







# RAPPORT TECHNIQUE PROUVÉ

## Translation from PROUVÉ to ACTAS


**Auteur** : Hitoshi Ohsaki and Ralf Treinen

**Date** : Mai 2007

**Rapport PROUVÉ numéro** : 11

**Loria**
CNRS UMR 7503,
Campus Scientifique - BP 239
54506 Vandoeuvre-ls-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 Dveloppement
38, 40 rue du Gnral Leclerc
92794 Issy Moulineaux Cedex
www.rd.francetelecom.fr

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

Research Center for Verification and Semantics,
National Institute of Advanced Industrial Science and Technology (AIST),
Midorigaoka 1-8-31, Ikeda, Osaka 563-8577, Japan

# 1 The PROUVÉ Protocol Specification Language

The PROUVÉ specification language for cryptographic protocols is described in [KLT05]. Its purpose 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 together with equational axioms defining the semantics of message constructors.

The sub-language used to define protocols is inspired by existing imperative programming languages. This approach is different from the traditional so-called Alice-Bob notation which is popular in the field of cryptographic protocols. This Alice-Bob notation intends to define a protocol by presenting how an execution between honest participants looks like as seen by an outside observer. One important design objective of the PROUVÉ language is to give means to describe in an unambiguous way the actions of the protocol participants. This is achieved by describing a protocol as a set of distributed programs, so-called *roles* which are executed by (legitimate) protocol participants.

We use 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 exclude 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, conditional branching, case distinction, and non-deterministic choice. 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.

The accompanying assertion language allows to express safety properties of execution traces of protocol scenarios.

The design of the PROUVÉ specification language has been inspired by the languages muCAPSL [MD02] and CASRUL [JRV00].

# 2 The ACTAS Verification Tool

ACTAS is an integrated system for manipulating associative and commutative tree automata (AC-tree automata for short). It has various functions such as for Boolean operations of AC-tree automata, computing rewrite descendants, and solving emptiness and membership problems. In order to deal with high-complexity problems in reasonable time, over- and under-approximation algorithms are provided. This functionality enables automated verification of safety property in infinite state models. This is useful in the domain of, e.g. network security, in particular, for security problems of cryptographic protocols using equational properties of cryptographic promites.

In runtime of model construction, a tool support for analysis of state space expansion is provided. The intermediate status of the computation is displayed in numerical data table, and the line graphs are generated. Besides, a graphical user interface of the system provides a user-friendly environment for convenient use.

The ACTAS tool is developed by Hitoshi Ohsaki and Toshinori Takai at the National Institute of Advanced Industrial Science and Technology (AIST), Kansai branch, Japan [OT05]. It is available at `http://staff.aist.go.jp/hitoshi.ohsaki/actas/index.html`.

# 3 The Stateless Fragment of PROUVÉ

We now define the so-called *stateless* fragment of the PROUVÉ specification language. In general, a PROUVÉ protocol specification consists of five parts which we describe for the special case of stateless protocols in the following. Finally, we define which form of *assertions* can be treated by the translator for the stateless fragment.

## 3.1 Signature

This optional section defines an extension of the default-signature by user-defined functions. We have no particular restriction on the kind of signature extension allowed in the stateless fragment.

## 3.2 Axioms

This optional section gives a list of equational axioms which define the semantics of the (both default and user-defined) function symbols.

## 3.3 Roles

This required section of a protocol specification defines a set of named and parameterized programs which are intended for execution by legitimate protocol participants. A role definition is given by its name, a list of parameters (which in our case must all be call-by-value parameters), and a list of instructions. In the stateless fragment we allow only two kinds of instructions:

- a send instruction of the form `send(s)` where `s` is a term, possibly containing parameters of the role. Its semantic is to send the value of this term on the public channel.
- a receive-send block, consisting of
  - an optional declaration of local variables to this block, which must in our case all by logical (write-once) variables;
  - a receive-instruction of the form `recv(r)`, where `r` may contain parameters of the role or variables declared to this block;
  - a send-instruction `s` as before, except that `s` may now also contain variables local to that block which have been instantiated by the preceding receive-instruction.

  The semantics of this block is that on recpetion of a term matched by $r$ with matching substitution $\sigma$ the term $s\sigma$ is send on the public channel.

