RAPPORT TECHNIQUE PROUVÉ

The PROUVÉ Protocol Specification Language

Auteur : Ralf Treinen

Date : March 23, 2005

Rapport PROUVÉ numéro : 3

Version : 1.3.4

Loria
CNRS UMR 7503,
Campus Scientifique - BP 239
54506 Vandoeuvre-lès-nancy cedex
www.loria.fr

Laboratoire Spécification Vérification
CNRS UMR 8643, ENS Cachan
61, avenue du président-Wilson
94235 Cachan Cedex, France
www.lsv.ens-cachan.fr

Laboratoire Verimag
CNRS UMR 5104,
Univ. Joseph Fourier, INPG
2 av. de Vignate,
38610 Gières, France
www-verimag.imag.fr

Cril Technology
9/11 rue Jeanne Braconnier
92360 Meudon La Foret Cedex, France
www.cril.fr

France Telecom
Div. Recherche et Développement
38, 40 rue du Général Leclerc
92794 Issy Moulineaux Cedex
www.rd.francetelecom.fr

Adresse : Laboratoire Spécification et Vérification,
CNRS UMR 8643, ENS Cachan
61, Avenue du Président Wilson, 94230 Cachan, France

Résumé : In this report we describe the PROUVÉ specification language for cryptographic protocols. A main feature of the language is that it separates the roles of a protocol, which are defined in a simple imperative programming language, from the scenarion which defines how instances of the roles are created.

Versions of this report are numbered according to the scheme *n.m.k* where *n*, *m*, and *k* are natural numbers. The version *n.m.k* of this report describes version *n.m* of the language.

The PROUVÉ Protocol Specification Language

Ralf Treinen

March 23, 2005

Contents

Chapter 1

Introduction

The purpose of the PROUVÉ protocol specification language is to give means to describe both protocols and the context in which they are used. It should allow to specify a signature of message constructors, and a type discipline with both default types and user-defined types. The language should allow to specify algebraic properties of message constructors by equational axioms.

The sub-language used to define protocols is oriented on existing imperative programming language. Other projects, like EVA, choose to use a more abstract, supposedly user-friendly language using the so-called Alice-Bob notation. This approach raises the need for a compiler from the user language to some kind of programming language that serves as input language for the verification tools. It is useful to understand how a correct protocol run is supposed to look like but does not describe in an unambiguous way the actions of the protocol participants. We felt that this approach hides from the difficulties of the Alice-Bob notation, and choose to have the user specify the protocol as seen by the protocol agents, rather then from the point of view of an outside observer of a correct protocol execution. As a consequence, this sub-language describes *roles* which are the programs executed by (legitimate) protocol participants.

We used constructs from imperative programming languages as building blocks for the sub-language describing roles. However, our language is deliberately not Turing-complete in that we excluded constructs that are either not useful for describing protocols or too complex to handle by our verification tools. The basic instructions are sending and receiving messages and pattern matching. Instructions are composed by serial composition and conditional branching. All variables have to be explicitly declared (either as local variables, or as parameters). Variables can be write-once (that is, logical variables) or mutable variables.

Roles are composed in so-called *Scenarios*. The sub-language for scenarios contains constructs for parallel and sequential composition, and non-deterministic choice of values. It has assignment statements, and allows to instantiate roles that have been previously defined.

Disclaimer: The design of the PROUVÉ language is a collective work of the members of the PROUVÉ project. The author of this report does not claim exclusive authorship of the PROUVÉ language.

Chapter 2

The Syntax of Specification Files

2.1 Notation

In the grammar, meta constuctions are in red, and key words are in underlined.

2.2 The Overall Structure

A specification consists of an optional signature which when given extends the default signature (see Chapter 3), an optional list of equational axioms, a list of role specifications, and a scenario.

$$\langle\text{specification}\rangle \quad \rightarrow \quad \langle\text{signature}\rangle\ ?\ \langle\text{axioms}\rangle\ ?\ \langle\text{role}\rangle\ *\ \langle\text{globals}\rangle\ ?\ \langle\text{scenario}\rangle$$

2.3 Signatures

2.3.1 Context-Free Syntax

A signature is a list of symbol declarations with their type. A single symbol declaration may define several symbols at once which are given by a comma-separated list.

$$\begin{array}{rcl}
\langle\text{signature}\rangle & \rightarrow & \underline{\text{signature}}\ (\ \langle\text{ne-identlist}\rangle\ :\ \langle\text{typeexp}\rangle\ ;\ )*\ \underline{\text{end}} \\
\langle\text{ne-identlist}\rangle & \rightarrow & (\ \langle\text{ident}\rangle\ ,\ )*\ \langle\text{ident}\rangle
\end{array}$$

Type expressions are composed out of basic types and type variables by arrow, tupling, and application of named type constructors. Type variables are distinguished from names of basic types by a leading apostrophe (').

