Embedding CSP in PVS. An Application to Authentication Protocols

Bruno Dutertre[†] Steve Schneider[‡]

Technical Report 736

May 7, 1997

Department of Computer Science
Queen Mary and Westfield College, University of London
Mile End Road, London E1 4NS, UK

† Dept. of Computer Science, Queen Mary and Westfield College, e-mail: bruno@dcs.qmw.ac.uk
‡ Dept. of Computer Science, Royal Holloway, Egham, Surrey TW20 0EX, e-mail: steve@dcs.rhbnc.ac.uk

Abstract

In [28], Schneider applies CSP to the modelling and analysis of authentication protocols and develops a general proof strategy for verifying authentication properties. This paper shows how the PVS theorem prover can provide effective mechanical support to the approach.

Contents

1 Introduction

When exchanging information over insecure networks, users often require some means of verifying each other's identity. Authentication protocols are intended to provide this service. Roughly speaking, authentication means "making sure your correspondent is really who or what he pretends he is". However, authentication comes in a number of flavours [11] and protocols may differ in the kind of authentication properties they were designed for and the kinds they actually provide. In order to study such properties, various formal approaches have been advocated [5, 31, 23, 17].

Schneider [28] presents a method for the analysis and verification of authentication protocols based on CSP [13]. The objective is to make precise the exact authentication properties offered by particular protocols and to make explicit the assumptions necessary for these properties to be achieved. The approach is based on a general 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 they can produce. Deciding whether a protocol satisfies a particular authentication property amounts to deciding whether a CSP process is unable to produce certain sequences of events.

The overall structure of the network is independent of the protocol under consideration: the users communicate with each other through a medium entirely controlled by the enemy. The latter can freely intercept or destroy the messages circulating in the network and is able to modify messages and to generate new ones. The only restriction on the enemy's capabilities are assumptions related to the particular encryption mechanism used. These specify what kind of information the enemy can deduce from the messages exchanged between users and determine what messages it can generate.

The authentication properties considered in [28] are expressed in terms of messages circulating in the network: a message m_a authenticates another message m_b if the occurrence of m_a guarantees the previous occurrence of m_b. If m_a is a message received by a and m_b is a corresponding messages that can only be generated by b then a has evidence that she is communicating with b. This approach generalises to sets of messages authenticating other sets of messages: when one of the authenticating messages occurs then one of the authenticated messages must previously have occurred.

Schneider [28] shows how properties of the previous form can be verified in practice. The general strategy is to specify a function which assigns to every message an integer value called its rank. In order to show that T authenticates R, one has to assign a non-positive rank to every element of T and show that when messages of R are prevented, only messages of positive rank can circulate in the network. A key theorem reduces the latter property to local constraints on the protocol participants together with simple conditions on the rank function. An important benefit of the approach is then to decompose the proof of an authentication property – a global property of the network – into the verification of simpler, local properties of the users and rank function.

By using the decomposition theorem, most of the verification effort concentrates on finding an appropriate rank function. This may often be a non-trivial exercise; the rank function has to satisfy many constraints which depend on the particular authentication property to check, on the definition of users, and on the assumptions made about the encryption scheme. In practice, subtle mistakes or omissions can easily occur and checking all the conditions requires considerable attention to the details. A tool allowing one to experiment with rank functions and to mechanically check that rank conditions are satisfied can be a considerable help. This paper shows how the PVS theorem prover [7, 21] can provide the necessary tool support. We present a set of PVS theories allowing us to specify and reason about CSP processes and to implement the whole verification approach presented in [28]. The basic developments include the definition of the general network model, the proof of all the key theorems, the implementation of specialized CSP proof rules and the verification of their soundness. These basic theories and rules have allowed us to verify mechanically several authentication properties of the Needham-Schroeder public key protocol [20].

The remainder of this document is organised as follows. Section 2 gives a brief introduction to CSP and presents the general approach to analysing authentication protocols. The section describes the network model and defines the authentication properties considered; it then outlines the general proof strategy based on rank functions and presents the key decomposition theorems.

Section 3 describes our representation of CSP in PVS based on an embedding of CSP's trace semantics. The fundamental notions of traces and processes are given a concrete PVS representation: a trace is a list of events and a process is a non-empty, prefix-closed set of traces. The CSP operators and all other elements of CSP are then defined in a straightforward way. With the basic constituents available, Section 4 develops the general proof strategy for authentication properties. This

involves the formalization and mechanical verification of the main theorems and the definition of dedicated proof rules.

In section 5, we apply the previous tools to the Needham-Schroeder public key protocol. The application illustrates the flexibility of the CSP modelling: we can study situations where a single run, consecutive runs or concurrent runs of the protocol are considered. Each variant can be formally described in CSP (and in PVS) and in each case, several authentication properties can be verified. We examine how PVS proofs of authentication properties can be constructed in practice.

Section 6 discusses the advantages and possible limits of using PVS as a mechanical theorem prover for CSP. The section also gives comparisons with related works in the area of formal analysis of security protocols and with other approaches to mechanising CSP.

2 Authentication Protocols in CSP

2.1 CSP notation

CSP is an abstract language for describing concurrent systems which interact via message passing. 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 described from a set of primitive processes which can be combined using operators such as prefixing, choice, or parallel composition. Different semantic models are available and the most appropriate has to be chosen according to the applications and properties under consideration. In the simplest model, the so-called trace semantics, processes are characterised by the sequences of events they may engage in. This is sufficient for investigating safety properties. Other models can distinguish between processes according to non-deterministic and divergence properties and are useful for studying liveness properties [13].

In this paper, only the trace model of CSP is considered. We assume that a fixed set Σ of all possible events is given and each process is characterised by a set of finite sequences of elements of Σ. Each such sequence is called a *trace* of the process and represent a possible sequence of communications one can observe on the process's interface. The set of traces of a process is prefix-closed: if one observes a trace tr then all the prefixes of tr have been observed before.

The particular dialect we use includes a primitive process, *Stop*, and three primitive operations: prefix, choice, and parallel composition. The syntax 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 Σ, I a non-empty set, and A a subset of Σ.

Stop is the process which cannot engage in any event at all; it is equivalent to deadlock.

The expression $a \rightarrow P$ is a prefix construction; the process $a \rightarrow P$ is able initially to perform only the event a after which it behaves as P.

The process $\Box_{i \in I} P_i$ is the choice among the indexed family of processes P_i ($i \in I$). The choice process can behave as any one of the arguments P_i. In the case where only two processes are involved, choice is denoted by $P_1 \Box P_2$ and behaves either as P_1 or as P_2.

The process $P_1 \,|[A]|\, P_2$ is the parallel composition of P_1 and P_2 with synchronization on events in A. If one of the processes is willing to perform an event of A then it has to wait until the other is ready to perform the same event. On events

that do not belong to A, P_1 and P_2 do not synchronize; they perform any such event independently of each other.

The expression $P_1 \,|||\, P_2$ denotes the parallel composition of two processes which do not synchronize at all; it is equivalent to $P_1 \,|[\emptyset]|\, P_2$.

Processes are defined by means of equations. For example, the equation

$$E \quad = \quad a \rightarrow (b \rightarrow Stop \,\Box\, c \rightarrow Stop)$$

defines a process E which can initially perform the event a followed by either b or c. An equivalent definition (in the trace model) could be

$$E \quad = \quad a \rightarrow b \rightarrow Stop \,\Box\, a \rightarrow c \rightarrow Stop.$$

In general, the definitions can be recursive and can involve parameters. For example the following equations define a family of processes $COUNT(n)$ for $n \in \mathbb{N}$ which are mutually recursive.

$$\begin{aligned} COUNT(0) &= up \rightarrow COUNT(1) \\ COUNT(n+1) &= up \rightarrow COUNT(n+2) \,\Box\, down \rightarrow COUNT(n) \end{aligned}$$

The process $COUNT(0)$ can be interpreted as a counter. It can generate any sequence of events *up* or *down* which does not contain more *down* than *up* events.

The semantic models ensure that all such recursive definitions are well founded. Any system of equations of the above form has at least one solution given by a least fixed point construction [13]. Furthermore, if all the CSP expressions in the right hand side of equations are guarded then the solution is unique.

Events can be atomic like *up* and *down* above or can be of the form

$$c.x \quad \text{or} \quad c.x_1.\ldots.x_n.$$

Such events symbolise the communication of a value x or of a tuple of values $x_1, \ldots, x_n$ on a channel c. For example, a fifo buffer might be modelled by a process $FIFO$ with two communication channels *in* and *out*; reception of a value x would then be represented by the event $in.x$ and emission of a value y by the event $out.y$.

In order to simplify the notations, channel names are used to denote sets of events. For example, the composition of $FIFO$ with a process $PRODUCER$ and a process $CONSUMER$ could be written

$$(CONSUMER \,|||\, PRODUCER) \,|[in, out]|\, FIFO.$$

In this expression, $CONSUMER$ and $PRODUCER$ do not directly interact with each other but their composition synchronise with $FIFO$ on all events of the form $in.x$ or $out.y$.

2.2 A general model for authentication protocols

The approach proposed in [28] for analysing authentication protocols assumes a general network architecture, a variant of the Dolev-Yao model [8]. The network consists of a set of user processes which execute the protocol and an enemy which has full control over the communication medium. The users communicate with each other by sending messages through the insecure medium and the enemy can block, re-address, duplicate, and fake messages.

Each user process has a unique address in the network and its interface with the medium consists of two channels, one for transmission and one for reception as

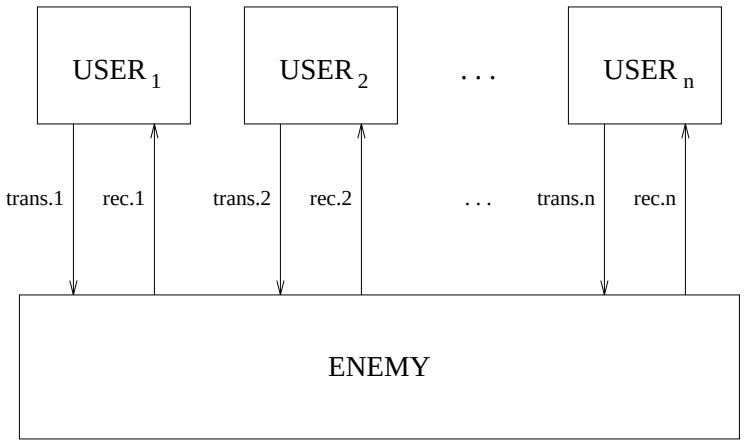

Figure 1: The network

shown in Fig. 1. Accordingly, the communications are modelled by two different types of events. A transmission event is of the form

$$trans.i.j.m$$

where i and j are users' addresses and m is the content of the transmission. Such an event is interpreted as "user i sends a message m destined for user j". Similarly, a reception event is denoted by

$$rec.i.j.m$$

and means "user i receives a message m, apparently from user j".

The communication channels are private: user i can generate events only of the form $trans.i.j.m$ or $rec.i.j.m$. Other users cannot send messages on i's transmission channel or intercept messages on i's reception channel.

In a benign environment, every transmission event $trans.i.j.m$ would be followed by a corresponding reception event $rec.j.i.m$ but our model does not offer any guarantee of this sort. The enemy intercepts every transmission $trans.i.j.m$ and can decide to block it or to deliver the message to a user other than j; it can also read and change the content m and modify the originator field of reception events. Furthermore, the enemy can generate spurious reception events which do not correspond to any transmission.

Authentication protocols use encryption. In order to check authentication properties, we have to assume that the enemy cannot decipher secret messages. The capabilities of the enemy to fake messages will then depend on the encryption mechanisms used and the amount of non-secret information available. We model these capabilities by a relation $\vdash$ which describe when new messages may be generated from existing ones: if S is a set of messages then $S \vdash m$ means that knowledge of all the messages in S is enough to produce m. The relation $\vdash$ depends on the encryption techniques used but we can always assume that certain natural properties are satisfied. For example, in any reasonable model, $\vdash$ satisfies the two following conditions:

$$\begin{aligned} \forall S, m \qquad & m \in S \Rightarrow S \vdash m \\ \forall S, S', m \qquad & S \subseteq S' \wedge S \vdash m \Rightarrow S' \vdash m. \end{aligned}$$

The first one simply means that any known message can be generated, the second that increased knowledge gives an increased possibility of generating messages.

The enemy is described using a parametric process $ENEMY(S)$ where the parameter S is a set of messages. This process models the behaviour of an enemy who has knowledge of all the messages in S. During the execution of a protocol, S represents the information the enemy has gathered: the set contains all the messages the enemy has intercepted from the users and includes a set $INIT$ representing the information available initially.

The process $ENEMY(S)$ can either allow a user to transmit a message or generate a new message from S and send it to an arbitrary destinator. Hence, $ENEMY(S)$ can choose either to engage in an event of the form $trans.i.j.m$ after which it behaves as $ENEMY(S \cup \{m\})$ or in an event $rec.i.j.m$ where $S \vdash m$ after which it behaves as itself:

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

The behaviour of the enemy at the start of a protocol is modelled by a process $ENEMY$ defined by:

$$ENEMY \quad = \quad ENEMY(INIT).$$

The $ENEMY$ is generic and independent of any protocol. The definition is parameterised by the 'generates' relation $\vdash$ and by the initial set $INIT$. Due to the assumptions on $\vdash$ above, the behaviour of a simple message passing device is included among the possible behaviours of the enemy. In other words, $ENEMY$ may decide to transmit messages along without any alteration or interference.

The description of the users depends entirely on the protocol being modelled. Each agent is specified as a process $USER(i)$ where i is the user's identity, that is, its address in the network. In general, authentication protocols involve two main participants which will be described as $USER(a)$ and $USER(b)$ and possibly some trusted third party (an authentication server). We could assume that other users in the network are hostile and collude with the enemy. If i is not one of the main participants, the process $USER(i)$ would then be similar to $ENEMY$:

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

$$USER(i) \quad = \quad Un_i(INIT_i),$$

where $INIT_i$ is the initial information known by $USER(i)$. However, it can be shown that the enemy alone can fake all such users if it already knows $INIT_i$. From a and b's point of view, the composition of hostile users with $ENEMY$ has the same trace behaviour as $ENEMY$ alone provided the sets $INIT_i$ are included in $INIT$. We can then simplify the model by assuming that the $ENEMY$ knows the information $INIT_i$ and that all the users except a, b and the server remain silent. For all such users we set:

$$USER(i) \quad = \quad Stop.$$

The whole network can now be specified as the parallel composition of the users and the enemy:

$$NET \quad = \quad (\,|||_i\, USER(i)\,)\; |[trans, rec]|\; ENEMY.$$

The users do not communicate directly with each other; they are composed with the $|||$ operator. On the other hand, the enemy and the users synchronize on all transmission and reception events.

2.3 Checking authentication properties

Authentication properties of a process P are a special case of safety properties, that is, constraints that all the traces of P must satisfy. Such constraints are written using specific functions on traces and standard mathematical notations. The empty trace is denoted by $\langle\rangle$. If tr is a trace, $M(tr)$ is the set of messages of the trace tr: a message m belongs to $M(tr)$ if some event of the form $trans.i.j.m$ or $rec.i.j.m$ occurs in tr. If D is a set of events, $tr\restriction D$ is the projection of tr on D, that is, the maximal subsequence of tr all of whose events are drawn from D. As usual, channel names can be used to denote sets of events; $tr\restriction trans$ and $tr\restriction rec$ are the subsequences of transmission and reception events of tr, respectively.

These operators allow us to express simple properties of traces such as the absence or presence of certain events. For example, the equality

$$tr\restriction D = \langle\rangle$$

holds if no event of D occurs in tr. Properties of processes are written as statements of the form

$$P \quad \textbf{sat} \quad \phi(tr)$$

where P is a process expression and $\phi(tr)$ a logical formula with free variable tr. Such statements mean that all the traces of P satisfy the property $\phi(tr)$. For example,

$$P \quad \textbf{sat} \quad tr\restriction D = \langle\rangle$$

indicates that P cannot produce events of D.

Authentication involves two disjoint sets of events R and T. A process P satisfies the property T `authenticates` R if occurrence of any event of T in a trace of P is preceded by the occurrence of some element of R. The corresponding trace constraint is as follows:

$$T \texttt{ authenticates } R \quad \triangleq \quad tr\restriction R = \langle\rangle \Rightarrow tr\restriction T = \langle\rangle.$$

For a single trace, the above formula does not specify that events of R occur before events of T. However, if tr is a trace of a process P such that

$$P \quad \textbf{sat} \quad T \texttt{ authenticates } R,$$

then not only tr but also all its prefixes satisfy the trace constraint. Hence, if an event of T occurs in tr, it is preceded by an event of R.