Figure 1 gives an example of a role specification belonging to the stateless fragment.

## 3.4 Global Variables

In this optional section we may declare some logical (write-once) variables of any of the types `int`, `nonce`, or `message`. Definition of skolemized variables as defined in [KLT05] is not allowed in the stateless fragment.

```
role Alice (myname: principal; mynonce: nonce)
begin
  send([keybase,exp(keybase,mynonce)]);
  declare
    x,y: symkey;
  begin
    recv([x,y]);
    send(symcrypt(sym,exp(x,mynonce),secret));
  end;
end
```

Figure 1: The role *Alice* in the IKA1 protocol.

```
scenario
  begin
    new(nonce_a);
    new(nonce_b);
    new(nonce_c);
    parallel
      Alice(alice, nonce_a)
    | Bob(bob, nonce_b)
    | Charley(charley, nonce_c)
    end
  end
end
```

Figure 2: The scenario of the IKA1 protocol.

### 3.5 Scenarios

This required section defines how roles are instantiated. In the stateless fragment, this section consists of an (possibly empty) sequence of new instructions (a $\mathsf{new}(x)$ instruction assigns a fresh value to the variable $x$), followed by a parallel execution of several role invocations. A role invocation consists of the name of a role and a list of formal parameters, a formal parameter being a term possibly containing variables.

Figure 1 gives an example of a scenario belonging to the stateless fragment.

### 3.6 Assertions

An assertion consists of four parts:

1. The name of the protocol specification it uses.
2. The list of function symbols which can be used by an intruder to construct new terms.
3. An *initial* formula which expresses an initial condition on the protocol execution traces to be considered. In case of the stateless fragment, this is a just a finite list of ground terms which describe the initial knowledge of an intruder.

4. A *safety* formula which is intended to be true everywhere on all protocol execution traces which satisfy the initial formula. In case of the stateless fragment, this formula is of the form `issecret(t)` where $t$ is a ground term, meaning that the intruder is not able to deduce the term $t$.

# 4 Translation of the Stateless Fragment to ACTAS

In this section we describe how the ACTAS input file is constructed from a given PROUVÉ specification in the stateless fragment.

## 4.1 Comment

The comment section contains the name of the translated assertion file and a summary of the used translation options.

## 4.2 Signature

The signature section contains :

- The declaration of all the variables used locally in the rules (but not the global variables, since these are treated as constants), and of the additional variables used in the generated term rewrite system.
- All constants, both from the default signature and user-defined ones
- A list of function symbols for which *AC*-axioms have been given, except when the translation was done using the `-noac` optios. With the `-noac` option, all axioms including AC axioms are translated into term rewrite rules.

## 4.3 The Term Rewrite System

The term rewrite system consists of several parts:

- rules corresponding to send instructions: If a role $R$ with formal parameters $\bar{x}$ contains a send instruction $\mathtt{send}(s)$ then we generate the term rewrite rule
$$\mathtt{S\_}R(\bar{x}) \to s$$
- rules corresponding to receive-send blocks: If a role $R$ with formal parameters $\bar{x}$ contains a block $\mathtt{recv}(r);\mathtt{send}(s)$ then we generate the term rewrite rule
$$\mathtt{RS\_}R(\bar{x},r) \to s$$
- rules for the Dolev-Yao computation rules, like for instance:
$$decrypt(x,y,crypt(x,inverse(y),z)) \to z$$
- rules corresponding to the equational axioms: For every equational axiom $e_1 = e_2$ we generate the two rules
$$e_1 \to e_2 \qquad\qquad e_2 \to e_1$$
As an exception, AC axioms are not translated like this, except when the option `-noac` is used for the translation.

### 4.4 The Automaton Describing the Intruder Capabilities

The automaton describing the intruder capabilities consists of several parts. In the following description, we will use generalized transition rules of the form $t \to q$ where $t$ is a ground term (probably containing state symbols) and $q$ is a state symbol. This can obviously be translated into a system of flat transition rules by using some fresh auxiliary states.

For every type $t$ occurring as type of some (sub-)term of the protocol, the initial intruder knowledge, or the secret term we have a state symbol $q_t$. (TODO: explain handling of polymorphic types!!)