$$\begin{array}{rcl}
\langle\text{typeexp}\rangle & \rightarrow & \langle\text{ident}\rangle \\
& | & {}'\langle\text{ident}\rangle \\
& | & (\ \langle\text{ne-typeexp-list}\rangle\ )
\end{array}$$

$$
\begin{array}{rcl}
 & | & \langle\text{typeexp}\rangle\ -> \ \langle\text{typeexp}\rangle \\
 & | & \langle\text{ident}\rangle\ (\ \langle\text{ne-typeexp-list}\rangle\ ) \\
\langle\text{ne-typeexp-list}\rangle & \rightarrow & (\ \langle\text{typeexp}\rangle\ ,)*\ \langle\text{typeexp}\rangle
\end{array}
$$

2.3.2 Static Semantics Checks

- Only basic types and type constructors listed in Section 3.1 are permitted. Type constructors may only be used according to their arity.
- Only arrow-free type expressions, or type expressions of the form $t_1 \rightarrow t_2$, where t_1 and t_2 are arrow-free, are allowed. The rationale of this restriction is that it guarantees any well-typed expression to have a non-functional type.
- The set of free variables of a type expression is defined as follows:

$$
\begin{array}{rcl}
\mathcal{V}(\text{basictype}) & = & \emptyset \\
\mathcal{V}(\text{typevariable}) & = & \{\text{typevariable}\} \\
\mathcal{V}((t_1,\ldots,t_n)) & = & \bigcup_{i=1}^{n} \mathcal{V}(t_i) \\
\mathcal{V}(\text{typeconstructor}(t_1,\ldots,t_n)) & = & \bigcup_{i=1}^{n} \mathcal{V}(t_i) \\
\mathcal{V}(t_1 \rightarrow t_2) & = & \mathcal{V}(t_2) - \mathcal{V}(t_1)
\end{array}
$$

  Only type expressions without free variables are allowed. The rationale of this restriction is that it guarantees every well-typed expression to have a variable-free type.

- Pre-defined functions (see Section 3.2) must not be re-defined in the signature, and no function symbol must be defined twice. The rationale of this restriction is that it guarantees any well-typed expresison to have exactely one minimal type [1].

2.4 Variable Declarations

Variable declarations are used in the several syntactic constructs.

2.4.1 Context-Free Syntax

A variable can be declared as *logical* (this is the default) or as *mutable* (indicated by the key word mutable). Logical variables can be assigned only once while mutable variables can be assigned any number of times. If no type is given in the declaration then the universal type *message* is assumed.

$$
\begin{array}{rcl}
\langle\text{declaration}\rangle & \rightarrow & \langle\text{ne-identlist}\rangle \mid \langle\text{ne-identlist}\rangle\ :\ \langle\text{typeexp}\rangle \\
 & | & \langle\text{ne-identlist}\rangle\ :\ \underline{\text{mutable}}\ \ \langle\text{typeexp}\rangle\ ? \\
\langle\text{nt-decllist}\rangle & \rightarrow & (\ \langle\text{declaration}\rangle\ ;)*\ \langle\text{declaration}\rangle
\end{array}
$$

[1] This restriction may in the future be replaced by a weaker one if necessary

2.4.2 Static Semantics Checks

- All type expressions in a declaration must be arrow-free and variable-free.
- No variable may be declared twice in a declaration list.

2.5 Axioms

2.5.1 Context-Free Syntax

The optional axiom listing consists of a list of equational axioms, preceeded by an optional list of variable declarations.

$$\langle\text{axioms}\rangle \quad \rightarrow \quad \underline{\text{axioms}}\ (\ \underline{\text{declare}}\ \langle\text{declarations}\rangle\ \underline{\text{begin}}\ )?\ (\ \langle\text{expr}\rangle\ =\ \langle\text{expr}\rangle\ ;\ )*\ \underline{\text{end}}$$

2.5.2 Static Semantics Checks

- All variables used in the axioms must be declared (hence, the declaration part may only be dropped when all axioms are ground).
- All variable declarations must be logical.
- The two sides of an equational axiom must be of the same type.

2.6 Roles

2.6.1 The Overall Structure

A role definition consists of a name, a list of parameters, and an instruction block.

$$\langle\text{role}\rangle \quad \rightarrow \quad \langle\text{ident}\rangle\ (\ \langle\text{nt-decllist}\rangle\ ?)\ \langle\text{block}\rangle$$

2.6.2 Expressions

$$\begin{array}{rcl}
\langle\text{expr}\rangle & \rightarrow & \langle\text{integer}\rangle \mid \langle\text{ident}\rangle \\
& \mid & \langle\text{expression}\rangle\ \langle\text{binop}\rangle\ \langle\text{expression}\rangle \\
& \mid & \langle\text{expression}\rangle\ \langle\text{binrel}\rangle\ \langle\text{expression}\rangle \\
& \mid & \langle\text{ident}\rangle\ (\ \langle\text{ne-exprlist}\rangle\ ) \\
& \mid & [\ \langle\text{expression}\rangle\ ,\ \langle\text{ne-exprlist}\rangle\ ] \\
& \mid & [[\ ]] \mid [[\ \langle\text{ne-exprlist}\rangle\ ]] \\
& \mid & (\ \langle\text{expression}\rangle\ ) \\
& \mid & \{(\ \langle\text{expression}\rangle\ :\ \langle\text{expression}\rangle\ ,)*\ \langle\text{expression}\rangle\ :\ \langle\text{expression}\rangle\ \} \\
\langle\text{ne-exprlist}\rangle & \rightarrow & (\ \langle\text{expression}\rangle\ ,)*\ \langle\text{expression}\rangle
\end{array}$$

Note that tuples have at least two elements.

