least fixed points of monotonic functions. Given such a function `F` of type `[[U -> process[T]] -> [U-> process[T]]]`, `mu(F)` denotes the least fixed point of `F`. This form of the `mu` operator is necessary for defining recursive processes with parameters. A simpler form is available for the non-parametric case.

The preceding elements allow us to define CSP processes in PVS. In order to reason about such processes, we provide various lemmas such as the associativity of choice and parallel composition [6]. For specifying properties of processes we imitiate the **sat** operator. Properties are predicates on traces, that is, sets of traces, and the satisfaction relation is

```
|>(P, E): bool = subset?(P, E).
```

For example, we can translate the statement $P$ **sat** $tr \restriction D = \langle \rangle$ to

```
P |> { tr | proj(tr, D) = null }.
```

Various rules about satisfaction and induction theorems for reasoning about fixed points are provided [6].

# 4 The Authentication Model in PVS

## 4.1 Network

It is routine to specify the network. Events are represented by an abstract data type parameterized by the types of user identities and messages.

```
event[I, M: TYPE]: DATATYPE
  BEGIN
    trans(t_snd, t_rcv: I, t_msg: M): trans?
    rec(r_rcv, r_snd: I, r_msg: M): rec?
  END event
```

From this specification, PVS generates an axiomatic definition of the data type. The functions `trans` and `rec` are constructors; events are either of the form `trans(i, j, m)` or `rec(i, j, m)`. Functions such as `t_snd` give access to the components of events. The two functions `trans?` and `rec?` are recognisers of type `[event -> bool]` and characterize transmission and reception events, respectively.

The enemy is defined using the least fixed point operator. The process depends on two type parameters as above and on a message generation relation.

```
enemy[Identity, Message: TYPE,
      |- : [set[Message], Message -> bool]]: THEORY
  BEGIN

  ...

  F(X)(S): process[event] =
        (Choice! i, j, m: trans(i, j, m) >> X(add(m, S)))
     \/ (Choice! i, j, (m | S |- m): rec(i, j, m) >> X(S))

  enemy: [set[Message] -> process[event]] = mu(F)

  END enemy.
```

For such a definition to be sound, PVS requires us to show that `F` is monotonic by generating a proof obligation (TCC) [11].

Users can be arbitrary processes provided they satisfy the interface constraints. The type `user_process` below captures this restriction:

```
LocalEvents(i): set[event] =
  {e | EXISTS m, j: e = trans(i,j,m) OR e = rec(i,j,m)}

user_process: TYPE =
  [i: Identity -> {P | subset?(sigma(P), LocalEvents(i))}].
```

The function `sigma` gives the set of events `P` can generate. Any `user` of the above type is a function of domain `Identity` and range `process[T]` such that the set of events generated by `user(i)` is included in `LocalEvents(i)`. For defining networks, we use the function

```
network(baddy, P): process[T] = Par(fullset)(baddy, Interleave(P)),
```

where `baddy` is any process and `P` is of type `[Identity -> process[T]]`. The constant `fullset` is the set of all transmission and reception events.

### 4.2 Key Theorems

The main results of Sec. 2 are proved in a theory **network** which has the same parameters as **enemy** and makes the following assumption:

```
montonic_gen: ASSUMPTION
  FORALL A, B, m: subset?(A, B) AND (A |- m) IMPLIES (B |- m).
```

Within the theory, **monotonic_gen** can be used like an axiom, but PVS generates a TCC to check that the assumption holds when one imports a particular instance of **network**.

A first lemma follows from the previous assumption and is proved using the induction rule for least fixed points:

```
Gen(S): set[Message] = { m | S |- m }

Prop(S): set[trace[event]] =
  { tr | subset?(rec_msg(tr), Gen(union(S, trans_msg(tr)))) }

enemy_prop: THEOREM enemy(S) |> Prop(S).
```

Informally, this means that any message **enemy(S)** can produce is generated from the set **S** and the messages the enemy has intercepted from users.