- Initial intruder knowledge: For every term $t$ in the initial intruder knowledge we have a transition
$$t \to q_{type(t)}$$
- Functions known to the intruder: For every function symbol
$$f : type_1, \ldots, type_n \to type$$
available to the intruder (these are the function symbols from the default signature and those function symbols from the user-defined signature that have been declared public in the assertion file) we have a transition:
$$f(q_{type_1}, \ldots, q_{type_n}) \to q_{type}$$
- The type lattice: If the type $type_1$ is smaller than the type $type_2$ then we have a transition
$$q_{type_1} \to q_{type_2}$$
Note that, due to the existence of polymorphic type constructors in PROUVÉ, there are infinitely many types in PROUVÉ. However only a finite number of types can be relevant for a given protocol specification and assertion.
- Triggering send actions: If a role $R$, which is invoked with actual parameters $\bar{a}$, contains a send instruction $\texttt{send}(s)$ then we generate the term rewrite rule
$$\texttt{S\_}R(\bar{a}) \to q_{type(s)}$$
- Triggering receive-send blocks: If a role $R$, which is invoked with actual parameters $\bar{a}$, contains a block $\texttt{recv}(r);\texttt{send}(s)$ then we generate the term rewrite rule
$$\texttt{RS\_}R(\bar{a}, q_{type(r)}) \to q_{type(s)}$$

The accepting state is $q_{message}$.

In case we use a translation with a bounded depth of message exchanges (using option `-depth` $n$ for $n$ message exchanges) w obtain a more complicated translation:

- Every state $q$ as explained below exists in $n+1$ copies $q_{step_0}, \ldots, q_{step_n}$.
- The initial intruder knowledge is recognized in step 0.
- The transitions for the function symbols known to the intruder and for the type lattice exists for each step.

- The transitions for triggering actions have on the left-hand side states of step $i$ and on the right-sides states of step $i+1$, for every $i \in 0 \ldots n-1$.
- We have new transitions to move up in time: For every type $t$ and $i \in 0 \ldots n-1$ we have a transition

$$q_{t_{step\ i}} \rightarrow q_{t_{step\ i+1}}$$

The accepting state of this automaton is $q_{message_{step\ 0}}$.

### 4.5 The Automaton Describing the Secret Knowledge

The automaton for the secret knowledge $t$ is

$$t \rightarrow q_{secret}$$

where $q_{secret}$ is the accepting state.

## 5 Examples

### 5.1 Diffie-Helman

The Diffie-Helman key distribution protocol [DH76] is expressed in PROUVÉ as follows:

```
# Diffie-Helman key exchange

signature
  kap: (symkey,int) → symkey;
  mult: (int,int) → int;
  g: symkey;
  secret: message;
  alice_expo, bob_expo: int;
end

axioms
  declare
    x, y, z : int;
  begin
    kap(kap(g,x),y) = kap(g,mult(x,y));
    mult(x,mult(y,z)) = mult(mult(x,y),z);
    mult(x,y) =mult(y,x);
  end

role Alice(my_expo: int)
  begin
    send(kap(g,my_expo));
    declare
      bob_message: symkey;
    begin
      recv(bob_message);
```

```
      send(symcrypt(sym,kap(bob_message,my_expo),secret));
    end;
  end

role Bob(my_expo: int)
  begin
    declare
      alice_message: symkey;
    begin
      recv(alice_message);
      send(kap(g,my_expo));
    end;
  end

scenario
    parallel
      Alice(alice_expo) | Bob(bob_expo)
    end

end
```

---

The pertaining assertion file is:

```
uses
  "diffie-helman.prv"

public
  kap

initial
  x_M = [[ g ]]

always
  issecret(secret)
```