2.6.3 Patterns

The construction $(exp : type)$ serves to check the type of an expression (this is useful for checking the type of a value obtained in a <u>recv</u> instruction).

$$
\begin{array}{rcl}
\langle \text{pattern} \rangle & \rightarrow & \langle \text{expression} \rangle \mid (\langle \text{expression} \rangle : \langle \text{typeexp} \rangle) \\
\langle \text{patterns} \rangle & \rightarrow & (\langle \text{pattern} \rangle ;)* \langle \text{pattern} \rangle
\end{array}
$$

2.6.4 Instructions

$$
\begin{array}{rcl}
\langle \text{block} \rangle & \rightarrow & (\underline{\text{declare}} \ \langle \text{nt-decllist} \rangle ;)? \ \underline{\text{begin}} \ \langle \text{instlist} \rangle \ \underline{\text{end}} \\
\langle \text{instruction} \rangle & \rightarrow & \langle \text{raw-instruction} \rangle \mid \langle \text{ident} \rangle : \langle \text{raw-instruction} \rangle \\
\langle \text{raw-instruction} \rangle & \rightarrow & \langle \text{block} \rangle \\
& \mid & \underline{\text{send}} \ (\langle \text{expression} \rangle) \mid \underline{\text{recv}} \ (\langle \text{patterns} \rangle) \\
& \mid & \underline{\text{fail}} \\
& \mid & \langle \text{ident} \rangle := \langle \text{expression} \rangle \\
& \mid & \underline{\text{new}} \ (\langle \text{ident} \rangle) \\
& \mid & \underline{\text{match}} \ \langle \text{expression} \rangle \ \underline{\text{with}} \ \langle \text{patterns} \rangle \ \underline{\text{end}} \\
& \mid & \underline{\text{if}} \ \langle \text{expression} \rangle \ \underline{\text{then}} \ \langle \text{instlist} \rangle \ (\underline{\text{else}} \ \langle \text{instlist} \rangle)? \ \underline{\text{fi}} \\
& \mid & \underline{\text{choice}} \ \langle \text{instlist} \rangle \ (| \ \langle \text{instlist} \rangle)* \ \underline{\text{end}} \\
& \mid & \underline{\text{case}} \ \langle \text{expression} \rangle \ \underline{\text{of}} \ (\langle \text{case} \rangle ;)* \ \langle \text{case} \rangle \ \underline{\text{else}} \ \langle \text{instlist} \rangle \ \underline{\text{esac}} \\
\langle \text{case} \rangle & \rightarrow & \langle \text{patterns} \rangle -> \langle \text{instlist} \rangle \\
\langle \text{instlist} \rangle & \rightarrow & (\langle \text{instruction} \rangle ;)*
\end{array}
$$

Instructions may be prefixed with a label which serves to identify control points in a role.

A <u>recv</u> instruction never fails. It blocks until a value is received that is matched by one of the patterns.

Patterns are applied in sequentiel order.

In a case statement, the cases are tried in sequential order. If none of the cases applies then the *else* case is entered.

In an expression, all variables have to be instantiated (to be checked by static analysis).

In a pattern, the variables may be non-instantiated. However, all *mutable* variables occuring in a patten must be instantiated (to be checked by static analysis). However there are some sub-terms that always have to be instantiated, in particular the key argument of a crypt operation. The static analysis has to verify that it is possible to instantiate the variables such that this constraint is satisfied.

In case of a branching instruction (like case) all banches must instantiate the same variables.

2.7 Declaration of Global Variables

There is another kind of variable declaration with explicit skolemisation. This extended form of variable declaration can only be used in order to define global variables to be used in scenarios, not in role definitions. If a variable is declared in dependance of other quantified names (like $x(y,z)$ where y and z are quantified in some surrounding scope) then there is a new instance of variable x for every y and every z. However, if this declaration is also in the scope of another universal quantifier (say, $forall\ v$) then $x(y,z)$ is shared by all possible values of v. This can be thought of extending a simple variable declaration to a declaration of a Skolem function.

$$
\begin{array}{rcl}
\langle\text{globals}\rangle & \rightarrow & \underline{\text{variables}}\ \langle\text{skolem-decllist}\rangle\ \underline{\text{end}} \\
\langle\text{skolem-declaration}\rangle & \rightarrow & \langle\text{skolem-identlist}\rangle \mid \langle\text{skolem-identlist}\rangle\ :\ \langle\text{typeexp}\rangle \\
 & \mid & \langle\text{skolem-identlist}\rangle\ :\ \underline{\text{mutable}}\ \langle\text{typeexp}\rangle\ ? \\
\langle\text{skolem-decllist}\rangle & \rightarrow & (\ \langle\text{skolem-declaration}\rangle\ ;\ )* \\
\langle\text{skolem-identlist}\rangle & \rightarrow & (\ \langle\text{skolem-ident}\rangle\ ,\ )*\ \langle\text{skolem- ident}\rangle \\
\langle\text{skolem-ident}\rangle & \rightarrow & \langle\text{ident}\rangle\ (\ \langle\text{ne-identlist}\rangle\ )
\end{array}
$$