Different authentication properties can all be formally expressed as "T authenticates R" for adequately chosen sets T and R [28]. In order to check that a given protocol ensures such a property we have first to specify the protocol as a family of processes $USER(i)$ and then to show that

$$NET \quad \textbf{sat} \quad T \texttt{ authenticates } R. \tag{1}$$

Informally this requires that traces of the network which do not contain events of R do not contain events of T either. The traces of the process

$$NET\,|[R]|\,Stop$$

are easily seen to be all the traces of NET in which no event of R occurs. It follows that condition (1) is equivalent to

$$NET \,|[R]|\, Stop \quad \textbf{sat} \quad tr\restriction T = \langle\rangle. \tag{2}$$

The latter property can be verified using a rank function as shown in [28]. The idea is to assign to every message m an integer $\rho(m)$ called its rank in such a way that messages in T have non positive rank while only messages of positive rank can be produced by $NET \,|[R]|\, Stop$. The main problem is then to show that the property "messages have positive rank" is invariant for the process $NET \,|[R]|\, Stop$. Due to the structure of the network, this invariance property can be decomposed into a local property of each user and a property of the enemy. Using the generic definition of $ENEMY$, the verification can be further decomposed into constraints between the rank function ρ, the set of messages $INIT$, and the relation $\vdash$.

More precisely, let $\mathcal{M}$ be the message space for a given protocol. A rank function ρ is simply a function from $\mathcal{M}$ to the integers. We denote by ρ^+ the set of messages of positive rank:

$$\rho^+ \quad = \quad \{m \in \mathcal{M} \mid \rho(m) > 0\}.$$

Since every enemy event is a communication with some user, preventing users from generating events of R is enough to ensure that the whole network does not produce events of R. Formally, this corresponds to the following identity

$$NET \,|[R]|\, Stop \quad = \quad (\,|||_i\, USER(i) \,|[R]|\, Stop\,) \,|[trans, rec]|\, ENEMY.$$

The lemma below relies on this equation and gives two conditions sufficient to ensure that all messages produced by $NET \,|[R]|\, Stop$ are of positive rank.

Lemma 1 *If the two following conditions hold*

(a1) $ENEMY$ **sat** $M(tr\restriction trans) \subseteq \rho^+ \Rightarrow M(tr\restriction rec) \subseteq \rho^+$,

(a2) for all i, $USER(i) \,|[R]|\, Stop$ **sat** $M(tr\restriction rec) \subseteq \rho^+ \Rightarrow M(tr\restriction trans) \subseteq \rho^+$,

then

$$NET \,|[R]|\, Stop \textbf{ sat } M(tr) \subseteq \rho^+.$$

Intuitively, conditions (a1) and (a2) state that when events of R are prevented, the enemy and the users cannot send messages of non-positive rank if they only receive messages of positive rank. By induction, this implies all the messages circulating in the network have positive rank. The lemma is similar to Abadi and Lamport's decomposition technique for invariant properties based on assumption/guarantee specifications [1].

By definition of the $ENEMY$, condition (a1) holds if the two following relations are satisfied:

(b1) $INIT \subseteq \rho^+$,

(b2) $\forall S, m.\ S \subseteq \rho^+ \wedge S \vdash m \Rightarrow \rho(m) > 0$.

Condition (b1) means that only messages of positive rank are initially known to the enemy and (b2) that without the knowledge of some message of non-positive rank, only message of positive rank can be generated.

For convenience, we introduce the following abbreviation:

$$\texttt{maintains}\ \rho \quad \triangleq \quad M(tr\restriction rec) \subseteq \rho^+ \Rightarrow M(tr\restriction trans) \subseteq \rho^+,$$

and the essential theorem below gives four sufficient conditions for an authentication property to be satisfied.

Restrict.1	$Stop \ \|[R]\| \ Stop = Stop$
Restrict.2	$(a \rightarrow P) \ \|[R]\| \ Stop = Stop \quad \text{if } a \in R$
Restrict.3	$(a \rightarrow P) \ \|[R]\| \ Stop = a \rightarrow (P \ \|[R]\| \ Stop) \quad \text{if } a \notin R$
Restrict.4	$(P \ \|\|\| \ Q) \ \|[R]\| \ Stop = (P \ \|[R]\| \ Stop) \ \|\|\| \ (Q \ \|[R]\| \ Stop)$

Figure 2: Examples of restriction rules

Theorem 1 *If there exists a rank function ρ such that*

(c1) $INIT \subseteq \rho^+$,

(c2) $\forall S, m.\ S \subseteq \rho^+ \wedge S \vdash m \Rightarrow \rho(m) > 0$,

(c3) $T \cap \rho^+ = \emptyset$,

(c4) $\forall i.\ USER(i) \ |[R]| \ Stop$ **sat** `maintains` ρ,

then

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

In practice, using this theorem requires one to find a rank function which satisfies the four conditions (c1) to (c4). This function depends both on the authentication property to be verified and on the protocol description. Conditions (c1) to (c3) are usually easy to check; they do not involve any form of reasoning about processes. On the other hand, condition (c4) is a property of CSP processes which has to be checked for all the protocol participants. Specialised proof rules for CSP are required. Figures 2 and 3 show several examples of such rules. The rules of Fig. 2 allow one to manipulate and simplify process expressions of the form $P \ |[R]| \ Stop$; Fig. 3 shows proof rules used to establish that a particular process satisfies `maintains` ρ.

3 An Embedding of CSP in PVS

Using rank functions considerably simplifies the verification of authentication properties; it is much easier to check the four conditions of Theorem 1 than to directly prove that a network satisfies "T authenticates R". In particular, the theorem allows us to consider the protocol participants individually instead of having to manipulate a large and complex CSP process. Still, finding a rank function for checking a particular property of a specific protocol is not trivial. Rank functions have to satisfy multiple constraints which arise from conditions (c1) – (c3) of Theorem 1 and from the protocol specifications. Many of these constraints are not explicit but appear indirectly during the verification of condition (c4). In practice, it is difficult not to omit some of these constraints and checking manually that they are all satisfied is particularly prone to errors.

In order to overcome these difficulties, some form of mechanical support is desirable. A tool can help produce all the constraints a rank function must satisfy and can detect errors or omissions during proofs. There exist specialised proof tools for CSP such as the model checker FDR [26, 10] which has been used in various applications, including the analysis of security protocols [17, 16]. FDR is a powerful tool

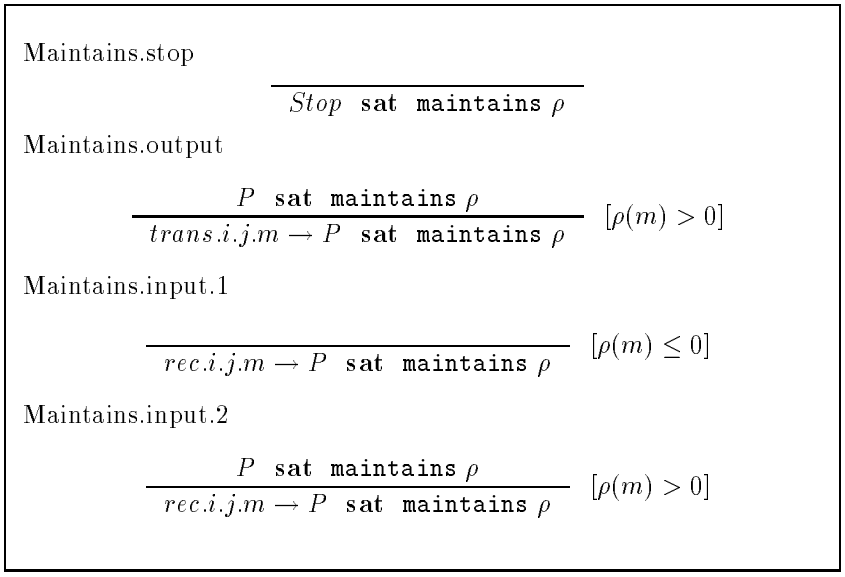

Figure 3: Examples of `maintains` rules

but it has some limitations; it does not handle arbitrary CSP processes but only those which can be represented by finite automata and it does not provide much facility for numerical reasoning. A theorem prover allowing us to reason about arbitrary processes and to handle numerical functions and properties is better suited to our approach.

In this section, we show how the necessary mechanical support can be provided by the PVS theorem prover. PVS is an interactive prover based on a form of higher order logic and is largely similar to other systems such as HOL [12], Isabelle/HOL [25] or IMPS [9]. Nonetheless, PVS supports a richer type system than standard higher order logic (as presented in [2] for example) and relies on an original approach to type checking. The PVS logic includes subtypes and dependent types and these features render type checking undecidable; the type checker may require users to show that their specifications are type consistent by generating proof obligations known as TCCs[1]. Several examples of specifications will illustrate this mechanism in the sequel. Other characteristics of PVS will also play an important role notably the presence of powerful decision procedures and the possibility of defining abstract data types. Many aspects of PVS are not covered in this document; a more complete description of the language and prover as well as examples of applications can be found elsewhere [22, 29].

In a first step, we need to develop a PVS representation of the CSP notion of processes. Since we are concerned exclusively with CSP trace semantics, an obvious choice is to represent processes by prefix-closed sets of traces. Traces themselves can be simply considered as lists of events. The definition of lists in PVS is then a crucial element of our model. It also illustrates general mechanisms which apply to other abstract data types and will be used in the sequel.

3.1 Lists

In PVS, list is a pre-defined notion; the system includes a generic type `list[T]` where `T` is a type parameter. Specific instances such as lists of real numbers or lists

[1] Type Correctness Conditions.

of booleans are denoted by `list[real]` or `list[bool]`. Lists are an example of recursive data structures which are defined in PVS as abstract data types:

```
list [T: TYPE]: DATATYPE
  BEGIN
    null: null?
    cons(car: T, cdr: list): cons?
  END list.
```

This declares the data type `list` with the usual constructors `null` and `cons` and the two accessors `car` and `cons`. The data type declaration also includes two recognisers `null?` and `cons?` which characterise empty and non-empty lists respectively.

From such a specification, PVS synthesizes a theory which contains an axiomatic definition of the data type. For PVS, a theory is simply a specification module which can contain a finite collection of type and constant declarations, definitions, theorems, and axioms. The theory derived from the list specification is called `list_adt`. It has a single parameter `T` – the same as the data type – and declares the type `list` and the associated constructors, accessors, and recognisers. The theory then contains various axioms which specify relations between accessors, constructors, and recognisers.

A fragment of `list_adt` is shown in Fig. 4. The theory starts by declaring the type `list` and the two recognisers `null?` and `cons?`. The latter are two functions of type `[list -> boolean]` and implicitly define two subtypes of `list` denoted by `(cons?)` and `(null?)`. The former is intended to be the type of non-empty lists; the type expression `cons?` is actually an abbreviation for the sub-type

```
{l : list | cons?(l)}
```

and has the same set-theoretical interpretation. An object `l` is of type `(cons?)` if it is of type `list` and satisfies the predicate `cons?`. Similarly, `(null?)` is an abbreviation for

```
{l : list | null?(l)}
```

and is intended to be the type of the empty list. The other components of the data type specification are declared as functions of the appropriate domains and ranges as indicated in Fig. 4.

Some of the axioms of `list_adt` are given in Fig. 4 (for readability, the variable names have been changed). There is an extensionality axiom for each constructor and an axiom to describe the effect of accessors. Among the other axioms, the induction rule `list_induction` is of course essential. In addition to the axioms shown, the theory defines various predicates on lists. In particular all data types are equipped with a well founded relation denoted by `<<` such that `l1 << l2` is true when `l1` is a strict sub-term of `l2`. In the case of lists, `null` has no strict sub-terms and the sub-terms of a list `cons(x, l)` are `l` and all the sub-terms of `l`. The data type axiomatisation is not minimal, for example the eta rule `list_cons_eta` can be derived from extensionality and other axioms; similarly the induction axiom can be derived from the fact that `<<` is well founded.

Many of the axioms of Fig. 4 are known to the prover and are automatically applied as rewrite rules during proofs. For example, the following lemma is a simple consequence of `list_car_cons` and `list_cdr_cons` and can be proved by invoking the decision procedures.

```
test: LEMMA
   car(cdr(cons(x,y))) = car(cdr(cons(x, cons(car(y),l))))
```

```
list_adt[T: TYPE]: THEORY
  BEGIN
    list:  TYPE

    null?, cons?:  [list -> boolean]

    cons:  [[T, list] -> (cons?)]

    car :  [(cons?)  -> T]

    cdr :  [(cons?)  -> list]

    list_null_extensionality:  AXIOM
      FORALL (l1, l2:(null?)):  l1 = l2

    list_cons_extensionality:  AXIOM
      FORALL (l1, l2:(cons?)):
        car(l1) = car(l2) AND cons(l1) = cons(l2) IMPLIES l1 = l2

    list_cons_eta:  AXIOM
      FORALL (l:(cons?)):  cons(car(l), cdr(l)) = l

    list_car_cons:  AXIOM
      FORALL (x:T, l:list):  car(cons(x, l)) = x

    list_cdr_cons:  AXIOM
      FORALL (x:T, l:list):  cdr(cons(x, l)) = l

    list_inclusive:  AXIOM
      FORALL (l:list):  null?(l) OR cons?(l)

    list_induction:  AXIOM
      FORALL (p:[list -> boolean):
        p(null) AND
          (FORALL (x:T, l:list):  P(l) IMPLIES P(cons(x, l)))
        IMPLIES (FORALL (l:list):  P(l))

    ...

  END list_adt
```

Figure 4: Axiomatic specification of the list data type

```
test :
  |-------
{1} (FORALL (l:  list[T], x:  T, y:  (cons?[T])):
      car(cdr(cons(x, y))) = car(cdr(cons(x, cons(car(y), l)))))

Rule?  (assert)
Simplifying, rewriting, and recording with decision procedures,
Q.E.D.
```

Figure 5: A simple proof using decision procedures.

The proof is shown in Fig. 5. It contains only one sequent – the initial goal – and a single proof command which simply calls the decision procedures. In this case, the two sides of the equation simplify to `car(y)` and the prover concludes that the equality holds.

In addition to the list data type, PVS predefines various functions on lists such as `length`, `reverse`, and `append`. The function `append` which concatenates two lists is defined as follows:

```
l1, l2: VAR list[T]

append(l1, l2): RECURSIVE list[T] =
  CASES l1 OF
    null: l2,
    cons(x, y): cons(x, append(y, l2))
  ENDCASES
 MEASURE length(l1).
```

This example illustrates how recursive functions can be defined over abstract data types. The `CASES` construct is used for pattern matching and the `MEASURE` clause to ensure that the function is well-defined. In general, all recursive definitions must include a measure clause. In most cases, the measure is an expression of a well-founded type – usually the natural numbers – and PVS requires the user to prove that the measure is strictly decreasing at each recursive call. This guarantees that the function "always terminates". For the example above, PVS generates the following proof obligation:

```
append_TCC1: OBLIGATION
   FORALL (x: T, y:list[T], l1, l2):
      l1 = cons(x, y) IMPLIES length(y) < length(l1).
```

The function `length` is defined recursively like `append` but the measure is different:

```
length(l) : RECURSIVE nat =
  CASES l OF
    null: 0,
    cons(x, y): 1 + length(y)
  ENDCASES
 MEASURE reduce_nat(0, (lambda (x : T), (n : nat) : n + 1)).
```

This time the measure is defined with the `reduce_nat` combinator. Such a combinator is automatically constructed by PVS in the same way as data type axioms are generated. For lists, `reduce_nat` is a function of type

```
[nat, [T, nat -> nat] -> [list[T] -> nat]]
```

which satisfies the two identities:

```
reduce_nat(a, f)(null) = a,
reduce_nat(a, f)(cons(x, y)) = f(x, reduce_nat(a, f)(y)).
```

Although length could be defined directly using `reduce_nat` it is more convenient to have a recursive definition. The prover has special features for handling cases expressions and recursive functions which do not apply to `reduce_nat`.

An alternative approach to defining `length` could be to use the well founded relation `<<`:

```
length(l): RECURSIVE nat =
  CASES l OF
    null: 0,
    cons(x, y): 1 + length(y)
  ENDCASES
MEASURE l BY <<.
```

In this definition, the measure clause simply specifies the relation to be used for showing termination. PVS produces two proof obligations, the first to check that the relation is well founded, the second to show that arguments to recursive calls are decreasing. For example, the definition above would yield

```
length_TCC1: OBLIGATION
  well_founded?(lambda (x, y:list[T]) : x << y)

length_TCC2: OBLIGATION
  FORALL (x: T, y:list[T], l): l = cons(x, y) IMPLIES y << l.
```

Both TCCs follow immediately from the definition of `<<` and the axiom which states that `<<` is well founded.