---

The ACTAS input generated by our translator is:

```
[Comment]

  Translation of assertion diffie-helman.pra
  Translation options:

[Signature]

  const: 0, true, false, sym, asym, rsa, des, g, secret, alice_expo, bob_expo
  var: alice_message, bob_message, my_expo, x, y, z
```

```
  AC: mult

[R-rule: TRS1]

  # Actions of Alice
  S_Alice(my_expo) -> kap(g,my_expo)
  RS_Alice(my_expo,bob_message) -> symcrypt(sym,kap(bob_message,my_expo),secret)

  # Actions of Bob
  RS_Bob(my_expo,alice_message) -> kap(g,my_expo)

  # Dolev-Yao simplification roles
  inverse(inverse(x)) -> x
  decrypt(x,y,crypt(x,inverse(y),z)) -> z
  decrypt(x,inverse(y),crypt(x,y,z)) -> z
  symdecrypt(x,y,symcrypt(x,y,z)) -> z

  # Rewriting rules corresponding to the equational axioms
  kap(kap(g,x),y) -> kap(g,mult(x,y))
  kap(g,mult(x,y)) -> kap(kap(g,x),y)

[T-rule(q_message): TAintruder]

  # initial intruder knowledge
  g -> q_symkey

  # function symbols known to the intruder
  inverse(q_pubkey) -> q_privkey
  inverse(q_privkey) -> q_pubkey
  inverse(q_symkey) -> q_symkey
  decrypt(q_algo,q_privkey,q_message) -> q_message
  crypt(q_algo,q_pubkey,q_message) -> q_message
  symdecrypt(q_symalgo,q_symkey,q_message) -> q_message
  symcrypt(q_symalgo,q_symkey,q_message) -> q_message
  s(q_int) -> q_int
  0 -> q_int
  true -> q_bool
  false -> q_bool
  sym -> q_symalgo
  des -> q_symalgo
  asym -> q_algo
  rsa -> q_algo
  kap(q_symkey,q_int) -> q_symkey

  # the type lattice
  q_int -> q_message
  q_symalgo -> q_message
```

```
  q_symkey -> q_message

  # triggering actions
  alice_expo -> q_aux_1
  S_Alice(q_aux_1) -> q_symkey
  alice_expo -> q_aux_2
  RS_Alice(q_aux_2,q_symkey) -> q_message
  bob_expo -> q_aux_3
  RS_Bob(q_aux_3,q_symkey) -> q_symkey

[T-rule(q_secret): TAsecret]

  secret -> q_secret
```

---

Running the ACTAS system on this file reveals that the term `secret` is recognized by the automaton obtained by the ?-fold iteration of calculating rewrite descendants, and hence finds the well-known flaw of this protocol. Here is a term which is bothe accepted by the automaton and rewritten by the rewrite system into `secret`:

| Subterm | State | Rewrites to modulo $AC(mult)$ |
|---|---|---|
| $symdecrypt($ | $q\_message$ | $secret$ |
| $sym,$ | $q\_symalgo$ | |
| $kap($ | $q\_symkey$ | $kap(g, mult(alice\_expo, s(0)))$ |
| $S\_Alice($ | $q\_symkey$ | $kap(g, alice\_expo)$ |
| $alice\_expo),$ | $q\_aux\_1$ | |
| $s($ | $q\_int$ | |
| $0)),$ | $q\_int$ | |
| $RS\_Alice($ | $q\_message$ | $symcrypt(sym, kap(g, mult(alice\_expo, s(0))), secret)$ |
| $alice\_expo,$ | $q\_aux\_2$ | |
| $kap($ | $q\_symkey$ | |
| $g,$ | $q\_symkey$ | |
| $s($ | $q\_int$ | |
| $0))))$ | $q\_int$ | |

## 5.2 Ping-Pong Protocols

The following example is the third protocol example presented in [DEK82]. There are several attacks against this protocol.

The protocol is as follows:

```
# ping-pong protocol 1: [Dolev,Even,Karp '82]
# X → Y     X, {M}Pk(Y)
# Y → X     Y, {M}Pk(X)

signature
  alice, bob, charley: principal;
  pubkey_of: principal → pubkey;
  secret: message;
```

```
end

role Alice(my_name, partner_name: principal)
begin
  send([my_name,crypt(asym,
        pubkey_of(partner_name),[my_name,
                  crypt(asym,pubkey_of(partner_name),secret)])]);
end

role Bob(my_name: principal)
begin
  declare
    m: message;
    partner_name: principal;
  begin
    recv([partner_name,crypt(asym,pubkey_of(my_name),[partner_name,
          crypt(asym,pubkey_of(my_name),m)])]);
    send([my_name,crypt(asym,pubkey_of(partner_name),m)]);
  end;
end

scenario
  begin
   parallel
      Alice(alice,bob)
    | Bob(bob)
    end;
  end
end
```