2.8 L-Expressions

L-expression are expressions that may denote a memory location, that is expressions that are allowed to occur on the left-hand side of an assignment, or as argument of a new instruction. In roles the only legal l-expressions would be identifiers (this is why we did not need this notion for the definition of role). In case of scenarios, l-expression may be either identifiers, or identifiers applied to some expressions.

$$
\langle\text{l-expression}\rangle \ \rightarrow\ \langle\text{ident}\rangle\ (\ \langle\text{ne-exprlist}\rangle\ )?
$$

2.9 Scenario

$$
\begin{array}{rcl}
\langle\text{scenario}\rangle & \rightarrow & \underline{\text{scenario}}\ \langle\text{sceneinst}\rangle\ \underline{\text{end}} \\
\langle\text{sceneinst}\rangle & \rightarrow & \langle\text{raw-sceneinst}\rangle \mid \langle\text{ident}\rangle\ :\ \langle\text{raw-sceneinst}\rangle \\
\langle\text{raw-sceneinst}\rangle & \rightarrow & \langle\text{ident}\rangle\ (\ \langle\text{exprlist}\rangle\ ) \\
 & | & \langle\text{l-expression}\rangle\ :=\ \langle\text{expression}\rangle \\
 & | & \underline{\text{new}}\ (\ \langle\text{l-expression}\rangle\ ) \\
 & | & \underline{\text{makekey}}\ (\ \langle\text{expression}\rangle\ ,\ \langle\text{l-expression}\rangle\ ) \\
 & \mid & \underline{\text{if}}\ \langle\text{expression}\rangle\ \underline{\text{then}}\ \langle\text{scenelist}\rangle\ (\ \underline{\text{else}}\ \langle\text{scenelist}\rangle\ )?\ \underline{\text{fi}} \\
 & | & \underline{\text{forever}}\ \langle\text{scenelist}\rangle\ \underline{\text{end}} \\
 & | & \underline{\text{parallel}}\ (\ \langle\text{sceneinst}\rangle\ |)*\ \langle\text{sceneinst}\rangle\ \underline{\text{end}} \\
 & | & \underline{\text{begin}}\ \langle\text{scenelist}\rangle\ \underline{\text{end}} \\
 & | & \underline{\text{exists}}\ \langle\text{valuedec}\rangle\ .\ \langle\text{sceneinst}\rangle \\
 & | & \underline{\text{forall}}\ \langle\text{valuedec}\rangle\ (\ \langle\text{relation-constraint}\rangle\ )?.\ \langle\text{sceneinst}\rangle
\end{array}
$$

$$
\begin{array}{rcl}
 & | & \underline{\text{declare}}\ \langle\text{nt-decllist}\rangle\ ;\ \underline{\text{begin}}\ \langle\text{scenelist}\rangle\ \underline{\text{end}} \\
\langle\text{valuedec}\rangle & \rightarrow & \langle\text{ne-identlist}\rangle\ :\ \langle\text{typeexp}\rangle\ (\ ,\ \langle\text{ne-identlist}\rangle\ :\ \langle\text{typeexp}\rangle\ )? \\
\langle\text{scenelist}\rangle & \rightarrow & (\ \langle\text{sceneinst}\rangle\ ;)* \\
\langle\text{relation-constraint}\rangle & \rightarrow & [\ \langle\text{interval}\rangle\ (\ ,\ \langle\text{interval}\rangle\ )*] \\
\langle\text{interval}\rangle & \rightarrow & \langle\text{omega-number}\rangle\ |\ \langle\text{omega-number}\rangle\ ..\ \langle\text{omega-number}\rangle \\
\langle\text{omega-number}\rangle & \rightarrow & \langle\text{positive-integer}\rangle\ |\ *
\end{array}
$$

Scenarios may be prefixed with a label which serves to identify control points in a scenario.

The first case in the definition of a scenario serves both for invoking roles.

The key word forever indicates an infinite sequential loop.

The key words exists and forall allow to quantify over values of a type.

The meaning of the relational constraint is the following: A quantified expression of the form

$$\underline{\text{forall}}\ x_1 : t_1, \dots, x_n : t_n\ \textit{constraint}\ .\ S$$

should be understood as

$$\exists R :\ \textit{constraint}\ \forall x_1 : t_1, \dots, x_n : t_n . R(x_1, \dots, x_n) \Rightarrow S$$

where we say that a n-ary relation R satisfies a constraint $[l_1..u_1, \dots, l_n..u_n]$ iff

$$\bigwedge_{i=1}^{n} \forall x_1 : t_1, \dots, x_{i-1} : t_{i-1}, x_{i+1} : t_{i+1}, \dots, x_n : t_n . l_i \leq \#\{(x_1, \dots, x_n) \mid R(x_1, \dots, x_n)\} \leq u_i$$

We allow to abbreviate an interval $n..n$ as n, and $*$ stands for infinity. Hence, in case of binary relations (n=2), $[1,1]$ denotes a relation which is a bijective function, $[1..*,1]$ a function which is one-to-one, and $[0..1,1]$ a function which is onto. If no constraint is given then the default is $[0..*, \dots, 0..*]$, which is satisfied by any relation.