The set of predefined list functions is relatively rich but it was convenient to extend it with a new function and various theorems and lemmas. The most important extensions are given in Fig. 6. We defined a function `add` which appends an element `a` to a list `l` and derived an associated induction theorem. The theorem is similar to the pre-defined `list_induction` except that it works in the reverse order. This is a key simplification when reasoning about prefix-closed sets. Proposition `add_induction` is itself proved by induction on the length of `l` using lemmas such as those shown in Fig. 6.

3.2 Basic CSP

PVS allows theories to be parameterised as were `list_adt` and `more_list_prop` above. The type `list` and all the functions from `list_adt` depend implicitly on the parameter `T`. As a consequence, `list` is a generic type; to every instantiation of `T` corresponds a type `list[T]`. We can use this facility to define generic types representing traces and processes. Given a type `T`, `trace[T]` and `process[T]` are the types of traces and processes with events of type `T`.

Theory `traces` shown in Fig. 7 contains the definition of `trace[T]` and operations on list which are related to CSP. The type `trace` is simply a synonym for

```
more_list_props[T: TYPE]: THEORY
  BEGIN
    l, l1, l2:  VAR list[T]
    cons_l:  VAR (cons?[T])
    a, b, c:  VAR T
    P: VAR [list[T] -> bool]
    ...
    add(l, a):  (cons?[T]) = append(l, cons(a, null))
    ...
    length_add:  LEMMA
      length(add(l, a)) = length(l) + 1
    ...
    length_non_zero:  LEMMA
      length(l) = 0 IFF l = null
    cons_to_add:  LEMMA
      FORALL cons_l:  EXISTS a, l:  cons_l = add(l, a)
    add_induction:  PROPOSITION
      P(null) AND (FORALL l, a:  P(l) IMPLIES P(add(l, a)))
        IMPLIES (FORALL l:  P(l))
  END more_list_props
```

Figure 6: Adding an element at the end of a list.

```
traces[T: TYPE]: THEORY
  BEGIN
    IMPORTING more_list_props

    trace :  TYPE = list[T]

    a, b, c:  VAR T

    A, B, C: VAR setof[T]

    t, t1, t2, t3:  VAR trace

    ...

    prefix(t1, t2):  bool = EXISTS t:  t2 = append(t1, t)

    prefix_equiv:  LEMMA
      prefix(t1, t2) IFF null?(t1)
        OR (cons?(t1) AND cons?(t2) AND car(t1) = car(t2)
          AND prefix(cdr(t1), cdr(t2)) )

    null_prefix:  LEMMA prefix(null, t)

    prefix_null:  LEMMA prefix(t, null) IFF t = null

    ...

    proj:  [trace, set[T] -> trace] = filter

    proj_null:  LEMMA proj(null, A) = null

    proj_cons:  LEMMA
      proj(cons(a, t), A) =
        IF A(a) THEN cons(a, proj(t, A)) ELSE proj(t, A) ENDIF
    ...

    sigma :  [trace -> setof[T]] = list2set

    ...

    sigma_null :  LEMMA sigma(null) = emptyset

    sigma_cons :  LEMMA sigma(cons(a, l)) = add(a, sigma(l))

    END traces
```

Figure 7: Traces, Prefix and Projection.

`list[T]`. Projection and prefix are two important operations on `traces` useful for defining processes.

The prefix relation is defined using `append` as indicated in Fig. 7 and lemma `prefix_equiv` gives an equivalent recursive formulation.

PVS already defines a function `filter` which computes the projection of a list on a set. The two lemmas `proj_null` and `proj_cons` correspond to the recursive definition of `filter`. For PVS, the types `setof[T]` or `set[T]` are the same as `[T -> bool]`, that is, PVS represents sets by their characteristic functions. Hence, the boolean expression `A(a)` in `proj_cons` can be interpreted as "`a` is an element of the set `A`".

Once the notions of traces and prefix are specified it is straightforward to define processes. The type `process[T]` is a subtype of `set[trace[T]]`; every process is a prefix-closed set of traces which contains at least the empty trace `null`. This is clearly the same as requiring processes to be non-empty and prefix-closed. The type definition is shown in Fig. 8.

Like any PVS set, a process is also a boolean function. The statement `t` *is a trace of* `P` can written either `member(t, P)` or `P(t)`; the two expressions are equivalent. Furthermore, as a predicate, a process `P` defines a sub-type of `traces` in the same way as list recognisers define sub-types of `list`. The type `(P)` is the type of traces which satisfy the predicate `P`, that is, the type of traces which belong to the set `P`. For example, the formula

```
EXISTS (t: (P)): length(t) > 0
```

is satisfied if `P` contains a non-empty trace.

The definitions of the primitive processes and operators are direct translations of the CSP trace semantics: the only trace of *Stop* is the empty trace, the set of traces of $P_1 \Box P_2$ is the union of the traces of P_1 and of P_2, etc. Most primitives are specified easily in PVS using pre-defined set and list operations. The only complication comes from the parallel composition $P_1 \,|[A]|\, P_2$. In order to describe this construct, we use a specific predicate on traces denoted by `prod`. Given a set of events `A` and three traces `t1`, `t2`, and `t`, the expression

```
prod(A)(t1, t2, t)
```

is true if `t1` and `t2` can be executed concurrently by two processes which synchronise on A, and `t` can be observed as a result. The function `prod` is of type

```
[setof[T] -> [trace, trace, trace -> boolean]]
```

and is defined by recursion on `t` as follows.

```
prod(A)(t1, t2, t): RECURSIVE bool =
  CASES t OF
    null: null?(t1) AND null?(t2),
    cons(x, y):
      IF A(x)
      THEN
        cons?(t1) AND cons?(t2) AND car(t1)=x AND car(t2)=x
        AND prod(A)(cdr(t1), cdr(t2), y)
      ELSE
         (cons?(t1) AND car(t1)=x AND prod(A)(cdr(t1), t2, y))
      OR (cons?(t2) AND car(t2)=x AND prod(A)(t1, cdr(t2), y))
      ENDIF
  ENDCASES
 MEASURE length(t)
```

Three cases are distinguished: either the traces `t1`, `t2`, and `t` are all empty, or they all start by the same event of `A`, or `t` and one of `t1` and `t2` start by an event which does not belong to `A`. This corresponds to the rules of synchronisation between two processes: events of `A` have to be performed simultaneously while events not in `A` have to be performed independently.

With the previous function, the parallel composition of two processes which synchronise on `A` is defined by:

```
Par(A)(P1, P2): process =
  { t | EXISTS (t1:(P1)), (t2:(P2)): prod(A)(t1, t2, t) }.
```

A trace `t` can be performed by the parallel composition if there is a trace `t1` of `P1` and a trace `t2` of `P2` such that `prod(A)(t1, t2, t)` holds.

The function `Par` and the other primitive operators are defined in the theory `processes` shown in Fig. 8. The theory also includes a function `sigma` which yields the set of events a process may perform.

As illustrated by the parallel composition operator, the syntax of process expressions are different in CSP and in PVS. Ideally, we would like to be able to use in PVS a syntax close to the standard notation. Unfortunately, the syntax of PVS is fixed; a certain number of symbols can be used as operators but the user cannot introduce new symbols or new syntactic forms. We have approximated most of CSP's operators by existing PVS symbols but there was nothing similar to the parallel composition syntax: $P_1|[A]|P_2$. Other operations such as the unbounded choice also required a different PVS notation. The correspondence between the two notations is given in table 1.

The type system ensures that all the definitions in theory `processes` are consistent. For example, when type checking the definition of `Par`, PVS generates a TCC to ensure that the given set of traces is effectively a process:

```
Par_TCC1: OBLIGATION
  FORALL (A, P1, P2):
     EXISTS (t1: (P1)), (t2: (P2)): prod[T](A)(t1, t2, null[T])
AND prefix_closed(
      {t | EXISTS (t1: (P1)), (t2: (P2)): prod[T](A)(t1, t2, t)}).
```

This TCC requires us to show that the two clauses of the definition of `process` are satisfied. The first half is easy; the non-trivial part is to show that the set is prefix-closed. The proof relies on the fact that both `P1` and `P2` are prefix-closed and on the following property of `prod`:

```
prefix_prod: LEMMA prod(A)(t1, t2, t) AND prefix(u, t) IMPLIES
  EXISTS u1, u2: prefix(u1, t1) AND prefix(u2, t2)
         AND prod(A)(u1, u2, u).
```

This lemma is easily proved by induction on `u` and using simple properties of `prefix`.

3.3 Parametric processes

The various constructs presented so far cover all the CSP primitives except the unbounded choice. The latter is usually defined as an operation on sets of processes: given a family of processes $\{P_i | i \in I\}$, the set of traces of $\Box_{i \in I} P_i$ is the union of the sets of traces of all the P_i. We can define a similar operator in PVS:

```
SP: VAR setof[process[T]]

Choice(SP): process[T] = {t | null?(t) OR EXISTS (P:(SP)): P(t)}.
```

```
processes[T: TYPE]: THEORY
  BEGIN
    IMPORTING traces

    S: VAR setof[trace[T]]

    t, t1, t2 :  VAR trace[T]

    prefix_closed(S): bool =
      FORALL t1, t2:  prefix(t1, t2) AND S(t2) IMPLIES S(t1)

    process:  TYPE = { S | S(null) AND prefix_closed(S) }

    P, P1, P2:  VAR process

    a:  VAR T

    A: VAR setof[T]


    Stop:  process = { t | t = null };

    \/(P1, P2):  process = union(P1, P2);

    >> (a, P): process =
      { t | null?(t) OR EXISTS (t1:(P)): t = cons(a, t1) }

    Par(A)(P1,P2):  process =
      { t | EXISTS (t1:(P1)), (t2:(P2)):  prod(A)(t1, t2, t) };

    //(P1, P2):  process = Par(emptyset)(P1, P2)


    sigma(P): setof[T] = { a | EXISTS (t :(P)): sigma(t)(a) }

  END Processes
```

Figure 8: Basic Process Constructs

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	$P_1 \,\|[A]|\, P_2$	`Par(A)(P1, P2)`
	$P_1 \,\|\|\|\, P_2$	`P1 // P2`
	$\|\|\|_{i \in I} P_i$	`Interleave!  i :  P(i)`

Table 1: Syntax of process expressions.

The function takes a set of processes as argument and yields a single process. The definition allows `SP` to be empty and in this case, the resulting process contains only the empty trace. This is consistent with the fact that *Stop* is the neutral element of choice. If `SP` is not empty, `Choice(SP)` is the union of the processes which belong to `SP`.

In practice, the above function is not very convenient. As the CSP notation suggests, unbounded choice is most of the time used with an indexed family of processes. It is easy to define another `Choice` operator which allows us to do the same in PVS:

```
P: VAR [U -> process[T]]

Choice(P): process[T] = {t | null?(t) OR EXISTS i : P(i)(t)}.
```

This new `Choice` function has the following polymorphic type

```
[[U -> process[T]] -> process[T]]
```

where the two parameters `U` and `T` are the types of indices and of events, respectively. The argument of `Choice` is a function from `U` to `process[T]`, that is, a family of processes indexed by elements of `U`. We also say that `P` is a parametric process.

PVS supports overloading and the two `Choice` functions can (and do) co-exist but several features make the second easier to manipulate.

A special syntax is available to abbreviate the application of functions such as the second version of `Choice` to lambda expressions. For example, a term such as

```
Choice(lambda i: P(i) // Q(i))
```

can be written as

```
Choice! i: P(i) // Q(i)
```

which is reasonably close to the CSP term $\Box_{i \in I}\ P_i \ ||| \ Q_i$. With the first form of choice, one would have to write

```
Choice({ R | EXISTS i: R = P(i) // Q(i) }).
```

The special syntax for application can be used with any function which has a single functional parameter.

Although the argument to `Choice` is of type `[U ->process[T]]`, the operator also applies to multiple indexed families. PVS considers that the types `[U1,...,Un -> process[T]]` and `[[U1,...,Un] -> process[T]]` are identical. A function with n arguments can also be seen as a function with a single n-tuple argument. As a consequence, choice expressions with several index variables such as

```
i, j: VAR nat

Choice! i, j: P(i, j) // Q(i)
```

are valid. PVS automatically infers the right instantiation for the parameter `U`, namely the type `[nat, nat]` of pairs of natural numbers. It also finds the right instantiation for `T` using the types of `P` and `Q`.

The form of dependent types supported by PVS also gives the possibility to specify constraints on index ranges. For example, we can define a process which produces two successive events of type `nat` with the second at least as large as the first:

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

In this example, the parameter `U` is instantiated with the dependent type:

```
[i:nat, {j:nat | i <= j}],
```

that is, the type of pairs of natural numbers where the second component is larger than or equal to the first.

Unbounded choice is a generalisation of binary choice to arbitrary families of processes. It is also convenient to be able to apply other operators to more than two processes. For example, the definition of $NETWORK$ in section 2.2 applies the operator $|||$ to the family of processes $USER(i)$. No new CSP operation is involved but the expression

$$|||_i \, USER(i)$$

simply denotes the iterated application of $|||$ to a *finite* set of processes. We could adopt the same approach with PVS by defining, for, example a function which iterate associative operators on finite sequences of processes. However, this method would be quite restrictive and could rapidly lead to extra type checking difficulties.

Instead, it is preferable to generalise the parallel composition to arbitrary – possibly infinite – sets of processes. This non-standard extension of CSP does not pose any theoretical problem in the traces model and in fact generalises the results presented in section 2. We can consider networks with infinitely many users and all the theorems still hold. Moreover, the PVS statement and proofs of these theorems are much simpler if infinite parallel composition is allowed.

The basis of the new operator is an extension of the predicate `prod(A)` which is defined as shown in Fig. 9. The new predicate works like the simpler `prod` but its first argument (`t`) is a family of traces. Such a family is a function of type `[U -> traces[T]]` where the parameters `U` and `T` play the same role as in the definition of unbounded choice. The only difference is that the index type is required to be non-empty (cf. Fig. 9); this is only a slight restriction and greatly simplifies certain results. The new predicate `prod` is of type

```
[set[T] -> [[U -> trace[T]], trace[T] -> bool]];
```

and `prod(A)(t, u)` holds if all the traces `t(i)` are compatible and `u` is a possible result of their combination. The definition is by recursion on `u` and includes the same three cases as the simpler `prod`: either all the traces `t(i)` and `u` are empty, or they all start with the same event of `A`, or one of the traces `t(i)` and `u` start with the same event which does not belong to `A`.

The unrestricted parallel composition is defined using the new `prod` as shown in Fig. 9. A trace `u` belongs to the process `Par(A)(P)` if there is a family `t` of traces such that each `t(i)` is a trace of `P(i)` and `prod(A)(t, u)` is true. This is expressed compactly using a dependent type `[i:U -> (P(i))]` (cf. Fig. 9). We can then introduce a function `Interleave` which extends the operator `//` to arbitrary families of processes. `Interleave` forms the parallel composition of processes which do not synchronise on any event.

Both `Par(A)` and `Interleave` have a single functional argument and the same notation as above can be used; expressions such as

```
Interleave! i, j: P(i,j)
```

are allowed.