Now, given a function **rho** of type **[Message->int]**, we define two trace predicates:

```
RankUser(rho): set[trace[event]] =
  {tr | pos_trans(rho, tr) IMPLIES pos_rec(rho, tr) }

RankEnemy(rho): set[trace[event]] =
  {tr | pos_rec(rho, tr) IMPLIES pos_trans(rho, tr) },
```

where **pos_rec(rho, tr)** and **pos_trans(rho, tr)** are true if all the reception or transmission events of **tr**, respectively, have positive rank by **rho**. The first half of the main theorem is a corollary of **enemy_prop**:

```
rank_property: COROLLARY  positive(rho, INIT)
    AND (FORALL S: positive(rho, S) implies positive(rho, Gen(S))
  IMPLIES enemy(INIT) |> RankEnemy(rho).
```

If the two premisses are satisfied then the enemy cannot generate messages of non-positive ranks unless it receives such messages from the users.

For convenience, we use a specific restriction operator; the process $P|[R]|Stop$ is written **P # R** in PVS. We then get the following essential property:

```
main_result: LEMMA  baddy |> RankEnemy(rho)
    AND (FORALL i: user(i) # R |> RankUser(rho))
  IMPLIES network(baddy, user) # R |> {tr | positive(rho, tr)}.
```

In this lemma, the two premisses are symmetric: **baddy** does not generate messages of negative rank if the users send messages of positive ranks and **user(i) # R** does not send messages of negative ranks if it only receives messages of positive ranks. By induction, the two conditions imply that no message of positive rank can ever appear in a trace of **network(baddy,user) # R**. From this and lemma **rank_property** we obtain the main theorem:

```
authentication_by_rank: THEOREM
      positive(rho, INIT)
  AND (FORALL S, m:
        positive(rho, S) AND (S |- m) IMPLIES rho(m) > 0)
  AND (FORALL i: user(i) # R |> RankUser(rho))
  AND non_positive(rho, T)
IMPLIES network(enemy(INIT), user) |> auth(T, R).
```

### 4.3 Automating the Verifications

The previous theorem is the main tool for verifying authentication properties. When using it, most of the effort concentrates on properties of the form `user(i) # R |> RankUser(rho)`. There are also hidden conditions which arise from the type of `user`: we have to prove that `user(i)` can only generate events which belong to `LocalEvents(i)`. All these proofs can be partially automated by using the PVS rewriting facilities and by defining specific proof strategies.

The interface constraints are of the form `subset?(sigma(P), E)` where `P` is a CSP expression and `E` is a set of events. We can systematically develop rules which rewrite the above inclusion in a simpler form according to the top-level operator of `P`. There is such a rule for every CSP primitive; a few examples are given below:

```
interface_pref: LEMMA subset?(sigma(a >> P), E) IFF
    E(a) AND subset?(sigma(P), E)

interface_choice3: LEMMA subset?(sigma(Choice(P)), E) IFF
    FORALL i: subset?(sigma(P(i)), E)

interface_stop: LEMMA subset?(sigma(Stop), E)

interface_par: LEMMA subset?(sigma(P1), E) AND subset?(sigma(P2), E)
    IMPLIES subset?(sigma(Par(A)(P1, P2)), E).
```

All such lemmas can be installed as automatic rewrite rules which PVS applies in conjunction with built-in simplification and decision procedures. The first two rules are inconditional rewritings which applies in the left to right direction; `interface_stop` is also inconditional but matching terms are rewritten to `true`. The last rule is conditional: terms matching the right hand side of the implication reduce to `true` provided the premisses are also reduced to `true` by the decision procedures or by further rewriting.

Lemma `interface_pref` introduces expressions of the form `E(a)`. For user processes, `E` is `LocalEvents(i)` for some fixed `i` and two more rewrite rules are necessary:

```
local_transmission: LEMMA LocalEvents(i)(trans(i, j, m))

local_reception: LEMMA LocalEvents(i)(rec(i, j, m)).
```

In practice, automatic rewriting with these two rules and the preceding lemmas prove almost all interface constraints. The few exceptions are usually due to fixed points. The presence of quantifiers in the rules for fixed points interrupts the chain of rewrites and little manual intervention is required before rewriting can proceed.

For properties of the form `P # R |> RankUser(rho)`, rewrite rules can also largely reduce the proof effort. However, rewriting is not sufficient and the rules must be supplemented with specific proof strategies.