2.10 Lexical Tokens

The following lexer tokens are defined in the syntaxe of OcamlLex :

```
let number   =  ['0'-'9']
let letter   =  ['a'-'z''A'-'Z']
let ident    =  letter(letter|number|'_')*
let integer  =  number+
```

2.11 Infix Operators

All infix operators associate to the left. Following is a table of infix operators in increasing order of precedence (that is, operators at the top of the table are stronger binding than operators at the bottom of the table).

<table>
<tr><th>Operator</th><th>Comment</th></tr>
<tr><td>.</td><td>Selection operator in association lists</td></tr>
<tr><td>+, -
::</td><td>Arithmetical addition, substraction
cons operation on polymorphic lists</td></tr>
<tr><td>=, <, >, =<, >=, !=</td><td>Comparison operators</td></tr>
<tr><td>&&</td><td>Logical conjunction</td></tr>
<tr><td>||</td><td>Logical disjunction</td></tr>
</table>

Chapter 3

The Default Signature

3.1 Pre-Defined Types

Following is the list of pre-defined type constants. *message* is the universal type comprising all non-functional types.

Name	Comment
`message`	the type of all (non-functional) values
`int`	the type of integers
`bool`	the type of boolean values
`nonce`	the type of nonces
`algo`	the enumeration type of encryption algorithms
`pubkey`	the type of public encryption keys
`privkey`	the type of private decryption keys (or private signing keys)
`principal`	the type of principals

Furthermore, we have the following type constructors:

Name	Arity	Comment
`list`	1	polymorphic lists
`table`	2	polymorphic association lists

3.2 Pre-Defined Constants and Functions

3.2.1 Bool

The domain of the type `bool` consists of the two values *true* and *false*. The following operations yield boolean values:

Name	Type	Comment
`true`, `false`,	`bool`	Constant values
`&&`	(`bool`, `bool`) → `bool`	Conjunction (infix operator)
`||`	(`bool`, `bool`) → `bool`	Disjunction (infix operator)
`not`	`bool` → `bool`	Negation
`=`, `<`, `>`, `=<`, `>=`, `!=`	(`'a`, `'a`) → `bool`	Polymorphic comparison operators

3.2.2 Int

Numerical constants, binary operators +, -.

3.2.3 Algo

This is an enumeration type for different encryption and signature algorithms. The following constants are defined:

Name	Type	Comment
`rsa`	`algo`	Rivest-Shamir-Adleman method
`des`	`algo`	Data Encryption Standard
`sym`	`algo`	A not further specified symmetric encryption method
`asym`	`algo`	A not further specified asymmetric encryption method

3.2.4 Keys

The instruction `makekey(algo, var)`, where *algo* is instantiated and of type *algo*, assigns to *var* a new key. In case of asymmetric encryption the variable *var* will then hold the *private* part of the key pair. The *public* part of the key pair can be obtained as $inv(key)$. Static analysis will ensure that *inv* can only be applied when a makekey operation was executed on the same *var*. The `inv` operation is an exception in that it's type is overloaded.

Name	Type
`sign`	(`algo`,`message`,`privkey`)$\longrightarrow$`message`
`crypt`	(`algo`,`message`,`pubkey`)$\longrightarrow$`message`
`inv`	`pubkey`$\longrightarrow$`privkey` `privkey`$\longrightarrow$`pubkey`

3.2.5 Nonce

There are no constants of this type, or functions that yield these values (a nonce is created by new). The only test operation on nonces is equality test (explicit, or implicit in pattern matching).

3.2.6 Lists

$[[]]$ is the empty list

$[[a_1,a_2,a_3]]$ is the list consisting of the three values a_1, a_2, a_3.

$x :: y$ yields the list with head x and rest y.

$element(x,y)$ tests whether x is contained in the list y.

Lists are monomorphic, that is all elements must be of the same type. Note that this type may be the universal type *message*. Hence, $[[1,true]]$ is of type *list(message)*.

3.2.7 Tuples

The set of types is closed under tupling. For instance, $[a_1, a_2, a_3]$ is the triple consisting of the three values a_1, a_2, a_3. Tuples always contain at least two elements, that is there are no zero-tuples or one-tuples. As a concequence, the expression [[42]] denotes a list of length 1, and not a one-tuple of a one-tuple.

There are no operations defined on tuples. The only way to access the components of a tuple is to use pattern-matching.

3.2.8 Finite Functions

Finite functions (or asociation lists) are written in the form

$$\{key_1 : value_1, key_2 : value_2, \ldots, key_n : value_n\}$$

If l is a finite function then $l.x$ yields the value of l for key x.