3.4 Properties of processes

From the semantic definitions of processes and operators, we can derive many important properties. Figure 10 gives a sample of rules which are useful for simplifying

```
multiprod[U: NONEMPTY_TYPE, T: TYPE]: THEORY
  BEGIN
  IMPORTING traces, ...
  i:  VAR U
  t:  VAR [U -> traces[T]]
  A: VAR set[T]
  u:  VAR traces[T]

  ...

  prod(A)(t, u):  RECURSIVE bool =
    CASES u OF
      null:  (FORALL i:  t(i) = null),
      cons(x, y):
        IF A(x) THEN
          (FORALL i:  cons?(t(i)) AND car(t(i)) = x)
            AND prod(A)(lambda i :  cdr(t(i)), y)
        ELSE
          EXISTS i:  cons?(t(i)) AND car(t(i)) = x
            AND prod(t with [(i) := cdr(t(i))], y)
        ENDIF
    ENDCASES
   MEASURE length(u)

  ...

  END multiprod

multipar[U: NONEMPTY_TYPE, T: TYPE]: THEORY
  BEGIN
  IMPORTING process, multiprod, ...
  i:  VAR U
  P: VAR [U -> process[T]]
  A: VAR set[T]
  u:  VAR trace[T]

  Prod(A)(P): process[T] =
    { u | EXISTS (t:  [i:U -> (P(i))]):  prod(A)(t, u) }

  Interleave(P): process[T] = Par(emptyset)(P)

  ...

  END multipar
```

Figure 9: Extended Parallel Composition

```
choice_commutes:  LEMMA (P1 \/ P2) = (P2 \/ P1)

choice_assoc:  LEMMA (P1 \/ (P2 \/ P3)) = ((P1 \/ P2) \/ P3)

choice_idempotent:  LEMMA (P \/ P) = P

choice_stop1:  LEMMA (P \/ Stop[T]) = P

choice_stop2:  LEMMA (Stop[T] \/ P) = P

par_commutes:  LEMMA Par(A)(P1, P2) = Par(A)(P2, P1)

par_assoc:  LEMMA
    Par(A)(P1, Par(A)(P2, P3)) = Par(A)(Par(A)(P1, P2), P3)

par_stop:  LEMMA
    Par(A)(P, Stop) = { t | P(t) AND proj(t, A) = null }

par_full:  LEMMA Par(fullset)(P1, P2) = intersection(P1, P2)
```

Figure 10: Examples of Algebraic Rules

process expressions: choice is associative, commutative, and idempotent; *Stop* is neutral element for choice; the parallel composition is associative and commutative; etc. Most of these properties are classic laws of CSP's process algebra but the collection is far from complete. Other lemmas could be included, notably various distributivity properties; we did not need them for reasoning about authentication protocols.

We also included properties not directly expressed as CSP identities but where processes are taken from a set-theoretical point of view: the process $P \; |[A]| \; Stop$ is the subset of P whose traces do not contain events of A; two processes which synchronise on all events behave like their intersection; etc. The following property is of a similar nature and is an important lemma for proving the main theorems about networks:

```
interleave_disjoint: LEMMA
      (FORALL i: subset?(sigma(P(i)), S(i)))
  AND (FORALL i, j: i/=j IMPLIES disjoint?(S(i), S(j)))
IMPLIES
  subset?(Interleave(P), { u | FORALL i: P(i)(proj(u, S(i))) }).
```

Informally, this says that if all the processes P_i have mutually disjoint interfaces then the projection of any trace of $(|||_i P_i)$ on P_j's interface is a trace of P_j. In the lemma, process interfaces are characterised by sets of events: each set S_j contains all the events P_j may perform.

In addition to the previous lemmas, we need to specify and verify safety properties. It is consistent with our semantical approach to write such properties directly with the PVS logic. A priori, there is then no need to develop any particular construct for describing safety properties. For example, we can translate a CSP statement

$$P \quad \textbf{sat} \quad tr \restriction D = \langle\rangle$$

to the PVS formula

```
FORALL (tr: (P)): proj(tr, D) = null.
```

However, adding an extra PVS construct similar to the **sat** operator has several advantages. It improves readability by clearly separating processes and properties and can make PVS specification look more familiar to CSP users. More importantly, a satisfaction operator allows us to translate CSP proof rules into PVS rewrite rules which are convenient in mechanical verifications.

Safety properties are predicates on traces and can be considered as functions of type `[trace[T] -> bool]` or equivalently as constants of type `pred[trace[T]]` or `set[trace[T]]`; the three types are identical. As previously, we cannot use the exact CSP syntax but we have to choose a symbol among the existing PVS operators. We use the operator `|>` to denote satisfaction. The example above – the statement that P does produce events of D – translates then to[2]

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

The definition of `|>` is trivial; P satisfies a predicate E if P is a subset of E. In our framework, E can be any set of traces and, in particular, E can be a process. In this case, the symbol `|>` can be interpreted as the classic refinement relation between CSP processes (see [13]).

Figure 11 shows the definition of the satisfaction relation and a few obvious but useful properties. Lemmas `sat_choice1` or `sat_choice3` are two examples of PVS rewrite rules; they are the PVS counterparts of general proof rules of CSP. The condition `Stop[T] |> E` in lemma `sat_choice3` may seem superfluous but it is necessary in case `U` is an empty type. In the sequel, we will specialise the rule to particular classes of predicates `E` and the extra condition will vanish.

3.5 Fixed points and induction

The previous constructs allow us to write in PVS any process expression, to show certain equivalences between processes, and to specify safety properties. PVS allows us to define directly non-recursive processes, such as

```
E: process[T] = a >> ((b >> Stop) \/ (c >> Stop)),
```

but definition of processes by systems of equations as shown in section 2 is not directly supported. For example, we cannot define the process $COUNT$ of that section by

```
Count(n): process[T] =
  IF n = 0 THEN up >> Count(1)
  ELSE  (up >> Count(n+1)) \/ (down >> Count(n-1))
  ENDIF.
```

Since unrestricted recursion is not sound in general, such a definition is not valid in PVS. Using the allowed form of recursive definition does not work either because no strictly decreasing measure can be associated with the parameter n. Instead, the existence of a parametric process which satisfy the above relation can be justified using Tarski's fixed point theorem. $COUNT$ is the least fixed point of the mapping H defined by the two equations

$$\begin{aligned} H(X)(0) &= up \rightarrow X(1) \\ H(X)(n+1) &= up \rightarrow X(n+2) \,\Box\, down \rightarrow X(n), \end{aligned}$$

[2] Provided `tr` has been previously declared as a trace variable.

```
P, Q: VAR process[T]

E, F: VAR pred[trace[T]]

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

sat_transitive1:  LEMMA
    (P |> Q) AND (Q |> E) IMPLIES (P |> E)

sat_transitive2:  LEMMA
    (P |> E) AND subset?(E, F) IMPLIES (P |> F)

sat_choice1:  LEMMA (P \/ Q) |> E IFF P |> E AND Q |> E

sat_free_par1:  LEMMA (P // Q) |> E IMPLIES P |> E

...

R: VAR [U -> process[T]]

i:  VAR U

sat_choice3:  LEMMA
  Choice(R) |> E IFF Stop[T] |> E AND (FORALL i:  R(i) |> E)
```

Figure 11: Satisfaction Relation and Immediate Properties

where X is any mapping from natural numbers to processes. In order to manipulate recursive processes in PVS, we define a least fixed point operator `mu` so that we can write

```
H(X)(n): process[T] =
  IF n = 0 THEN up >> X(1)
  ELSE  (up >> X(n+1)) \/ (down >> X(n-1))
  ENDIF

Count : [nat -> process[T]] = mu(H).
```

The fixed point construction for such parametric processes is shown in Fig. 12. We define an order relation `<=` between parametric processes as shown in the figure. For this partial order, any set `SX` of parametric processes has a greatest lower bound `glb(SX)`. Any monotonic mapping `G` between parametric processes has a least fixed point `mu(G)` defined classically as the greatest lower bound of {`X | G(X) <= X`}. The main properties of `glb` and `mu` are given in Fig. 12; the proofs are all straightforward. A simpler fixed point operator is also defined which handles the case of non-parametric processes.

The operator `mu` is generic and can be used for any recursive parametric process but `mu` only applies to monotonic mappings. The domain of the function `mu` is the subtype `(monotonic?)` of

```
[[U -> process[T]] -> [U -> process[T]]]
```

and PVS generates TCCs to ensure that the arguments to `mu` are of the right type. For example, the definition of `Count` above yields the following proof obligation:

```
Count_TCC1: OBLIGATION monotonic?[nat, T](H).
```

Monotonicity holds in general of any function which is built from the primitive CSP operators, that is, any `F` such that `F(X)` (or `F(X)(x)` in the case of parametric processes) is a CSP expression. However, there is no simple way to use or even formulate this general result in our semantic approach. The concept "a CSP expression" is a meta-theoretical notion which is not present in our formalisation. As a result, a monotonicity TCC is generated and has to be proved every time the `mu` operator is used. In simple cases, such TCCs can be discharged automatically by PVS default proof strategies but most of the time manual assistance is required.

It would be possible to define a PVS data type representing CSP expressions and this would enable the preceding general result to be proved and applied. However, this would require extra work and provide little benefit. It may also be possible to suppress the monotonicity TCCs by using the type system and adding new constructors which apply to monotonic mappings and yield monotonic mapping. Here again, it is not clear whether such a development is worthwhile.

Instead of such sophisticated techniques, we provide a collection of specialised lemmas which can prove monotonicity almost automatically. By examining the definition of the order relation in Fig. 12, one can see that showing that a function is monotonic amounts to proving sequents of the form

```
  FORALL x: subset?(X(x), Y(x))
|--------------
  FORALL x: subset?(F(X)(x), F(Y)(x)).
```

Provided `F(X)(x)` and `F(Y)(x)` expand to CSP expressions, we can prove the above sequent using the rules shown in Fig. 13. The lemmas are organised in different theories according to the number of type parameters required and to the non-emptiness assumption used. Using these rules, a possible proof of `Count_TCC1` is as follows:

```
fixed_points[U, T: TYPE]: THEORY
  BEGIN
  IMPORTING process, ...
  x, x1, x2:  VAR U
  X, Y, Z: VAR [U -> process[T]]
  t:  VAR trace[T]
  <=(X, Y): bool = FORALL x:  subset?(X(x), Y(x))
  ...
  SX: VAR set[[U -> process[T]]]
  glb(SX)(x):  process[T] = { t | FORALL (X: (SX)): X(x)(t) }
  glb_is_bound:  LEMMA FORALL (X: (SX)): glb(SX) <= X
  glb_is_inf:  LEMMA
    (FORALL (X: (SX)): Y <= X) IMPLIES Y <= glb(SX)
  F: VAR [[U -> process[T]] -> [U -> process[T]]]
  monotonic?(F): bool = FORALL X, Y: X <= Y IMPLIES F(X) <= F(Y)
  G: VAR (monotonic?)
  mu(G): [U -> process[T]] = glb({X | G(X) <= X})
  closure_mu:  LEMMA G(mu(G)) <= mu(G)
  smallest_closed:  LEMMA G(x) <= X IMPLIES mu(G) <= X
  fixed_point:  LEMMA G(mu(G)) = mu(G)
  least_fixed_point:  LEMMA G(X) = X IMPLIES mu(G) <= X
  ...
```

Figure 12: Least Fixed Point Operator

```
monotonicity[T: TYPE]: THEORY
  BEGIN
  ...

  monotonic_stop:  LEMMA subset?(Stop[T], P)

  monotonic_pref:  LEMMA
    subset?(a >> P, a >> Q) IFF subset?(P, Q)

  monotonic_choice:  LEMMA
    subset?(P1, Q1) AND subset?(P2, Q2) IMPLIES
        subset?(P1 \/ P2, Q1 \/ Q2)

  monotonic_choice2:  LEMMA
    (FORALL (P: (SP)): EXISTS (Q : (SQ)): subset?(P, Q))
      IMPLIES subset?(Choice(SP), Choice(SQ))

  monotonic_par:  LEMMA
    subset?(P1, Q1) AND subset?(P2, Q2) IMPLIES
        subset?(Par(A)(P1, P2), Par(A)(Q1, Q2))

  monotonic_free_par:  LEMMA
    subset?(P1, Q1) AND subset?(P2, Q2) IMPLIES
        subset?(P1 // P2, Q1 // Q2)

  END monotonicity

monotonicity2[U, T: TYPE]: THEORY
  BEGIN
  ...

  monotonic_choice3:  LEMMA
    (FORALL i:  subset?(P(i), Q(i))) IMPLIES
        subset?(Choice(P), Choice(Q))

  END monotonicity2

monotonicity3[U:NONEMPTY_TYPE, T:TYPE]: THEORY
  BEGIN
  ...

  monotonic_par2:  LEMMA
    (FORALL i:  subset?(P(i), Q(i))) IMPLIES
        subset?(Par(A)(P), Par(A)(Q))

  monotonic_free_par2:  LEMMA
    (FORALL i:  subset?(P(i), Q(i))) IMPLIES
        subset?(Interleave(P), Interleave(Q))

  END monotonicity3
```

Figure 13: Monotonicity Rules

```
param_induction:  PROPOSITION
      (FORALL x:  Stop[T] |> E(x))
  AND (FORALL X: (FORALL x:  X(x) |> E(x)) =>
                    (FORALL x:  G(X)(x) |> E(x)))
  IMPLIES (FORALL x:  mu(G)(x) |> E(x))

...

induction:  PROPOSITION
      Stop[T] |> E
  AND (FORALL X: X |> E => G(X) |> E)
  IMPLIES mu(G) |> E
```

Figure 14: Induction Rules

```
(auto-rewrite-theory "monotonicity[T]")
(expand "monotonic?")
(expand "H")
(expand "<=")
(reduce :if-match all).
```

The first step installs all the lemmas of Fig. 13 as automatic rewrite rules. The next three steps expand definitions. At this point, the goal is an implication similar to the sequent above and the rewrite rules can be applied. This is done by the last command which also introduces Skolem constants, applies propositional simplifications, and instantiates quantified variables.

The `mu` operator allows us to define recursive processes and we need induction rules for reasoning about such processes. Figure 14 gives the two most common forms of induction theorems corresponding to the parametric and non-parametric case. These two induction rules and their variants are derived from the definition of the fixed points operators. Just as the general rules of satisfaction for the choice operator, simpler induction rules will be obtained for particular classes of trace properties.

4 The Authentication Model in PVS

All the PVS elements presented in the previous section provide the basic theories for specifying and reasoning about general CSP processes. In this section, we examine how these tools can be applied to modelling the network and how the notions of rank functions, authentication and the associated theorems can be developed.

4.1 Events

The general network is modelled in section 2.2 as a process which produces events of the form

$$rec.i.j.m \quad \text{and} \quad trans.i.j.m$$

where i and j are user identities and m belongs to a set $\mathcal{M}$ of messages. In PVS, we represent such events using a parametric data type specified in Fig. 15. The `event` type has two parameters `I` and `M` corresponding to the types of user identities and of

```
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

t:  VAR trace[event]
i, j:  VAR I
m:  VAR M

receptions(t):  trace[event] = proj(t, rec?)

transmissions(t):  trace[event] = proj(t, trans?)

rec_msg(t):  set[M] =
    {m | EXISTS i, j:  member(rec(i, j, m), sigma(t))}

trans_msg(t):  set[M] =
    {m | EXISTS i, j:  member(trans(i, j, m), sigma(t))}
```

Figure 15: Event Type and Related Operations

messages, respectively. Transmission and reception events are of the form `trans(i, j, m)` and `rec(i, j, m)`, and are of type `(trans?)` and `(rec?)`, respectively. Unlike `list`, the data type `event` is not recursive but is used to represent the disjoint union of two types of triples. PVS generates a theory `event_adt` which contains similar axioms to `list_adt`.

For the specification of trace properties, we introduce various operations on traces and sets of events. These are specialization of the general projection functions and other operations for extracting messages send or received from traces:

- `receptions(t)` is the sub-trace of reception events of `t`;
- `transmissions(t)` is the sub-trace of transmission events of `t`;
- `rec_msg(t)` is the set of messages occurring in a reception event of `t`;
- `trans_msg(t)` is the set of messages in a transmission event of `t`.

The definitions are given in Fig. 15.

In a similar way, we define functions which convert sets of events into set of messages and conversely, and we specify predicates related to rank functions. Rank functions are of type `[M -> int]` where the parameter `M` is the message type. We use various predicates such as `pos_trans(rho, tr)` or `pos_rec(rho, tr)` where `rho` is a rank function and `tr` is of type `trace[event]`. These predicates hold if all the messages contained in a transmission event of `tr` or a reception event of `tr` have positive rank.

4.2 Rank properties

Using the above type of events and the associated operations and predicates, we specify two trace constraints as used in lemma 1 and theorem 1 (section 2.3). The first property is the trace constraint for the enemy: if the enemy only receives

messages of positive ranks, then it can only transmit messages of positive rank. This property is defined as follows:

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

This corresponds to the condition

$$M(tr\lceil trans) \subseteq \rho^+ \Rightarrow M(tr\lceil rec) \subseteq \rho^+$$

of lemma 1. The symmetric property is denoted by `RankUser(rho)`:

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

and is equivalent to the property called `maintains` ρ in theorem 1.

Condition (c4) of the main theorem requires us to prove that processes satisfy the trace property `maintains` ρ. In PVS, this corresponds to proving properties of the form

```
P |> RankUser(rho)
```

where `rho` is a rank function and `P` is a process expression. Such proofs can be intractable without an adequate set of CSP proof rules. In PVS, such rules are given by the collection of lemmas shown in Fig. 16. Some of these are simply instances of the general rules of satisfaction for the choice operators or direct application of the general induction rules. The others are proved from the semantic definition of the prefix and parallel composition operators.