In a similar way as above, two sets of rewrite rules are constructed which apply to expressions of the form `P # R` or `P |> RankUser(rho)`. For example, the rules for prefix and `Stop` are:

```
restriction_pref: LEMMA
    (a >> P) # B = IF B(a) THEN Stop ELSE a >> (P # B) ENDIF

rank_user_output: LEMMA (trans(i,j,m) >> P) |> RankUser(rho)
    IFF rho(m)>0 AND P |> RankUser(rho)

rank_user_input: LEMMA (rec(i,j,m) >> P) |> RankUser(rho)
    IFF (rho(m)>0 IMPLIES P |> RankUser(rho))

restriction_stop: LEMMA Stop # B = Stop

rank_user_stop: LEMMA Stop |> RankUser[rho].
```

Such lemmas are more complex than the interface rules and automatic rewriting does not work as well. The first three examples illustrate some of the difficulties. PVS considers `restriction_pref` as a conditional rule which applies only if `B(a)` reduces to `true` or `false`. This cannot be expected in general since `B` depends on the authentication property being checked. The other two lemmas introduce terms involving rank functions which cannot systematically reduce to `true`. As a result, the chains or reductions required to apply conditional rules often fail.

Despite these limitations, some form of automation is still possible. The proofs of `P # R |> RankUser(rho)` have a regular pattern and the same sequences of proof commands are applied repeatedly. For example, if `P` is of the form `rec(i,j,m) >> Q` then the corresponding PVS proof starts by the following sequent:

```
  |----
{1} (rec(i,j,m) >> Q) # R |> RankUser(rho).
```

The natural first step is to apply `(rewrite "restriction_pref")`. This yields

```
  |----
{1} IF R(rec(i,j,m)) THEN Stop
     ELSE rec(i,j,m) >> (Q # R) ENDIF |> RankUser(rho).
```

The obvious case split generates two subgoals. The first one can be solved immediately using the rewrite rules for `Stop` and the second is:

```
  |----
{1} rec(i,j,m)>> (Q # R) |> RankUser(rho)
{2} R(rec(i,j,m)).
```

Lemma `rank_user_input` can be applied and further propositional simplification yields:

```
{-1} rho(m)>0
   |----
 {1} Q # R |> RankUser(rho)
 {2} R(rec(i,j,m)).
```

At this point, the same sequence of proof commands applies if `Q` is a prefix expression which starts by a reception event. More generally, successive goals in such proofs are of the form

```
    ...
   |----
 {1} P # R |> RankUser(rho)
    ...
```

and the syntactic form of `P` determines a sequence of commands which can be used systematically.

We exploit this regularity by defining specific proof strategies. An initialisation strategy installs automatic rules for `Stop` together with `rank_user_input` and `rank_user_output`. The other strategies correspond to particular CSP operators. For example, the strategy `(prefix)` applies the following command:

```
(try (rewrite "restriction_pref")
     (then* (lift-if) (assert) (prop))
     (skip)).
```

This performs the four steps of the proof sketched previously. The strategy attempts to rewrite the current goal with `restriction_pref`. If this fails, `(skip)` leaves the goal unchanged, otherwise, an expression of the form `IF R(a) THEN Stop ELSE a >> (P # R) ENDIF` is introduced. Three commands are then applied successively. In the second step, `(assert)` activates automatic rewriting. As a result, the first branch of the conditional is reduced to `true` and the second branch is rewritten by one of `rank_user_input` or `rank_user_output`. The effect of `(prefix)` depends on whether `a` is a transmission or a reception event and is described in Fig. 2.