3.3 Restriction to the `New` Instruction

The argument to <u>new</u> must be of one of the types *nonce*, *int*, or *message*.

Chapter 4

Examples

4.1 The IKA-1 Protocol

```
signature
        a, b, c: principal;
        g: pubkey;                # the base for the generation of the shared key
        m: message;                   # the secret message
        exp: (pubkey,nonce) → pubkey;
end

role A (myname: principal)
        declare
        mynonce: nonce;
        begin
                new(mynonce);
                send([g,exp(g,mynonce)]);
                declare
                x,y: pubkey;
                begin
                        recv([x,y]);
                        send(crypt(sym,exp(x,mynonce),m));
                end;
        end

role B (myname: principal)
        declare
        mynonce: nonce;
        begin
                new(mynonce);
                declare x,y: pubkey;
                begin
                        recv([x,y]);
                        send([exp(x,mynonce),y,exp(y,mynonce)]);
                end;
```

```
        end

role C (myname: principal)
        declare
        mynonce: nonce;
        begin
                new(mynonce);
                declare x,y,z : pubkey;
                begin
                        recv([x,y,z]);
                        send([exp(x,mynonce),exp(y,mynonce)]);
                end;
        end

scenario
        parallel A(a) | B(b) | C(c) end
end
```

4.2 The Needham-Schroeder Public Key Protocol

```
# The Needham–Schroeder Protocole with Lowe patch

# From the Spore repository :
#
# A,B,S :       Principal
# Na,Nb :       Nonce
# KPa,KPb,KPs,KSa,KSb,KSs :     Key
# KPa,KSa :     is a key pair
# KPb,KSb :     is a key pair
# KPs,KSs :     is a key pair
#
# 1.    A       →       S       :       A,B
# 2.    S       →       A       :       {KPb, B}KSs
# 3.    A       →       B       :       {Na, A}KPb
# 4.    B       →       S       :       B,A
# 5.    S       →       B       :       {KPa, A}KSs
# 6.    B       →       A       :       {Na, Nb, B}KPa
# 7.    A       →       B       :       {Nb}KPb

signature
        alice, bob, intruder: principal;
        init_connection_with: principal → message;
        pubkey_of: principal → pubkey;
end

# we assume that Alice initiates a session on receiving a request
# of the form init_connection_with(name)
role Alice (my_name: principal; my_key: privkey;
            server_signkey: pubkey)
```

```
declare
        my_nonce, bob_nonce: nonce;
        bob_pubkey: pubkey;
        bob_name: principal;
        begin
                recv(init_connection_with(bob_name));
                send([my_name,bob_name]);
                recv(sign(asym,inv(server_signkey),[bob_pubkey,bob_name]));
                new(my_nonce);
                send(crypt(asym,bob_pubkey,[my_nonce,my_name]));
                recv(crypt(asym,inv(my_key),[my_nonce,bob_nonce,bob_name]));
                send(crypt(asym,bob_pubkey,bob_nonce));
        end

role Bob (my_name: principal; my_key: privkey;
   server_signkey: pubkey)
        declare
        my_nonce, alice_nonce : nonce;
        alice_name: principal;
        alice_pubkey: pubkey;
        begin
                recv(crypt(asym,inv(my_key),[alice_nonce,alice_name]));
                send([my_name,alice_name]);
                recv(sign(asym,inv(server_signkey),[alice_pubkey,alice_name]));
                new(my_nonce);
                send(crypt(asym,alice_pubkey,[alice_nonce,my_nonce,my_name]));
                recv(crypt(asym,inv(my_key),my_nonce));
        end

role Server (my_signkey: privkey)
        declare
        requester, query: principal;
        begin
                recv([requester,query]);
                send(sign(asym,my_signkey,[pubkey_of(query),query]));
        end

role Send_initial_request (name: principal)
        begin
                send(init_connection_with(name));
        end

variables
        alice_key, bob_key, server_key: privkey;
end

scenario
        begin
                makekey(asym, server_key );
                Send_initial_request(bob);
```

```
            parallel
                    Alice  (alice,alice_key, inv(server_key))
                    | Bob  (bob,  bob_key,   inv(server_key))
                    | forever Server(server_key) end
            end
    end
end
```