All the lemmas of Fig. 16 can be used as rewrite rules in PVS proofs. The verification that `RankUser` is satisfied by `P` can then be performed by applying systematically the appropriate lemma according to the main operator of `P`. For example, consider the following specifications:

```
L: process[event] = (trans(i1, j1, m1) >> Stop[event])
                  \/ (rec(i2, j2, m2) >> Stop[event])

rho: VAR [M -> int].
```

We can prove the proposition below by systematically using the rules of Fig. 16.

```
test_rank: PROPOSITION rho(m1) > 0 IMPLIES L |> RankUser(rho)
```

After skolemization and expansion of `L`, we obtain the following goal:

```
[-1]  rho!1(m1) > 0
  |-------
{1}   (((trans(i1, j1, m1) >> Stop[event])
         \/ (rec(i2, j2, m2) >> Stop[event])) |> RankUser(rho!1))
```

and the command `(rewrite "rank_user_choice")` gives

```
[-1]    rho!1(m1) > 0
  |-------
{1}    (trans(i1, j1, m1) >> Stop[event]) |> RankUser(rho!1)
          AND (rec(i2, j2, m2) >> Stop[event]) |> RankUser(rho!1).
```

```
rank_user_stop:  LEMMA Stop[event] |> RankUser(rho)

rank_user_choice:  LEMMA
  (P1 \/ P2) |> RankUser(rho) IFF
      P1 |> RankUser(rho) AND P2 |> RankUser(rho)

rank_user_choice2:  LEMMA
  Choice(SP) |> RankUser(rho) IFF
      (FORALL (P: (SP)): P |> RankUser(rho))

rank_user_choice3:  LEMMA
  Choice(P) |> RankUser(rho) IFF
      FORALL i:  P(i) |> RankUser(rho)

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))

rank_user_par:  LEMMA
  P1 |> RankUser(rho) AND P2 |> RankUser(rho) IMPLIES
      Par(A)(P1, P2) |> RankUser(rho)

rank_user_par2:  LEMMA
  (FORALL i:  P(i) |> RankUser(rho)) IMPLIES
      Par(A)(P) |> RankUser(rho)

rank_user_free_par:  LEMMA
  P1 // P2 |> RankUser(rho) IFF
      P1 |> RankUser(rho) AND P2 |> RankUser(rho)

rank_user_free_par2:  LEMMA
  Interleave(P) |> RankUser(rho) IFF
      FORALL i:  P(i) |> RankUser(rho)

rank_user_fix1:  LEMMA
  (FORALL P: P |> RankUser(rho) IMPLIES F(P) |> RankUser(rho))
      IMPLIES mu(F) |> RankUser(rho)

rank_user_fix2:  LEMMA
  (FORALL P: (FORALL i:  P(i) |> RankUser(rho)) =>
              (FORALL i:  F(P)(i) |> RankUser(rho)))
  IMPLIES
      FORALL i:  mu(F)(i) |> RankUser(rho)
```

Figure 16: Proof Rules for RankUser

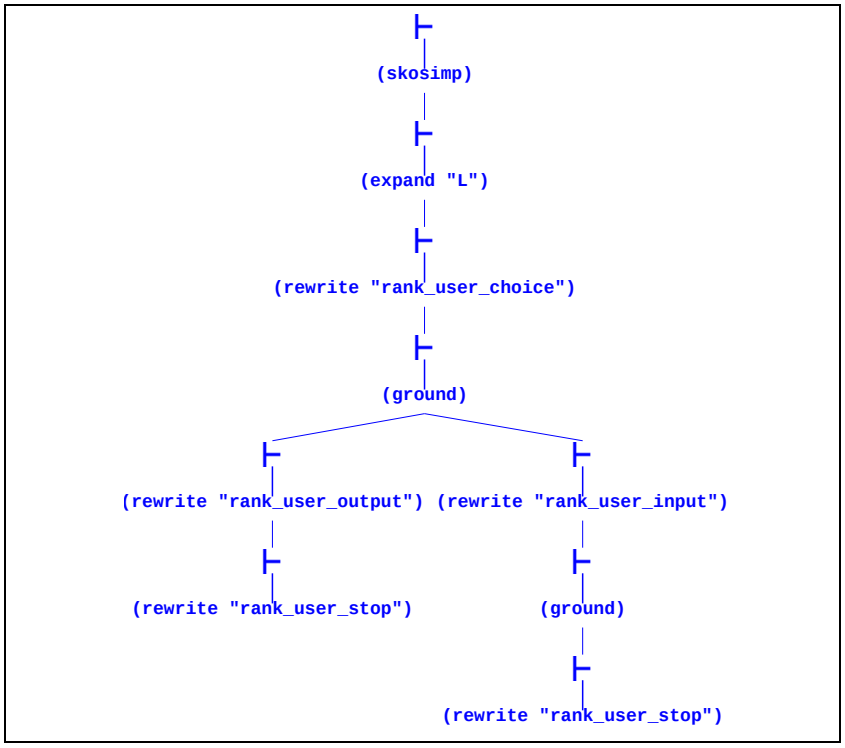

Figure 17: Proof tree for `test_rank`.

Propositional simplification splits this sequent in two sub-goals. The first is proved by applying successively `rank_user_output` and `rank_user_stop` and the second using `rank_user_input` then `rank_user_stop`. The corresponding proof tree is given in Fig. 17.

There are shorter ways of doing the above proof by installing the necessary lemmas as automatic rewrite rules. Unfortunately, this technique does not always work. Lemmas such as `rank_user_par` or `rank_user_fix1` are conditional rewrite rules. For example, `rank_user_par` is equivalent to

```
P1 |> RankUser(rho) AND P2 |> RankUser(rho) IMPLIES
      (Par(A)(P1, P2) |> RankUser(rho)) = true.
```

Automatic rewriting only applies this rule if the two premisses of the implication are reduced to true by the decision procedures or by further rewriting. In most situations, rewriting the premisses will introduce conditions of the form `rho(m)>0` via `rank_user_output` and `rank_user_input`. Unless these conditions are trivially true, the whole chain of rewritings fails.

4.3 Enemy and users

With the CSP constructors all available in PVS, it is straightforward to specify the enemy. The definition depends on two type parameters `Identity` and `Message` associated with the data type `event`. In addition, the definition is parameterised by a function `|-` which corresponds to the 'generates' relation of section 2.2. Since $ENEMY$ is recursive, the PVS construction relies on the least fixed point operator and uses an auxiliary mapping `F` as shown in Fig. 18. In this particular case, the

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

  IMPORTING event[Identity, Message], csp_traces

  i, j:  VAR Identity
  m:  VAR Message
  X: VAR [set[Message] -> process[event]]
  S: VAR set[Message]

  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
```

Figure 18: Definition of `enemy`

TCC generated to ensure that `F` is monotonic can be discharged automatically, without the help of the rules of Fig. 12. The proof is

```
(GRIND :exclude "add").
```

In order to describe the network and prove the main theorems, we need a general representation of the user processes. This is not completely trivial because we have to formalize assumptions about the interface of users. For this purpose, we define sets of events `LocalEvents(i)` which contain all the events $USER(i)$ may generate:

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

An essential property is that the communication channels between the enemy and the users are private. This is formalized as follows:

```
disjoint_events: LEMMA
   i /= j IMPLIES disjoint?(LocalEvents(i), LocalEvents(j)).
```

Now we can specify a PVS dependent type which characterises user processes:

```
P : VAR process[event]

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

Any `user` of this 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)`.

Using this type definition and the disjointness property above, it is easy to prove the following important result:

```
user: VAR user_process

local_traces: LEMMA
   Interleave(user)(tr) IMPLIES user(i)(proj(tr, LocalEvents(i))).
```

This is a direct application of lemma `interleave_disjoint`; if `tr` is a trace of the composition of all user processes then the projection of `tr` on `LocalEvents(i)` is a trace of `user(i)`.

Finally, we define a function `network` with two parameters, an arbitrary process `baddy` which represents the communication medium and a family of processes `P(i)` which represents the users. The whole network is a parallel composition of the processes `baddy` and `P(i)`:

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

This is similar to the definition of NET given previously: the enemy on one side and the composition of the users on the other synchronise on all reception and transmission events.

4.4 Key theorems

The specification and proof of theorems equivalent to those of section 2.3 is an important component of our PVS developments. All are defined in a theory `network` parameterised with the types of messages and user identities together with a message generation relation. The theory makes the following assumption about this relation:

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

This assumes that every message which can be generated from a set `A` can also be generated from any superset of `A`. This is the only property required of `|-` for all the results to hold. Within the theory, `monotonic_gen` can be used exactly like an axiom, but when one imports a particular instance of `network`, PVS generates a TCC to check that the assumption holds.

A first lemma follows from the definition of `enemy` and the above assumption: if `tr` is any trace of the process `enemy(S)` then every message in a reception event of `tr` is generated from the union of `S` and the set `trans_msg(tr)`. This property is specified by the predicate `Prop(S)` below and theorem `enemy_prop` gives the result.

```
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).
```

The proof is based on the induction rules for parametric processes. As a corollary, one obtains

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

This gives the first half of theorem 1: the two conditions (c1) and (c2) imply that the enemy satisfies the expected trace constraint.

The other results of section 2.3 involve the restriction of processes to events which do not belong to a given set R. This operation corresponds to parallel compositions of the form $P\,|[R]|\,Stop$. In PVS, such expressions are not easy to manipulate and it is convenient to introduce a specific operator as follows:

```
#(P, B): process[T] = Par(B)(P, Stop).
```

Hence, the restriction $P\,|[R]|\,Stop$ is denoted by `P # R` in PVS. Using this notation we get the following property of the network:

```
traces_of_network: LEMMA
   (network(baddy, user) # R)(tr) IMPLIES baddy(tr)
       AND FORALL i: (user(i) # R)(proj(tr,LocalEvents(i))).
```

This lemma states that, when events of `R` are prevented, any trace `tr` of the network is a trace of the enemy and that projection on the interface of user i gives a trace of `user(i) # R`. This is proved using previous properties such as `local_traces` and `par_full` and other lemmas about projection and restriction.

Using the preceding lemma, we can derive the fundamental property below:

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

If the enemy satisfies `RankEnemy(rho)` and the users satisfy `RankUser(rho)` when events of `R` are prevented then only message of positive rank can circulate in the network. Although this result is reasonably clear, the PVS proof is quite involved. It is based on an essentially brute force approach. The process definitions are expanded to sets of traces and the satisfaction relation to an inclusion of sets. The result is then obtained by induction on traces. Since the sets are prefix-closed we do not use the standard list induction axiom but the reverse form – `add_induction` – shown in Fig. 6.

Using `main_result` and `rank_property` we can finally prove 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).
```

This gives the same four conditions as theorem 1; `auth(T, R)` is the trace predicate corresponding to the authentication property "events of T cannot be observed unless events of R have been observed before".

In applications, we are often interested in protocols where only two users are active. The associated family of processes is defined below

```
protocol(a, b, user_a, user_b)(i): process[event] =
  IF i=a THEN user_a ELSIF i=b THEN user_b ELSE Stop ENDIF.
```

For such protocols, the previous theorem specialises to

```
authentication_by_rank2: THEOREM
      positive(rho, INIT)
  AND (FORALL S, m:
         positive(rho, S) AND (S |- m) IMPLIES rho(m) > 0)
  AND non_positive(rho, T)
  AND subset?(sigma(user_a), LocalEvent(a))
  AND user_a # R |> RankUser(rho)
  AND subset?(sigma(user_b), LocalEvent(b))
  AND user_b # R |> RankUser(rho)
IMPLIES
  network(enemy(INIT), protocol(a, b, user_a, user_b))
      |> auth(T, R).
```

This time, extra conditions are introduced. Previously, the interface properties of users were hidden in the type of `user`; in the new form, the interface requirements for the two processes `user_a` and `user_b` are explicit and this ensures that the expression `protocol(a, b, user_a, user_b)` is of type `user_process`.

Hence the translation to PVS requires additional verifications which are not explicit in the original theorem. The fact that user processes satisfy the interface constraints is usually taken for granted in manual verifications and can be checked by a simple inspection. In PVS such seemingly obvious properties have to be verified mechanically as anything else.

4.5 Specialised proof rules

The latter two theorems are the main tools used in practice when verifying authentication properties. This requires verification of possibly complex properties of processes, namely conditions of the form

```
P # R |> RankUser(rho).
```

The lemmas about `RankUser` shown in Fig. 16 can serve as rewrite rules in proofs of such properties but the rules do not apply directly. We need first to transform `P # R` into a CSP expression whose main operator is not a restriction. For this purpose, we can define lemmas for computing restrictions. These are organised in the same way as the monotonicity rules (Fig. 13) or the `RankUser` rules: one rule for each variant of all CSP operators and one induction rule for the two forms of fixed point constructors. A few examples are given in Fig. 19.

In a very similar way, we also use lemmas for reasoning about interface constraints. They systematically decompose properties of the form

```
subset?(sigma(P), E)
```

according to the main operator of `P`; for example, the rules for choice and prefix are

```
interface_choice: LEMMA
  subset?(sigma(P1 \/ P2, E)) IFF
     subset?(sigma(P1), E) AND subset?(sigma(P2), E)

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

In the case of the two main theorems, the interface constraints are proved by repeated application of these rules and of the two following lemmas:

```
restriction_stop:  LEMMA Stop # B = Stop

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

restriction_choice:  LEMMA
  (P1 \/ P2) # B = (P1 # B \/ P2 # B)

...

restriction_choice3:  LEMMA
  Choice(P) # R = Choice!  i:  P(i) # R

...

restriction_fix:  LEMMA
  Stop |> E AND (FORALL P: P # B |> E IMPLIES F(P) # B |> E)
    IMPLIES mu(F) # B |> E.
```

Figure 19: Rules for Restriction

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

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

A last set of proof rules are also available for direct reasoning about authentication properties, for example:

```
authentication_pref1: LEMMA
   B(a) IMPLIES (a >> P) |> auth(A, B)

authentication_pref2: LEMMA
   P |> auth(A, B) AND not A(a)
       IMPLIES (a >> P) |> auth(A, B).
```

5 The Needham-Schroeder Public Key Protocol

5.1 The Protocol

The whole PVS formalization presented previously has been applied to variants of the Needham-Schroeder public key protocol. The full protocol is described in [20] and involves two principals A and B and a server S for distributing public keys. A scaled-down version assumes that A and B already know each other's key and consists of three message exchanges:

$$\begin{array}{lll} 1 & A \rightarrow B: & \{N_a, A\}_{K_b} \\ 2 & B \rightarrow A: & \{N_a, N_b\}_{K_a} \\ 3 & A \rightarrow B: & \{N_b\}_{K_b}. \end{array}$$

In this description, N_a and N_b are nonces, that is, fresh unguessable values generated by A and B for a specific run of the protocol. An expression of the form $\{m\}_{K_x}$

denotes a message m encrypted with X's public key K_x. The corresponding private key is denoted by K_x^{-1} and we have the identity:

$$\{\{m\}_{K_x}\}_{K_x^{-1}} \quad = \quad \{\{m\}_{K_x^{-1}}\}_{K_x} \quad = \quad m.$$

Recently, Lowe has discovered a potential weakness in the above protocol [16, 17]. If A initiates a legitimate run with an agent C then C can initiate another run with B and make B act as if the other partner was A:

$$\begin{array}{lll} 1 & A \rightarrow C: & \{N_a, A\}_{K_c} \\ 1' & C(A) \rightarrow B: & \{N_a, A\}_{K_b} \\ 2' & B \rightarrow C(A): & \{N_a, N_b\}_{K_a} \\ 2 & C \rightarrow A: & \{N_a, N_b\}_{K_a} \\ 3 & A \rightarrow C: & \{N_b\}_{K_c} \\ 3' & C(A) \rightarrow B: & \{N_b\}_{K_b} \end{array}$$

C can rightly convince B that A is present but without A's knowledge or consent. Whether this can qualify as an attack depends on the goal for which the protocol is used. This is certainly an oddity which can be easily corrected as Lowe [17] indicates:

$$\begin{array}{lll} 1 & A \rightarrow B: & \{N_a, A\}_{K_b} \\ 2 & B \rightarrow A: & \{N_a, N_b, B\}_{K_a} \\ 3 & A \rightarrow B: & \{N_b\}_{K_b}. \end{array}$$

All the examples presented in [28] are based on these two variants of Needham-Schroeder, most of them on Lowe's improved version. All the verifications have been mechanically checked with PVS. We present a few examples in the next sections, starting by a simple authentication property of the original protocol.

5.2 Messages and Encryption

The first step in the analysis is to formalize public key encryption. We choose to represent identities and nonces by natural numbers but any infinite type would do. Plaintext is represented by a non-intepreted type `Text`.