Reception Events: $a = rec.i.j.m$

$$\frac{\Gamma, \rho(m) > 0 \ \vdash\ P \# R \ \mathbf{sat}\ \mathit{RankUser}(\rho), R(a), \Delta}{\Gamma \ \vdash\ (a \rightarrow P) \# R \ \mathbf{sat}\ \mathit{RankUser}(\rho), \Delta}$$

Transmission Events: $a = trans.i.j.m$

$$\frac{\Gamma \ \vdash\ \rho(m) > 0, R(a), \Delta \qquad \Gamma \ \vdash\ P \# R \ \mathbf{sat}\ \mathit{RankUser}(\rho), R(a), \Delta}{\Gamma \ \vdash\ (a \rightarrow P) \# R \ \mathbf{sat}\ \mathit{RankUser}(\rho), \Delta}$$

**Fig. 2.** Effect of the `prefix` strategy.

With similar strategies for the other CSP primitives, the proofs can be conducted at a fairly abstract level. The details of the PVS mechanics are hidden from the user who simply selects the right strategy. When none applies, the remaining sequents do not contain any process expression and correspond to simple properties of the rank function which have to be proved by other means. The structure of the proofs is for a large part independent of the rank function under investigation. It is easy to experiment with various rank functions and most of the proofs remain unchanged.

# 5 Applications

We have experimented the PVS mechanization on two versions of the Needham Schroeder public key protocol [10]. All the properties examined in [14] have been mechanically verified [6]. In the sequel, we consider the following variant proposed by Lowe[9][4]:

$$\begin{array}{l} A \rightarrow B : \{N_a, A\}_{K_b} \\ B \rightarrow A : \{N_a, N_b, B\}_{K_a} \\ A \rightarrow B : \{N_b\}_{K_b}. \end{array}$$

## 5.1 Encryption

The first step in the analysis is to model public key encryption. We follow [14] and represent messages by an abstract data type:

```
message: DATATYPE WITH SUBTYPES key, nonkey
  BEGIN
  text  (x_text: Text)                  : text?   : nonkey
  nonce (x_nonce: Nonce)                : nonce?  : nonkey
  user  (x_user: Identity)              : user?   : nonkey
  public(x_public: Identity)            : public? : key
  secret(x_secret: Identity)            : secret? : key
  conc  (x_conc, y_conc: message)       : conc?   : nonkey
  code  (x_code: key, y_code: message) : code?   : nonkey
  END message.
```

In this data type, the subtypes `key` and `nonkey` are similar to extra recognisers. The first five constructors define elementary messages of different natures and the last two correspond to concatenation and encryption.

Asymmetric cryptosystems satisfy the identity $\{\{m\}_{K_a}\}_{K_a^{-1}} = m$ but the equivalent PVS assumption

```
code(secret(i), code(public(i), m)) = m
```

is not sound; it contradicts the data type axioms. Instead we use a function `crypto` which performs message normalisation. We can then define the message generation relation `|-` as follows:

[4] $K_x$ and $K_x^{-1}$ are $x$'s public and private key; $\{m\}_K$ is $m$ encrypted using $K$.

```
Gen(S)(m): INDUCTIVE bool =
      S(m)
  OR (EXISTS m1, m2: Gen(S)(m1) AND Gen(S)(m2) AND m=conc(m1, m2))
  OR (EXISTS m1: Gen(S)(conc(m1, m)) OR Gen(S)(conc(m, m1)))
  OR (EXISTS m1, k: Gen(S)(m1) AND Gen(S)(k) AND m=crypto(k, m1));

|-(S, m): bool = Gen(S)(m).
```

Gen(S) is the set of messages which can be generated from S and is defined inductively. PVS automatically generates two induction axioms for Gen from which we can show the required monotonicity assumption:

```
gen_monotonic2: COROLLARY
  subset?(S1, S2) AND (S1 |- m) IMPLIES (S2 |- m).
```

### 5.2 Users and Verifications

The modelling allows several variants of the protocol to be analysed, from the simple case of two participants executing a single run to multiple runs executed concurrently. In a simple example, the initiator is described by the process below

```
userA: process[event] =
  Choice! i, x:
      ( trans(a, i, pub(i, conc(Na, user(a)))) >>
        ( rec(a, i, pub(a, conc3(Na, x, user(i)))) >>
          ( trans(a, i, pub(i, x)) >> Stop[event]))).
```

User $A$ non-deterministically initiates a single run with some user $i$ by sending the message $\{N_a, a\}$ encrypted with $K_i$. Then $A$ is ready to receive a message of the form $\{N_a, x, i\}_{K_a}$ coming from $i$ where $x$ is any nonce. $A$ responds to this message by sending back $\{x\}_{K_i}$.