4.3 The TMN Protocol

```
# key distribution protocol by Tatebayashi, Matsuzaki, and Newman
# see for instancs CACM Nov 1994, page 58

signature
        xor: (message,message) → message;
        null: message;
        mult: (message,message) → message;
        one: message;
        inverse: message → message;
end

axioms
        declare
        bx, by, bz, x, y, z;
        key: pubkey;
        begin
                # the xor theory
                xor(bx,by) = xor(by,bx);
                xor(bx,xor(by,bz)) = xor(xor(bx,by),bz);
                xor(bx,null) = bx;
                xor(bx,bx) = null;

                # multiplication
                mult(x,y) = mult(y,x);
                mult(x,mult(y,z)) = mult(mult(x,y),z);
                mult(x,one) = x;
                mult(x,inverse(x)) = one;

                # rsa
                crypt(rsa,key,mult(x,crypt(rsa,key,y))) = crypt(rsa,key,mult(x,y));
        end

role alice(my_name,bob_name: principal; server_key: pubkey)
        declare
        my_nonce: nonce;
        shared_nonce: message;
        server_message: message;
        begin
                new(my_nonce);
```

```
            send([crypt(asym, server_key, my_nonce), my_name, bob_name]);
            recv(server_message);
            shared_nonce ← xor(server_message, my_nonce);
        end

role bob(my_name: principal; server_key: pubkey)
        declare
        shared_nonce: nonce;
        alice_name: principal;
        begin
            recv(alice_name);
            new(shared_nonce);
            send(crypt(asym, server_key, shared_nonce));
        end

role server(seen_keys: mutable list(nonce); my_secretkey: privkey)
        declare
        initiator_nonce, partner_nonce: nonce;
        initiator_name, partner_name: principal;
        begin
            recv([crypt(asym, inv(my_secretkey), initiator_nonce),
      initiator_name, partner_name]);
            if element(initiator_nonce, seen_keys)
            then
                fail;
            else
                seen_keys ← initiator_nonce :: seen_keys;
                send([initiator_name, partner_name]);
                recv(crypt(rsa, inv(my_secretkey), partner_nonce));
                send(xor(initiator_nonce, partner_nonce));
            fi;
        end

variables
        Alice, Bob: principal;
        serverkey: privkey;
        seen_keys: mutable list(nonce);
end

scenario
        begin
            makekey(rsa, serverkey);
            seen_keys ← [[ ]];
            parallel
                alice(Alice, Bob, inv(serverkey))
                | bob(Bob, inv(serverkey))
                | forever server(seen_keys, serverkey); end
            end;
        end
end
```

4.4 The Electronic Purse Protocol with Symmetric Keys

```
# Electronic Purse version with symmetric keys

signature
        # the function F produces a key, which is to be shared between a purse
        # and a module, from the master key of the module and the name of the purse
        F: (message, message) → message;

        # the function FS takes a symmetric sign key k and list messages l, and
        # returns l signed with k.
        FS: (message, list(message)) → message;

end

# the role of a purse during one transaction
role purse(
   ep_acqkey, ep_isskey;       # the two keys
   myname: principal;          # the name of the purse
   balance: mutable int        # the balance of the purse
   )
        declare
        ep_challenge, sam_challenge: nonce;
        sam_name: message;
        amount: int;
        s2, s3, s6: message;
        begin
                new(ep_challenge);
                send([myname, ep_challenge]);
                recv([sam_name, sam_challenge, s2]);
                if s2 ≠ FS(ep_acqkey, [[myname, ep_challenge]]) then fail; fi;
                recv(amount);
                balance ← balance − amount;
                s6 ← FS(ep_isskey, [[sam_name, sam_challenge, ep_challenge, amount]]);
                s3 ← FS(ep_acqkey, [[sam_name, sam_challenge, ep_challenge, s6, amount]]);
                send([s3, s6, amount]);
        end


        # the role of an application module during one transaction
role secure_application_module(
   acqkey;                        # the key
   myname: principal;             # the name of the module
   balance: mutable int;          # the balance of the module
        trace: mutable list(message) # audit
   )
        declare
        ep_acqkey: message;
        id_ep, ep_challenge: message;
```

```
        s2, s3, s6: message;
        sam_challenge: nonce;
        amount: int;
        begin
                recv([id_ep, ep_challenge]);
                ep_acqkey ← F(acqkey, id_ep);
                s2 ← FS(ep_acqkey, [[id_ep, ep_challenge]]);
                send([myname, sam_challenge, s2]);
                recv([s3, s6, amount]);
                if
                s3 = FS(ep_acqkey, [[myname, sam_challenge, ep_challenge, s6, amount]])
                then
                        balance ← balance + amount;
                        trace ← s6 :: trace;
                else
                        fail;
                fi;
        end

        # the keyboard where the user types the sum
role keyboard ()
        declare amount: int;
        begin
                new(amount);
                send(amount);
        end

variables
        ep_isskey(principal): int;                # issuer key of a purse
        # alternative : one master issuer key ?
        acqkey: int;                              # master key
        ep_balance(principal) : mutable int;      # balance of a purse
        sam_balance(principal): mutable int;      # balance of a module
        trace(principal) : mutable list(message); # audit trail of a module

end

scenario
        begin

                # compute the master key
                new(acqkey);

                # compute the issuer keys per purse
                forall purse: principal . new(ep_isskey(purse));

                # initialize the traces
                forall module: principal . trace(module) ← [[]];

                # infinitely start parallel sessions between purses and sams, such that
```

```
                # at every moment a purse or an agent is engaged in at most one
                # transaction
                forever
                  forall purse,module: principal [ 1,1 ].
                     parallel
                       purse(F(acqkey,purse),
                            ep_isskey(purse), purse, ep_balance(purse))
                     | secure_application_module(acqkey,module,
                            sam_balance(module), trace)
                     | keyboard()
                     end;
                end;  # forever
        end
end        # scenario
```