This is an almost arbitrary decision but it is convenient for some properties to have an infinite number of nonces available. The message space is defined by an abstract datatype shown in Fig. 20.

The type `message` includes constructors for the five elementary kinds of messages: plaintext, nonces, user identities, and public and secret keys. Two more constructors denote concatenation and encryption. The declaration is slightly different from the examples of data type presented so far. In addition to the usual constructors, accessors and recognisers, two subtypes `key` and `nonkey` are defined. These represent two categories of messages and are similar to extra recognisers[3]. The subtype `key` corresponds to encryption keys and is the union of `(public?)` and `(secret?)`. This allows us to restrict the domain of the constructor `code` so that encryption applies to a pair key, message. The axiomatic specification of `message` includes the following definitions and declarations:

```
key(x): boolean = public?(x) OR secret?(x)
```

[3] We are only interested in `key` but PVS requires that all constructor be given a subtype.

```
messages :  THEORY
  BEGIN
  Identity :  NONEMPTY_TYPE = nat

  Text :  NONEMPTY_TYPE

  Nonce :  NONEMPTY_TYPE = nat

  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

  ...
```

Figure 20: Messages

```
key: TYPE = { x: message | public?(x) OR secret?(x) }

code: [[key, message] -> (code?)]

x_code: [(code?) -> key].
```

The identifier `key` is overloaded. It denotes both a predicate on `messages` and a subtype of `messages`.

Using a data type immediately gives us important properties. The different types of messages are disjoint; a nonce cannot be confused with a key or with a user identity. The extensionality axioms ensure that keys are unique: the public or secret keys of different users are distinct. As a whole, the data type representation is reasonably convenient but it does not allow us to directly specify that encryption with `public(i)` and encryption with `secret(i)` are inverse operations. We cannot assume

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

this would contradict the data type axioms. Instead, we use an encryption function which applies the previous rule:

```
crypto(k, m) : message =
   CASES m OF
     code(x, y) :
        CASES k OF
          public(i) :
             IF x = secret(i) THEN y ELSE code(k, m) ENDIF,
          secret(i) :
             IF x = public(i) THEN y ELSE code(k, m) ENDIF
```

```
      ENDCASES
    ELSE code(k, m)
  ENDCASES
```

This function is to be used everywhere in place of the constructor `code` and performs a normalization of messages.

There are three ways of generating new messages from old ones: encryption, concatenation, and extraction of a component from a concatenation. The message generation relation is the smallest relation $\vdash$ closed under the following rules:

$$\begin{aligned} m \in S &\Rightarrow S \vdash m \\ S \vdash m_1.m_2 &\Rightarrow S \vdash m_1 \wedge S \vdash m_2 \\ S \vdash m_1 \wedge S \vdash m_2 &\Rightarrow S \vdash m_1.m_2 \\ S \vdash k \wedge S \vdash m &\Rightarrow S \vdash \{m\}_k, \end{aligned}$$

where $m_1.m_2$ denotes concatenation. PVS supports such inductive definitions. We could define `|-` directly as above but it is more convenient to define first the set `Gen(S)` of messages which can be generated from `S`:

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

PVS interprets `Gen(S)` as the smallest set closed under the five clauses and generates two induction axioms accordingly. These two axioms are shown in Fig. 21. They are quite heavy to manipulate and a more comfortable induction theorem can be derived easily:

```
gen_msg_induction: PROPOSITION
         subset?(S, A)
    AND (FORALL m1, m2: A(conc(m1, m2)) <=> A(m1) AND A(m2))
    AND (FORALL m, k: A(m) AND A(k) => A(crypto(k, m)))
 IMPLIES subset?(Gen(S), A)
```

We can then define the message generation relation and use the preceding proposition to prove the required monotonicity property:

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

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

5.3 A Simple Verification

5.3.1 Users

We consider the first version of Needham-Schroeder without Lowe's fix. A single run of the protocol is executed by two principals A and B. We show that this variant ensures the following property: reception by B of the message $\{N_b\}_{K_b}$ gives assurance that A responded to B's nonce challenge, but does not necessarily

```
Gen_weak_induction:  AXIOM
  (FORALL (P: [set[message] -> [message -> bool]], S):
    (FORALL (m):
         (S(m)
      OR (EXISTS m1, m2:
            P(S)(m1) AND P(S)(m2) AND m = conc(m1, m2))
      OR (EXISTS m1:  P(S)(conc(m1, m)))
      OR (EXISTS m2:  P(S)(conc(m, m2)))
      OR (EXISTS m1, k:
            P(S)(m1) AND P(S)(k) AND m = crypto(k, m1)) )
    IMPLIES P(S)(m))
  IMPLIES
    (FORALL (m):  Gen(S)(m) IMPLIES P(S)(m)));

Gen_induction:  AXIOM
  (FORALL (P: [set[message] -> [message -> bool]], S):
    (FORALL (m):
         (S(m)
      OR (EXISTS m1, m2:  (Gen(S)(m1) AND P(S)(m1))
            AND (Gen(S)(m2) AND P(S)(m2)) AND m = conc(m1, m2))
      OR (EXISTS m1:
            Gen(S)(conc(m1, m)) AND P(S)(conc(m1, m)))
      OR (EXISTS m2:
            Gen(S)(conc(m, m2)) AND P(S)(conc(m, m2)))
      OR (EXISTS m1, k:  (Gen(S)(m1) AND P(S)(m1))
            AND (Gen(S)(k) AND P(S)(k)) AND m = crypto(k, m1)))
    IMPLIES P(S)(m))
  IMPLIES
    (FORALL (m):  Gen(S)(m) IMPLIES P(S)(m)));
```

Figure 21: Induction axioms for `Gen`

associates N_b with B. This is property (7) of [28]. The stronger property where A knows that the nonce was produced by B does not hold; this is the weakness identified by Lowe.

We start by declaring the identities of the two agents and the two nonces N_a and N_b. The variables `i` and `v` are of type `Identity` and `Nonce`, respectively.

```
a: Identity
b: { i | i /= a }

na: Nonce
nb: { v | v /= na }
```

These declarations assure us that the two identities and the two nonces are different. Since `Identity` and `Nonce` are equal to `nat` the two subtypes are not empty and the declarations are sound. We then introduce several abbreviations:

```
Ia: (user?) = user(a)
Ib: (user?) = user(b)

Na: (nonce?) = nonce(na)
Nb: (nonce?) = nonce(nb)

pub(i, x) : message = crypto(public(i), x)
sec(i, x) : message = crypto(secret(i), x).
```

The four constants are atomic messages containing the identities of the two agents and the two nonces. The two functions abbreviate encryption with public and secret keys.

The enemy has access to all public information, that is, the identity of all users, all the public keys, and all plaintext. We also assume that the enemy knows the secret key of every user except A and B and can generate any nonce except N_b:

```
INIT_nonce: set[message] =
   { m | EXISTS v: v /= nb AND m = nonce(v) }

INIT_secret: set[message] =
   { m | EXISTS i: i /= a AND i /=b AND m = secret(i) }

INIT: set[message] =
   { m | user?(m) OR text?(m) OR public?(m) OR
            INIT_nonce(m) OR INIT_secret(m) }.
```

The property considered is `auth(T3, R3)` where `T3` and `R3` are the following sets of events:

```
T3: set[event] = { e | e = rec(b, a, pub(b, Nb)) }

R3: set[event] = { e | EXISTS i: e = trans(a, i, pub(i, Nb)) }.
```

Informally, the authentication property means that, when B receives an answer to his nonce challenge, B can be sure that the answer originated from A and then that A is present. This answer was part of a legitimate run originated by A but as Lowe's attack demonstrates, this run may be with another principal than B.

B is running half the protocol in position of responder and acts as if A was the originator. On the other side, A is running the protocol in position of originator with an arbitrary agent. The behaviours of A and B can then be described by the following processes.

```
userA: process[event] =
  Choice! i, xn:
    ( trans(a, i, pub(i, conc(Na, Ia))) >>
    (    rec(a, i, pub(a, conc(Na, xn))) >>
    (       trans(a, i, pub(i, xn)) >> Stop[event] )))

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

The variable `xn` is of type `(nonce?)`; we assume that A can check and reject ill-formed messages in the second step of the protocol. The variable `y` is an arbitrary message; B accepts anything of the form $\{m, A\}_{K_b}$ in the first step.

It is easy to check that `userA` and `userB` are valid user processes; they both satisfy the required interface constraints:

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

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

The proofs of such lemmas is straight-forward; we simply apply the specialised rewrite rules shown previously. The proof of `interface_userA` is shown in Fig. 22. First, we install four automatic rewrite rules and then we expand the definition of `userA`. In the last command, we apply manually the rule `interface_choice3`; PVS finds a match and the resulting formula is reduced to `true` by automatic rewriting.

5.3.2 Rank Function

In order to verify the authentication property, we use the following rank function, given in [28]:

```
rank_code(k, n) : int =
   CASES k OF
      public(z) : IF z = a THEN n + 1 ELSE n ENDIF,
      secret(z) : IF z = a THEN n - 1 ELSE n ENDIF
   ENDCASES

rho(m) : RECURSIVE int =
   CASES m OF
      text(z)      : 1,
      nonce(z)     : IF z = nb THEN 0 ELSE 1 ENDIF,
      user(z)      : 1,
      public(z)    : 1,
      secret(z)    : IF z = a OR z = b THEN 0 ELSE 1 ENDIF,
      conc(z1, z2) : min(rho(z1), rho(z2)),
      code(k, z)   : rank_code(k, rho(z))
   ENDCASES
 MEASURE size(m).
```

We immediately get two of the necessary conditions for the main theorem to apply:

```
rank_init : LEMMA  positive(rho, INIT)

nonpositive_rank3 : LEMMA  non_positive(rho, T3).
```

```
interface_userA :

  |-------
{1} subset?(sigma(userA), LocalEvents(a))

Rule?  (AUTO-REWRITE "local_transmission" "local_reception"
               "interface_pref[event]" "interface_stop[event]")

Installing automatic rewrites from:
  local_transmission
  local_reception
  interface_pref[event]
  interface_stop[event]
this simplifies to:
interface_userA :

  |-------
[1] subset?(sigma(userA), LocalEvents(a))

Rule?  (EXPAND "userA")

Expanding the definition of userA,
this simplifies to:
interface_userA :

  |-------
{1} subset?(sigma(Choice!  i, xn:
                      (trans(a, i, pub(i, conc(Na, Ia))) >>
                        (rec(a, i, pub(a, conc(Na, xn))) >>
                          (trans(a, i, pub(i, xn)) >> Stop[event])))),
               LocalEvents(a))

Rule?  (REWRITE "interface_choice3")

Found matching substitution:
E gets LocalEvents(a),
P gets LAMBDA i, xn:  ....

Rewriting using interface_choice3,
Q.E.D.
```

Figure 22: Proof of `interface_userA`

These two lemmas are proved by a single command: `(GRIND)`. Roughly, this expands all the definitions, splits the resulting goal by using the propositional rules, and applies the decision procedures.

Another necessary condition is given by:

```
validity_rho : LEMMA
   FORALL S, m : positive(rho, S) AND (S |- m) IMPLIES rho(m) > 0.
```

This lemma relies on a property which follows easily from the induction theorem for messages:

```
rank_valid : PROPOSITION
     (FORALL m1, m2:
        rho(conc(m1, m2)) > 0 <=> rho(m1) > 0 AND rho(m2) > 0)
  AND (FORALL m, k:
        rho(m) > 0 AND rho(k) > 0 IMPLIES rho(crypto(k, m)) > 0)
IMPLIES
  (FORALL S, m: positive(rho, S) AND (S |- m) IMPLIES rho(m) > 0).
```

The proof of `validity_rho` starts by `(rewrite "rank_valid")`. This yields the two following subgoals:

```
validity_rho.1 :

  |-------
{1} (FORALL (m1: message), (m2: message):
      rho(conc(m1, m2)) > 0 <=> rho(m1) > 0 AND rho(m2) > 0)
[2] FORALL S, m: positive(rho, S) AND (S |- m) IMPLIES rho(m) > 0


validity_rho.2 :

  |-------
{1} (FORALL (m: message), (k: key):
       rho(m) > 0 AND rho(k) > 0 IMPLIES rho(crypto(k, m)) > 0)
[2] FORALL S, m: positive(rho, S) AND (S |- m) IMPLIES rho(m) > 0
```

and both are proved by `(delete 2)(grind)`.

5.3.3 Rank Preservation Properties

So far, all the proofs are routine; they mostly use automatic rewriting and decision procedures. Two more complex conditions remain to be checked before we can apply the main theorem:

```
rank_user_a3: LEMMA userA # R3 |> RankUser(rho)

rank_user_b3: LEMMA userB # R3 |> RankUser(rho).
```

Although the proofs are more involved than previously, these two lemmas can be verified in a fairly systematic way by applying the specialised CSP rules for restriction and for ranks (cf. Fig. 16 and 19).

The script of a first proof of `rank_user_a3` is shown in Fig. 23. The obvious first step expands the definition of `userA`. This yields the following sequent:

```
  |-------
{1} (Choice! i, xn:
```

```
("" 
 (EXPAND "userA")
 (REWRITE "restriction_choice3")
 (REWRITE "rank_user_choice3")
 (SKOLEM!)
 (NAME-REPLACE "i!2" "PROJ_1(i!1)")
 (NAME-REPLACE "nx!1" "PROJ_2(i!1)")
 (AUTO-REWRITE "rank_user_stop[Identity, message]"
               "rank_user_input[Identity, message]"
               "rank_user_output[Identity, message]"
               "restriction_stop[event]")
 (REWRITE "restriction_pref")
 (LIFT-IF)
 (ASSERT)
 (PROP)
 (("1" (DELETE 2) (GRIND))
  ("2"
   (REWRITE "restriction_pref")
   (LIFT-IF)
   (ASSERT)
   (PROP)
   (REWRITE "restriction_pref")
   (LIFT-IF)
   (ASSERT)
   (PROP)
   (DELETE 3 4)
   (GRIND)
   (REPLACE -1 + RL)
   (REPLACE-ETA "nx!1"))))
```

Figure 23: First proof of `rank_user_a3`.

```
          (trans(a, i, pub(i, conc(Na, Ia))) >>
              (rec(a, i, pub(a, conc(Na, xn))) >>
                  (trans(a, i, pub(i, xn)) >>
                      Stop[event]))) # R3) |> RankUser(rho).
```

At this point, we simplify the process expression using `restriction_choice3`. This moves the restriction operator inside the choice:

```
   |-------
{1} Choice! (i: [Identity, (nonce?)]):
       (trans(a, PROJ_1(i), pub(PROJ_1(i), conc(Na, Ia))) >>
          (rec(a, PROJ_1(i), pub(a, conc(Na, PROJ_2(i)))) >>
             (trans(a, PROJ_1(i), pub(PROJ_1(i), PROJ_2(i)))
                  >> Stop[event]))) # R3
         |> RankUser(rho).
```

PVS also replaces the pair of variables `i, xn` by a single one of type `[Identity, (nonce?)]` and introduces projections. We can now rewrite with `rank_user_choice3` and get:

```
   |-------
{1} FORALL (i: [Identity, (nonce?)]):
      (trans(a, PROJ_1(i), pub(PROJ_1(i), conc(Na, Ia))) >>
         (rec(a, PROJ_1(i), pub(a, conc(Na, PROJ_2(i)))) >>
            (trans(a, PROJ_1(i), pub(PROJ_1(i), PROJ_2(i)))
                >> Stop[event]))) # R3
          |> RankUser(rho).
```

We then introduce skolem constants and replace the projection expressions by fresh variables to improve readability:

```
   |-------
{1} (trans(a, i!2, pub(i!2, conc(Na, Ia))) >>
       (rec(a, i!2, pub(a, conc(Na, nx!1))) >>
           (trans(a, i!2, pub(i!2, nx!1)) >> Stop[event])))
        # R3 |> RankUser(rho).
```

Just like in the first sequent, we have to prove a property of the form `P # R3 |> RankUser(rho)` where `P` is a process expression. The proof can proceed by evaluating the restriction, apply the appropriate `rank_user` rule, etc. However, it is more efficient to install the rules which can be applied automatically by PVS:

```
    (AUTO-REWRITE "rank_user_stop[Identity, message]"
                  "rank_user_input[Identity, message]"
                  "rank_user_output[Identity, message]"
                  "restriction_stop[event]").
```

Now we can evaluate the restriction using `restriction_pref`:

```
   |-------
{1} IF R3(trans(a, i!2, pub(i!2, conc(Na, Ia))))
    THEN Stop
    ELSE trans(a, i!2, pub(i!2, conc(Na, Ia))) >>
           ((rec(a, i!2, pub(a, conc(Na, nx!1))) >>
             (trans(a, i!2, pub(i!2, nx!1)) >> Stop[event])) # R3)
    ENDIF |> RankUser(rho).
```

The next step, `(lift-if)`, moves the satisfaction constraints in the two branches of the conditional:

```
    |-------
  {1} IF R3(trans(a, i!2, pub(i!2, conc(Na, Ia))))
      THEN Stop |> RankUser(rho).
      ELSE trans(a, i!2, pub(i!2, conc(Na, Ia))) >>
             ((rec(a, i!2, pub(a, conc(Na, nx!1))) >>
               (trans(a, i!2, pub(i!2, nx!1)) >> Stop[event])) # R3)
                   |> RankUser(rho)
      ENDIF.
```

The `(assert)` commands invokes the decision procedures and PVS simplifies the formula by automatic rewriting. After a case split we obtain two subgoals. The first,

```
    |-------
  {1} rho(pub(i!2, conc(Na, Ia))) > 0
  {2} R3(trans(a, i!2, pub(i!2, conc(Na, Ia)))),
```

is solved by `(delete 2)(grind)`. The second contains again a formula of the form `P # R |> RankUser(rho)`:

```
    |-------
  {1} rec(a, i!2, pub(a, conc(Na, nx! >>
        (trans(a, i!2, pub(i!2, nx!1)) >> Stop[event])
           # R3 |> RankUser(rho)
  {2} R3(trans(a, i!2, pub(i!2, conc(Na, Ia)))).
```

The proof proceeds by repeating twice the same four steps as previously: rewriting with `restriction_pref`, transformation of the conditional, automatic rewriting, and case split. We end up with

```
 [-1] rho(pub(a, conc(Na, nx!1))) > 0
    |-------
  {1} R3(trans(a, i!2, pub(i!2, nx!1)))
  {2} rho(pub(i!2, nx!1)) > 0
  [3] R3(rec(a, i!2, pub(a, conc(Na, nx!1))))
  [4] R3(trans(a, i!2, pub(i!2, conc(Na, Ia))))
```

where all process expressions have been eliminated. Formulas `[3]` and `[4]` are clearly false and can be deleted, then we apply `(grind)` which yields:

```
 {-1} x_nonce(nx!1) = nb
    |-------
  {1} trans(a, i!2, code(public(i!2), nx!1))
        = trans(a, i!2, code(public(i!2), nonce(nb)))
  {2} i!2 = a.
```

We finish the proof by applying the extensionality axioms for the data types `event` and `message`. PVS has specialised proof commands which find the right instance of these axioms.

A clear pattern emerges from this example. Starting from a goal which contains a formula of the form `P # R |> RankUser(rho)`, we apply a restriction rule and a few proof commands which are completely determined by the main operator of `P`. For example, if `P` is a prefix expression `a >> Q` we can systematically apply the four steps:

```
(rewrite "restriction_pref")(lift-if)(assert)(prop).
```

```
("")
 (INIT-CSP "Identity" "message")
 (EXPAND "userA")
 (CHOICE3)
 (NAME-REPLACE "i!2" "PROJ_1(i!1)")
 (NAME-REPLACE "nx!1" "PROJ_2(i!1)")
 (PREFIX)
 (("1" (DELETE 2) (GRIND))
  ("2"
   (PREFIX)
   (PREFIX)
   (DELETE 3 4)
   (GRIND)
   (REPLACE -1 + RL)
   (REPLACE-ETA "nx!1"))))
```

Figure 24: A shorter proof of `rank_user_a3`.

This produces one or two subgoals depending on whether `a` is a transmission or a reception event. In both cases, we obtain a sequent with the formula `Q # R |> RankUser(rho)`.

In order to exploit this regularity we define specialized PVS strategies which perform the right sequence of commands. For example, the strategies for prefix and parametric choice are defined by:

```
(defstep prefix ()
   (try (rewrite "restriction_pref")
        (then* (lift-if) (assert) (prop))
        (skip))
   "CSP rule for (a >> P) # R |> RankUser(rho)"
   "Applying prefix rule")

(defstep choice3 ()
   (try (rewrite "restriction_choice3")
        (then (rewrite "rank_user_choice3")(skolem!))
        (skip))
  "CSP rule for Choice! i : P(i) # R |> RankUser(rho)"
  "Applying choice rule").
```

We also use another proof strategy which installs the necessary automatic rewrites. With these new commands, the proof of `rank_user_a3` can be largely reduced as shown in Fig. 24.

The rank preservation property for `userB` is verified in the same way. The proof tree for `rank_user_b3` is shown in Fig. 25. Finally, as an immediate application of the preceding results and of theorem `authentication_by_rank2`, we obtain the expected property:

```
authentication_origin3 : PROPOSITION
    network(enemy(INIT), protocol(a, b, userA, userB))
       |> auth(T3, R3).
```

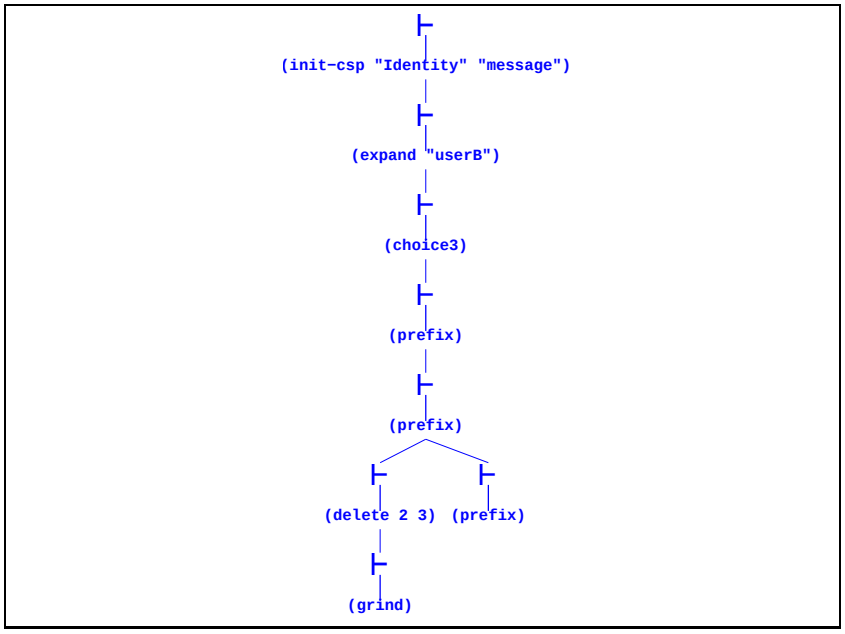

Figure 25: Proof tree for `rank_user_b3`.

5.4 Other Examples

The previous example illustrates the general PVS/CSP approach to authentication properties. For a particular variant of Needham-Schroeder and a given authentication properties, we construct a suitable rank function and prove the same lemmas as above. Most of the proofs are systematic; they use the CSP proof rules described in section 4.5, the automatic rewriting capabilities of PVS, and specialised proof commands for rank preservation.

The modelling allows us to consider variations in the behaviours of the two users. For example, we can assume that `userA` has a more permissive attitude than above and does not check as strictly the well-formedness of messages. This is modelled by the process below

```
userA2 : process[event] =
  Choice! i, x :
    ( trans(a, i, pub(i, conc(Na, Ia))) >>
    (    rec(a, i, pub(a, conc(Na, x))) >>
    (       trans(a, i, pub(i, x)) >> Stop[event] )))
```

which is similar to `userA` except for the type of `x`. This variable is of type `message` instead of `(nonce?)`: any message of the form $\{N_a, m\}_{K_a}$ is accepted by A in the second step of the protocol. The same authentication property as before still holds but the verification uses a more complex rank function.

We can also analyse responder authentication: we assume that A initiates a run with B and that B is willing to respond to a run initiated by any agent. The situation is symmetrical to previously. We can show that on reception of the second message, A can be sure that B is effectively the responder involved. This example is shown in Fig. 26.

It is as simple to describe Lowe's version of the protocol and to analyse the same authentication properties as before. Lowe's protocol also satisfies a stronger origin authentication property:

```
userA : process[event] =
  Choice!  xn :
    ( trans(a, b, pub(b, conc(Na, Ia))) >>
    (   rec(a, b, pub(a, conc(Na, xn))) >>
    (     trans(a, b, pub(b, xn)) >> Stop[event])))

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

...

T1 :  set[event] =
  { e | EXISTS xn :  e = rec(a, b, pub(a, conc(Na, xn))) }

R1 :  set[event] =
  { e | EXISTS xn :  e = trans(b, a, pub(a, conc(Na, xn))) }

...

responder_authentication :  PROPOSITION
  network(enemy(INIT), protocol(a, b, userA, userB))
          |> auth(T1, R1)
```

Figure 26: Responder authentication

```
T1 : set[event] = { e | e = rec(b, a, pub(b, Nb)) }

R1 : set[event] = { e | e = trans(a, b, pub(b, Nb)) }

authentication_origin : PROPOSITION
   network(enemy(INIT), protocol(a, b, userA, userB))
       |> auth(T1, R1)
```

Reception of the third message $\{N_b\}_{K_b}$ by B gives a guarantee that A responded to B's nonce challenge and as part of a run A initiated with B.

In all the previous examples, a single run of the protocol is considered and the role of the two users is fixed. It is possible to extend the analysis to more general situations where agents are able to perform multiple runs. Figure 27 shows the specifications of `userA` and `userB` in the case where users may engage in successive runs. The protocol considered is Lowe's variant.

Both `userA` and `userB` are parametric processes defined as least fixed points of two functions `Fa` and `Fb`. The process `userA(l)` where `l` is a natural number models the behaviour of A at the start of the lth run of the protocol. In such a run, A may act either as an originator or as a responder. The nonce used by A in either case is given by `Na(l)`. Similarly, `userB(l)` describes the behaviour of B at the start of the lth run of the protocol and `Nb(l)` is the corresponding nonce.

The analysis focuses on B's k_0th run where k_0 is a fixed but arbitrary number. We assume that in this run, B is in position of responder to a protocol it believes is initiated by A. In any other run, B can act either as an initiator or as a responder. The definition of `Na` and `Nb` ensures that the nonces `Na(l)` and `Nb(l)` where `l /= k0` are all different from `Nb(k0)`.

We want to show a property similar to `authentication_origin`: reception of the message $\{N_b(k_0)\}_{K_a}$ authenticates that A responded to B's nonce challenge. Formally this authentication property is defined by the two sets of events below:

```
T1 : set[event] = { e | e = rec(b, a, pub(b, Nb(k0))) }

R1 : set[event] = { e | e = trans(a, b, pub(b, Nb(k0))) }.
```

The rank function `rho` used for checking that `T1` authenticates `R1` is shown in Fig. 28. When multiple runs are considered, there is an increased risk of confusion between different runs and of replay attacks. The rank function is then substantially more complex than when a single run is analysed.

Despite the use of fixed points and parametric processes, the proofs are not much harder than before. Two TCCs are generated to check that `Fa` and `Fb` are monotonic functions. These two proof obligations are easily discharged using the monotonicity rules (cf. Fig. 29). All the other lemmas are the same as previously: the two processes satisfy the interface constraints and the rank function is valid. As usual the non-trivial part is to show that the users satisfy the rank preservation properties:

```
rank_user_a : LEMMA  userA(l) # R1 |> RankUser(rho)

rank_user_b : LEMMA  userB(l) # R1 |> RankUser(rho).
```

The proofs use the specialised proof commands as before and the induction rules. We progressively eliminate the CSP expressions from the sequents and the remaining goals are proved by brute force (i.e. `grind`) and with data type axioms. The proof tree for `rank_user_b` is shown in Fig. 30. Using the main theorem we then derive the expected property:

```
k0:  nat
nbk:  Nonce
f:  VAR [nat -> Nonce]

na:  { f | FORALL l:  f(l) /= nbk}
nb:  { f | FORALL l:  f(l) = nbk IFF l = k0 }

Na(l) :  (nonce?)  = nonce(na(l))
Nb(l) :  (nonce?)  = nonce(nb(l))

X : VAR [nat -> process[event]]

Fa(X)(l) :  process[event] =
    (Choice!  i, xn :
      ( trans(a, i, pub(i, conc(Na(l), Ia))) >>
      (   rec(a, i, pub(a, conc3(Na(l), xn, user(i)))) >>
      (     trans(a, i, pub(i, xn)) >> X(l + 1) ))))
  \/
    (Choice!  j, y :
      ( rec(a, j, pub(a, conc(y, user(j)))) >>
      (   trans(a, j, pub(j, conc3(y, Na(l), Ia))) >>
      (     rec(a, j, pub(a, Na(l))) >> X(l + 1) ))))

userA : [nat -> process[event]] = mu(Fa)

Fb(X)(l) :  process[event] =
  IF l = k0 THEN
    Choice!  y :
      ( rec(b, a, pub(b, conc(y, Ia))) >>
      (   trans(b, a, pub(a, conc3(y, Nb(k0), Ib))) >>
      (     rec(b, a, pub(b, Nb(k0))) >> X(k0 + 1) )))
  ELSE
    (Choice!  i, xn :
      ( trans(b, i, pub(i, conc(Nb(l), Ib))) >>
      (   rec(b, i, pub(b, conc3(Nb(l), xn, user(i)))) >>
      (     trans(b, i, pub(i, xn)) >> X(l + 1) ))))
  \/
    (Choice!  j, y :
      ( rec(b, j, pub(b, conc(y, user(j)))) >>
      (   trans(b, j, pub(j, conc3(y, Nb(l), Ib))) >>
      (     rec(b, j, pub(b, Nb(l))) >> X(l + 1) ))))
  ENDIF

userB : [nat -> process[event]] = mu(Fb)
```

Figure 27: Modelling repeated runs

```
critical(m) :  RECURSIVE bool =
  CASES m OF
    text(z)      :  FALSE,
    nonce(z)     :  z = nbk,
    user(z)      :  FALSE,
    public(z)    :  FALSE,
    secret(z)    :  FALSE,
    conc(z1, z2) :  critical(z1) OR critical(z2),
    code(k, z)   :  FALSE
  ENDCASES
 MEASURE size(m)

rank_pub_a(m, n) :  int =
  IF (conc?(m) AND critical(x_conc(m)) AND y_conc(m) = Ib)
  THEN 1
  ELSE n
  ENDIF

rank_pub_b(m, n) :  int =
  IF (conc?(m) AND conc?(x_conc(m)) AND
        critical(x_conc(x_conc(m))) AND y_conc(m) = Ia)
  THEN 1
  ELSE n
  ENDIF

rank_code(k, m, n) :  int =
  CASES k OF
    public(j) :
        IF j=a THEN rank_pub_a(m, n)
        ELSIF j=b THEN rank_pub_b(m, n)
        ELSE n
        ENDIF,
    secret(j) :  n
  ENDCASES

rho(m) :  RECURSIVE int =
  CASES m OF
    text(z)      :  1,
    nonce(z)     :  IF z = nbk THEN 0 ELSE 1 ENDIF,
    user(z)      :  1,
    public(z)    :  1,
    secret(z)    :  IF z = a OR z = b THEN 0 ELSE 1 ENDIF,
    conc(z1, z2) :  min(rho(z1), rho(z2)),
    code(k, z)   :  rank_code(k, z, rho(z))
  ENDCASES
 MEASURE size(m)
```

Figure 28: Rank function for repeated runs

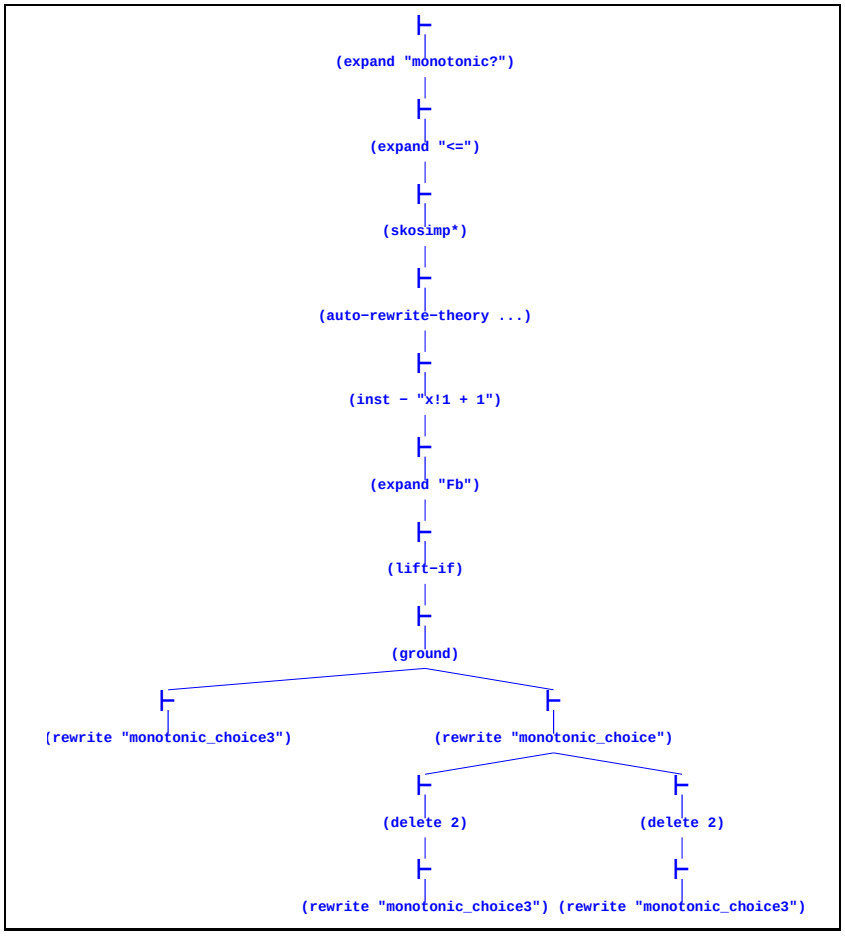

Figure 29: Proof tree for the monotonicity of `Fb`.