We assume that the responder $B$ behaves as if participating in a run with $A$:

```
userB: process[event] =
  Choice! y:
      ( rec(b, a, pub(b, conc(y, user(a)))) >>
        ( trans(b, a, pub(a, conc(y, Nb, user(b)))) >>
          ( rec(b, a, pub(b, Nb)) >> Stop[event]))),
```

and we want to prove that reception of $\{N_b\}_{K_b}$ by $B$ ensures that $B$ is effectively communicating with $A$. More precisely, we want to prove that the network satisfies $T$ **authenticates** $R$ where $T$ and $R$ are as follows:

```
T: set[event] = { e | e = rec(b, a, pub(b, Nb)) }
R: set[event] = { e | e = trans(a, b, pub(b, Nb)) }.
```

As shown in [9], this property does not hold for the original Needham-Schroeder protocol; reception of $\{N_b\}_{K_b}$ still ensures that $A$ sent the message but to a user which may be different from $B$. The property holds for Lowe's variant, provided the *ENEMY* does not know $N_b$ or the secret keys of $A$ and $B$.

The rank function we use for the property above is defined in [14]. It satisfies the essential property below:

```
rho(pub(i, m)) =
  IF i=a AND (EXISTS x: m=conc3(x, Nb, user(b))) THEN 1
  ELSE rho(m) ENDIF.
```

In order to apply the main theorem, the most important part of the verification is to show the four following lemmas:

```
interface_userA: LEMMA subset?(sigma(userA), LocalEvents(a))

interface_userB: LEMMA subset?(sigma(userB), LocalEvents(b))

rank_user_a: LEMMA userA # R |> RankUser(rho)

rank_user_b: LEMMA userB # R |> RankUser(rho).
```

The interface constraints are easily checked. The proof script for interface_userA is given below:

```
(AUTO-REWRITE "local_transmission" "local_reception"
    "interface_pref[event]" "interface_stop[event]")
(EXPAND "userA")
(REWRITE "interface_choice3").
```

The first command installs the automatic rewrite rules presented in Sec. 4.3 and the second expands the definition of userA. The third step is a manual application of the interface rule for unbounded choice. This triggers automatic rewriting and the goal is solved at once.

The proofs of the rank preservation properties apply the strategies for choice and prefix. On the remaining goals, we use the predefined strategy (grind) which expands the definitions of rho and R and applies the decision procedures. This is often enough to solve the goals but sometimes the extensionality axioms for the message data type are needed (see [6] for details). For example, the proof script of rank_user_a is

```
(INIT-CSP "Identity" "message")
(EXPAND "userA")
(CHOICE3)
(PREFIX)
(("1" (DELETE 2) (GRIND))
 ("2"
  (PREFIX)
  (PREFIX)
  (DELETE 3 4)
  (GRIND)
  (CASE "x_user(y_conc(conc(conc(....)))) = b")
  (("1" (APPLY (REPEAT (APPLY-EXTENSIONALITY))))
   ("2" (REPLACE -2) (ASSERT)))))).
```

The proofs of authentication properties we have performed with PVS are all similar to the previous example. The most general situations, where concurrent protocol runs are considered, require much more complex rank functions. But, even in such cases, the PVS proofs are not substantially harder. The same proof strategies are used together with (grind) and extensionality rules.

## 6 Discussion and Related Work

This paper shows that PVS can provide efficient support for a non-trivial application of CSP. The usefulness of the mechanisation is clear. In our experience, manual verification of the constraints on rank functions simply does not work. PVS has found many errors in our own manual proofs of authentication properties. The proofs we performed generalise the results presented in [9] to more complex variants of the Needham Schroeder protocol. In recent works, larger protocols have been verified [2].

The basis of our mechanisation is a semantics embedding of CSP and PVS is adequate for this purpose. Camilleri [4] and Thayer [17] present similar embeddings in HOL and IMPS. Our main contribution compared with these works is the application to the specific problem of authentication and the development of PVS theories for this purpose.

In the analysis of security protocols, tools exist which support various belief logics [3, 1]. Closer to our approach are methods based on modelling protocols as collections of rules for transforming and reducing messages. Tool support in this area [9, 8] is mostly based on analysis for reachability of an insecure state which corresponds to the existence of an attack. This usually requires finitary models and the inability to find an attack does not in itself guarantee correctness of the full-scale protocol. Our verification approach is then a useful complement to these state exploration techniques.