4.5 The Electronic Purse Protocol with Asymmetric Keys

```
# Electronic Purse version with asymmetric keys

signature
        # the function F produces a key, which is to be shared between a purse
        # and a module, from the master key of the module and the name of the purse
        F: (message,message) → message;

        # the function FS takes a symmetric sign key k and list messages l, and
        # returns l signed with k.
        FS: (message, list(message)) → message;

        # the hash function
        h: list(message) → message;

        b: int; # the generator of the group

        # standard arithmetic functions
        exp: (int,int) → int;
        mult: (int,int) → int;
        minus: int → int;

        # the signing key of the key creation authorit
        certkey: privkey;

end

axioms
        declare
        x, y, z: int;
        begin
           x + y = y + x;
                x + (y + z) = (x + y) + z;
```

```
                minus(x) + x = 0;
                x + 0 = x;
                exp(exp(x,y),z) = exp(x,mult(y,z));
                mult(exp(x,y),exp(x,z)) = exp(x,y+z);
        end

        # the role of a purse during one transaction
role purse(
   ep_acqkey, ep_isskey;        # the two keys
   myname: principal;           # the name of the purse
   balance: mutable int;        # the balance of the purse
   certificat: message;         # the certificate of the purse
   s: int                       # the secret s
   )
        declare
        ep_challenge,sam_challenge: nonce;
        sam_name: message;
        amount, coupon: int;
        s2,t,s6: message;
        p: int; # the public key of the purse
        x: int;
        c: int;
        begin
                p ← exp(b,minus(s));
                new(ep_challenge);
                send([myname,ep_challenge,p,certificat]);
                recv([sam_name,sam_challenge,s2]);
                if s2 ≠ FS(ep_acqkey,[[myname,ep_challenge]]) then fail; fi;
                recv(amount);
                s6 ← FS(ep_isskey,[[sam_name,sam_challenge,ep_challenge,amount]]);
                new(x);
                coupon ← exp(b,x);
                t ← h([[coupon,sam_name,sam_challenge,ep_challenge,s6,amount]]);
                send(t);
                recv(c);
                balance ← balance − amount;
                send([x + mult(s,c),s6,amount]);
        end


        # the role of an application module during one transaction
role secure_application_module(
   acqkey;                      # the key
   myname: principal;           # the name of the module
   balance: mutable int;        # the balance of the module
   signcheckkey: pubkey;        # the key to check a signature
        trace: mutable list(message) # audit
   )
        declare
        ep_acqkey: message;
```

```
        id_ep, ep_challenge: message;
        s2,t,u,s6: message;
        s3: int;
        sam_challenge: nonce;
        amount: int;            # the amount to charge
        p: int;                 # public key of the purse
        c: int;
        begin
                recv([id_ep,ep_challenge,p,
      sign(rsa,inv(signcheckkey),[id_ep,p])]);
                ep_acqkey ← F(acqkey,id_ep);
                s2 ← FS(ep_acqkey,[[id_ep,ep_challenge]]);
                send([myname,sam_challenge,s2]);
                recv(t);
                new(c);
                recv([s3,s6,amount]);
                u ← mult(exp(p,c),exp(b,s3));
                if
                t = h([[u,myname,sam_challenge,ep_challenge,s6,amount]])
                then
                        balance ← balance + amount;
                        trace ← [s6, sam_challenge, ep_challenge, amount] :: trace;
                else
                        fail;
                fi;
        end

        # the keyboard where the user types the sum
role keyboard ()
        declare amount: int;
        begin
                new(amount);
                send(amount);
        end

variables
        ep_isskey(principal): int;                # iss key of a purse
        secret(principal) :int;                   # secret key used by the purse
        acqkey: int;                              # master key
        ep_balance(principal) : mutable int;      # balance of a purse
        sam_balance(principal): mutable int;      # balance of a module
        trace(principal) : mutable list(message); # audit trail of a module

end

scenario
        begin

                # compute the master key
                new(acqkey);
```

```
                # compute the purse keys
                forall purse: principal . begin
                        new(ep_isskey(purse));
                        new(secret(purse));
                end;

                # initialize the traces
                forall module: principal . trace(module) ← [[ ]];

                # infinitely start parallel sessions between purses and sams, such that
                # at every moment a purse or an agent is engaged in at most one
                # transaction
                forever
                        forall purse, module: principal [ 1,1 ] .
                        parallel
                                purse(F(acqkey,purse),
                                ep_isskey(purse), purse, ep_balance(purse),
                                sign(asym,certkey,[purse,exp(b,minus(secret(purse)))]),
                                secret(purse))
                                | secure_application_module(acqkey,module,
                                sam_balance(module),
                                inv(certkey),
                                trace(module))
                                | keyboard()
                        end;
                end; # forever
        end     # begin
end         # scenario
```

Chapter 5

Discussion

Points that remain to decide later :

- axioms: are equations enough ? We certainly would like to provide for pre-defined standard theories, like $AC(+)$ etc.
- types *pubkey* and privkey: shouldn't they be parameterized by the encryption algorithm (like *pubkey(rsa)*).
- finite functions: what is the value if a key is not defined in an association list?
- Not yet implemented: types defined by the user
- Not yet implemented: inductive types