---

The pertaining assertion file is:

```
uses
  "pingpong3.prv"

public
  pubkey_of

initial
  x_M = [[ alice, bob, charley ]]

always
  issecret(secret)
```

---

The ACTAS input generated by the translator is:

```
[Comment]
```

```
  Translation of assertion pingpong3.pra
  Translation options:

[Signature]

  const: O, true, false, sym, asym, rsa, des, alice, bob, charley, secret
  var: m, my_name, partner_name, x, x0, x1, y, z

[R-rule: TRS1]

  # Actions of Alice
  S_Alice(my_name,partner_name) -> tuple_principal_message(my_name,crypt(asym,pubkey_of(partn

  # Actions of Bob
  RS_Bob(my_name,tuple_principal_message(partner_name,crypt(asym,pubkey_of(my_name),tuple_pri

  # Dolev-Yao simplification roles
  inverse(inverse(x)) -> x
  decrypt(x,y,crypt(x,inverse(y),z)) -> z
  decrypt(x,inverse(y),crypt(x,y,z)) -> z
  symdecrypt(x,y,symcrypt(x,y,z)) -> z
  pr_0(tuple_principal_message(x0,x1)) -> x0
  pr_1(tuple_principal_message(x0,x1)) -> x1

[T-rule(q_message): TAintruder]

  # initial intruder knowledge
  alice -> q_principal
  bob -> q_principal
  charley -> q_principal

  # function symbols known to the intruder
  inverse(q_pubkey) -> q_privkey
  inverse(q_privkey) -> q_pubkey
  inverse(q_symkey) -> q_symkey
  decrypt(q_algo,q_privkey,q_message) -> q_message
  crypt(q_algo,q_pubkey,q_message) -> q_message
  symdecrypt(q_symalgo,q_symkey,q_message) -> q_message
  symcrypt(q_symalgo,q_symkey,q_message) -> q_message
  s(q_int) -> q_int
  O -> q_int
  true -> q_bool
  false -> q_bool
  sym -> q_symalgo
  des -> q_symalgo
  asym -> q_algo
```

```
  rsa -> q_algo
  tuple_principal_message(q_principal,q_message) -> q_tuple_principal_message
  pr_0(q_tuple_principal_message) -> q_principal
  pr_1(q_tuple_principal_message) -> q_message
  pubkey_of(q_principal) -> q_pubkey

  # the type lattice
  q_algo -> q_message
  q_principal -> q_message
  q_pubkey -> q_message
  q_tuple_principal_message -> q_message

  # triggering actions
  alice -> q_aux_1
  bob -> q_aux_2
  S_Alice(q_aux_1,q_aux_2) -> q_tuple_principal_message
  bob -> q_aux_3
  RS_Bob(q_aux_3,q_tuple_principal_message) -> q_tuple_principal_message

[T-rule(q_secret): TAsecret]

  secret -> q_secret
```

## References


[DEK82] D. Dolev, S. Even, and R. M. Karp. On the security of ping-pong protocols. *Information and Control*, 55(1–3):57–68, October/November/December 1982.

[DH76] W. Diffie and M. Helman. New directions in cryptography. *IEEE Transactions on Information Society*, 22(6):644–654, november 1976.

[JRV00] Florent Jacquemard, Michaël Rusinowitch, and Laurent Vigneron. Compiling and verifying security protocols. In Michel Parigot and Andrei Voronkov, editors, *LPAR*, volume 1955 of *Lecture Notes in Computer Science*, pages 131–160, Reunion Island, France, November 2000. Springer-Verlag.

[KLT05] Steve Kremer, Yassine Lakhnech, and Ralf Treinen. The PROUVÉ manual: Specifications, semantics, and logics. Technical Report 7, projet RNTL PROUVÉ, December 2005. 49 pages.

[MD02] Jonathan K. Millen and Grit Denker. CAPSL and MuCAPSL. *Journal of Telecommunications and Information Technology*, 4:16–26, 2002.

[OT05] Hitoshi Ohsaki and Toshinori Takai. Actas : A system design for associative and commutative tree automata theory. *Electr. Notes Theor. Comput. Sci.*, 124(1):97–111, 2005.