Paulson [12] investigates the application of Isabelle/HOL to proving security properties of protocols by induction. He specifies protocols in terms of traces and rules about how traces can be augmented. This is clearly very close to the CSP trace model but gives no control over when rules may apply. In contrast, the CSP approach maintains the order of protocol steps and the order in which proof rules are applied. Paulson has some useful results about proof reuse while at present we know little about the potential reusability of rank functions.

In future developments we envisage to increase the level of proof automation. Currently, we provide specialised proof commands but the user still has to manually select the right rule. This could be automatized; the rule to apply can be determined from the top-level operator in a CSP process. Another extension would be to allow a more flexible modelling of the space of messages. The use of a PVS data type implies that messages constructed in different ways are different. Algebraic properties (such as the associativity of concatenation or the fact that public-key and private-key encryption are inverse of each other) cannot be introduced as equations. The approach of [14] remains valid even in the presence of more complex equational properties. This can be implemented with PVS but requires extra developments, such as a quotient construction for data types.

Another important avenue to explore will be the extent to which construction of rank functions can be assisted by the attempt to provide a PVS proof. Particular constraints on the rank function arise when instantiating the CSP rules and could be generated by a "blank" run of the PVS proof: the rank constraints appear as unresolved leaves in the proof-tree. Collecting the information may then help us to identify a suitable rank function.

In conclusion, we have presented a viable mechanical support in PVS for the verification of security protocols with respect to authentication properties. There is still much to be done to support verification in the presence of algebraic properties of cryptographic mechanisms, to improve automation, and to gain experience by investigating further protocols.

## Acknowledgements

This work was partially funded by DRA/Malvern.

## References

1. S. H. Brackin. Deciding Cryptographic Protocol Adequacy with HOL: The Implementation. In *TPHOLs'96*. Springer-Verlag, LNCS 1125, 1996.
2. J. Bryans and S. Schneider. Mechanical Verification of the full Needham-Schroeder public key protocol. Technical report, Royal Holloway, University of London, 1997.
3. M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. Technical Report 39, Digital Equipment Corporation, System Research Center, 1989.
4. A. J. Camilleri. Mechanizing CSP trace theory in higher order logic. *IEEE Transactions on Software Engineering*, 16(9):993–1004, 1990.
5. J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. In *Workshop on Industrial-Strength Formal Specification Techniques*, 1995.
6. B. Dutertre and S. Schneider. Embedding CSP in PVS. An Application to Authentication Protocols. Technical report, Royal Holloway, CSD-TR-97-12.
7. C. A .R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.
8. R. Kemmerer, C. Meadows, and Millen J. Three systems for cryptographic analysis. *Journal of Cryptology*, 7(2), 1994.
9. G. Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. In *Proc. of TACAS'96*. Springer-Verlag, LNCS 1055, 1996.
10. R. M. Needham and M. D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Comm. of the ACM*, 21(12):993–999, 1978.
11. S. Owre, N. Shankar, and J. M. Rushby. *The PVS Specification Language*. Computer Science Lab., SRI International, 1993.
12. L. Paulson. Proving Properties of Security Protocols by Induction. Technical Report TR409, Computer Laboratory, University of Cambridge, 1996.
13. S. Schneider. Security Properties and CSP. In *IEEE Symposium on Security and Privacy*, 1996.
14. S. Schneider. Using CSP for protocol analysis: the Needham-Schroeder Public Key Protocol. Technical Report CSD-TR-96-14, Royal Holloway, 1996.
15. J. U. Skakkebæk and N. Shankar. Towards a duration calculus proof assistant in PVS. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*. Springer-Verlag, LNCS 863, September 1994.
16. P. Syverson and P. van Oorschot. On Unifying Some Cryptographic Protocol Logics. In *Proc. of the 1994 IEEE Symp. on Research in Security and Privacy*, 1994.
17. F. J. Thayer. An approach to process algebra using IMPS. Technical Report MP-94B193, The MITRE Corporation, 1994.

This article was processed using the LaTeX macro package with LLNCS style