```
authentication_origin : PROPOSITION
   network(enemy(INIT),
           protocol(a, b, userA(0), userB(0))) |> auth(T1, R1).
```

Other examples have been analysed in a very similar way, including the case where agents can engage in concurrent runs. The specifications are very close to the case of successive runs and the proofs do not present any extra difficulty.

6 Discussion and Related Work

This report shows that PVS can provide efficient support for a non-trivial application of CSP. The usefulness of the mechanization is clear; PVS has found errors in our own manual proofs of authentication properties. A simple semantic embedding is sufficient for our purpose and PVS proved adequate for this development. Most of the mathematical machinery (i.e. lists and sets) is already present and the data type mechanism is particularly useful. More sophisticated embeddings could also be defined for example by including a type representing CSP expression. This would enable the derivation of meta-theoretical results such as the fact that systems of

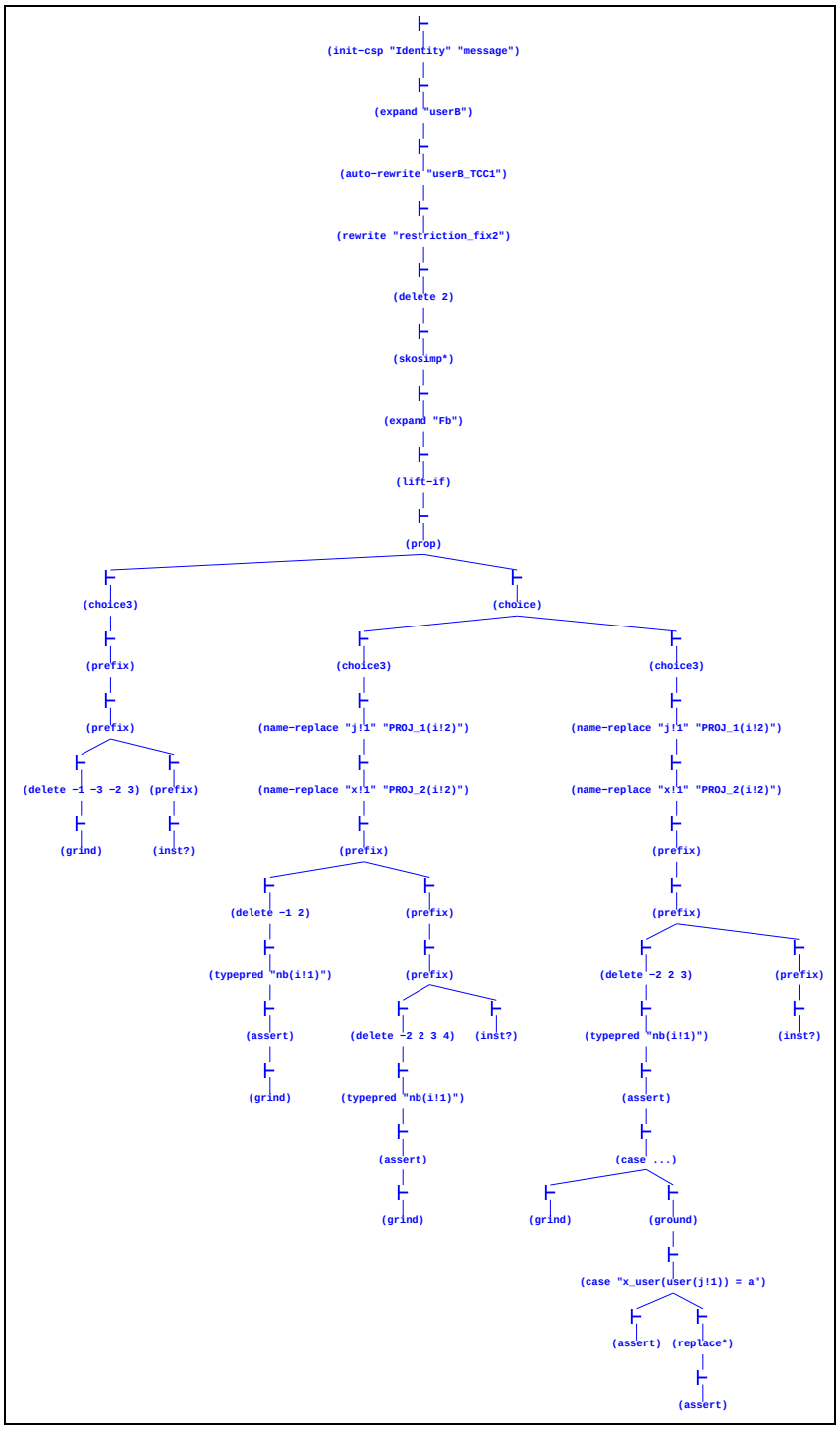

Figure 30: Proof tree for `rank_user_b`, successive runs.

mutually recursive process definitions always have a smallest solution. However, such extensions would require considerably more work for a relatively little gain.

Other formalisms than CSP have been embedded in PVS. As far as we are aware, the most extensive work in this area is the development of mechanical support for the duration calculus (DC) using PVS [30]. The approach used is largely similar to ours: the semantics of the duration calculus is directly defined in PVS. However, the embedding also includes a dedicated interface which makes the implementation largely transparent to users. The specifications can be written using a DC-like syntax and proofs are conducted using DC rules. This partly solves the main limitation of PVS for this kind of work: the rigidity of the specification language. Giving users the opportunity to define their own syntactic forms and symbols would be an appreciable improvement.

It is also clear that such embeddings are most effective if one does not try to strictly adhere to the original formalisms. Trade-off may be necessary between the purity of the embedding and usability. We made some adjustments to the CSP variant used by introducing unbounded parallel composition and by using a restriction operator in order to simplify the statement and proof of important theorems.

Other mechanizations of CSP with theorem provers can be found in the literature. Camilleri [6] presents a mechanization of CSP in HOL. The construction is similar to ours; it is a semantic embedding of the traces model but it also includes a HOL representation of the CSP syntax. There are minor differences in the CSP dialect considered and in the representation of events. Due to the use of parameters, the PVS formalization seems more general and flexible. Thayer [32] uses a more general approach for representing CSP with the IMPS prover. The development relies on abstract monoids which can cover many variants of CSP including timed CSP. Compared with the these works our main contribution is to have used the PVS embedding to a specific class of problems, namely the verification of authentication protocols.

Apart from existing mechanization of belief logics such as described in [3], tool support in this area, has mostly concentrated on analysing security protocols by searching for or attempting to construct attacks [16, 19, 14, 27]. The results of this kind of analysis is either the successful discovery of an attack, or else a bald statement that none can be found. The discovery of an attack provides useful insight into the flaws of a particular design, and hence can be useful in improving the design of the protocol. As with debugging, this process may be repeated until it reaches a point where no further flaws can be found, at which point analysis either ceases or must be continued by other means. While this approach has had some high-profile successes, an inability to find attacks does not in itself guarantee correctness of the protocol. For example, in the CSP based approach to searching for attacks, the model-checking tool FDR [10] is deployed. Model-checking of a system relies on the finitary nature of its state space, so its relationship with the infinite possibilities of attacks (such as arise from such aspects as the possibility of arbitrary depths of encryption and combining of messages), is far from trivial. Lowe [17] is considering a proof strategy based on the general form of a protocol run for establishing when absence of an attack on a finite state space is sufficient to establish that no attack exists in the infinite state space. Lazic and Roscoe [15] are developing a theory of data-independence which establishes conditions for results concerning a finite data space to extend to an arbitrary data space. Both of these approaches will give conditions for the results of model-checking on a restricted form of a protocol to extend to a fuller version.

Closer to our approach is that of Paulson [23, 24], who has investigated the application of the proof tool Isabelle/HOL to proving security properties of proto-

cols. He specifies security protocols in terms of traces of the system as a whole. The rounds of a protocol are translated into rules about how system traces may be augmented, and possible enemy activities also become rules. Once all of the rules have been identified, the aim is to prove that any system trace that can be generated using the rules must meet the required property; this is established by induction. He does not use an explicit rank function as we do, but he also aims to prove that particular messages can never occur in a trace, and this requires certain lemmas establishing that particular classes of message cannot occur. This is also a feature of the approach taken in [18], which applies language theory to establish that particular terms cannot be generated using given production rules. Mechanical assistance for proofs is invaluable, and Paulson has some useful results concerning reusability of proof strategies. It appears that a number of protocols have proofs of a similar shape, which allows efficient analysis of new protocols. At present the approach described in this paper still requires provision of the rank function in order for the proof to proceed, so analysis of fresh protocols will not be as automatic as Paulson reports for Isabelle/HOL. A distinguishing feature of this approach is to render each step in a protocol as an independent rule. This approach gives no control over when rules may be applied, in contrast to the CSP approach which essentially maintains the order of the protocol steps in the order in which proof rules are applied. participants. In principle, Paulson's approach allows more traces to be considered than are possible for the protocol. This pessimistic view may possibly result in a correct protocol being unprovable (incompleteness), though at least any verified protocol will be correct when its implementation places restrictions on the occurrence of messages: any verification will be sound.

Future directions

The modeling of the space of messages as an abstract data type means that messages that have been constructed in different ways are considered to be different. Algebraic properties of cryptographic mechanisms (such as commutativity of encryption, or distributivity of encryption over concatenation) have to be given explicitly as equations on the message space. The CSP approach of [28] remains valid in the presence of such equations, and the only additional proof obligation is to ensure that the rank function is well-defined: since rank functions are generally defined inductively, it is required to establish that the same message constructed two different ways should have the same rank function whichever construction is used.

Because of the way abstract data types are axiomatised, such equational reasoning is not immediate in PVS. Assuming that certain equations between messages are satisfied is in general not sound. In the case of the Needham-Schroeder protocol we have used instead a somewhat less elegant solution based on normalisation. This does not necessarily generalise to more complex algebraic relations between messages. There is no theoretical obstacle preventing us from introducing algebraic equations on the message types but this requires further developments. We envisage to construct enough PVS theories to allow us to manipulate quotient types. This would enable the representation of messages by equivalence classes of an abstract data type and would rejoin the original approach of [28].

Another important avenue to explore will be the extent to which construction of the rank function can be assisted by the attempt to provide a PVS proof. In effect, a proof can be provided with respect to particular constraints on the rank function which arise as instantiations of the side conditions of the rules that are used. If all of these constraints can be generated before any information about the rank function is required, then it may be possible to use them to identify a suitable rank function; the proof is completed once a function can be provided which meets

all the constraints.

We still have limited experience of the nature of the rank functions that are required for the proofs, though we are developing heuristics concerning appropriate functions for particular security mechanisms. We also need further experience concerning the structure of the proofs themselves. However, Paulson's conclusions concerning the reusability of his approach are encouraging, and it is important to apply the PVS theory that has been built up to more and larger examples to see whether our approach will offer the same benefits. The full seven message Needham-Schroeder protocol which includes communication with the server has now been analysed [4] and the verification discussed in this paper extended very cleanly to the full version.

In conclusion, we have presented viable mechanical support in PVS for the verification of security protocols with respect to authentication properties. It is an approach which complements and ideally follows the stage of debugging protocols by searching for attacks. There is still much to be done to develop this work to support mechanical verification in the presence of algebraic properties of the cryptographic mechanism, to develop proof tactics to capture common patterns in proofs, and to gain experience by investigating further protocols.

Acknowledgements

This work was partially funded by DRA/Malvern.

References

[1] M. Abadi and L. Lamport. Conjoining Specifications. Technical Report 118, Digital Equipment Corporation, System Research Center, December 1993.

[2] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth through Proof.* Academic Press, 1986.

[3] S. H. Brackin. Deciding Cryptographic Protocol Adequacy with HOL: The Implementation. In *TPHOLs'96*, pages 61–76. Springer-Verlag, LNCS 1125, August 1996.

[4] J. Bryans and S. Schneider. Mechanical Verification of the full Needham-Schroeder public key protocol. Technical report, Royal Holloway, University of London, 1997. in preparation.

[5] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. Technical Report 39, Digital Equipment Corporation, System Research Center, February 1989.

[6] A. J. Camilleri. Mechanizing CSP trace theory in higher order logic. *IEEE Transactions on Software Engineering*, 16(9):993–1004, September 1990.

[7] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. In *WIFT'95 Workshop on Industrial-Strength Formal Specification Techniques*, April 1995.

[8] D. Dolev and A. C. Yao. On the Security of Public Key Protocols. *IEEE Transactions on Information Theory*, 29(2), 1983.

[9] W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: An Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11:213–248, 1993.

[10] Formal Systems (Europe) Ltd. *Failure Divergence Refinement – User Manual and Tutorial*, 1993.

[11] D. Gollmann. What do we mean by Entity Authentication. In *IEEE Symposium on Security and Privacy*, 1996.

[12] M.J.C. Gordon and T.F. Melham. *Introduction to HOL. A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[13] C. A .R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[14] R. Kemmerer, C. Meadows, and Millen J. Three systems for cryptographic analysis. *Journal of Cryptology*, 7(2), 1994.

[15] R. Lazic and A. W. Roscoe. Using Logical Relations for Automated Verification of Data-independent CSP. *Electronic Notes in Theoretical Computer Science*, 5, 1997.

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

[17] G. Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. In *Proc. of TACAS'96*, pages 147–166. Springer-Verlag, LNCS 1055, 1996.

[18] C. Meadows. Applying formal methods to the analysis of a key management protocol. *Journal of Computer Security*, 1(1), 1992.

[19] J. Millen. The interrogator model. In *IEEE Symposium on Research in Security and Privacy*, 1995.

[20] R. M. Needham and M. D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[21] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[22] S. Owre, N. Shankar, and J. M. Rushby. *The PVS Specification Language.* Computer Science Lab., SRI International, April 1993.

[23] L. Paulson. Proving Properties of Security Protocols by Induction. Technical Report TR409, Computer Laboratory, University of Cambridge, December 1996.

[24] L. Paulson. Mechanised Proofs of Security Protocols: Needham-Schroeder with Public Keys. Technical Report TR413, Computer Laboratory, University of Cambridge, January 1997.

[25] Lawrence C. Paulson. A formulation of the simple theory of types (for Isabelle). In P. Martin-Löf and G. Mints, editors, *Proceedings of the International Conference on Computer Logic COLOG'88*, pages 246–274, Tallinn, Estonia, December 1988. Springer-Verlag LNCS 417.

[26] A. W. Roscoe. *Model-checking CSP.* Prentice-Hall, 1994.

[27] A. W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *Proc. of CSFW8.* IEEE Press, 1995.

[28] S. Schneider. Using CSP for protocol analysis: the Needham-Schroeder Public Key Protocol. Technical Report CSD-TR-96-14, Royal Holloway, University of London, November 1996.

[29] N. Shankar, S. Owre, and J. M. Rushby. *The PVS Proof Checker: A reference Manual.* Computer Science Lab., SRI International, March 1993.

[30] 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.

[31] P. Syverson and P. van Oorschot. On Unifying Some Cryptographic Protocol Logics. In *Proc. of the 1994 IEEE Symposium on Research in Security and Privacy*, pages 14–29, 1994.

[32] F. J. Thayer. An approach to process algebra using IMPS. Technical Report MP-94B193, The MITRE Corporation, 1994.