RAPPORT TECHNIQUE EVA

The EVA translator, version 6

Auteur	:	Florent Jacquemard
Date	:	July 2, 2003
Rapport EVA numéro	:	9
Version	:	1.2

TRUSTED LOGIC S.A.
5 rue du Bailliage
78000 Versailles, France
www.trusted-logic.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

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 last syntactic and semantic changes to the EVA protocol specification language as well as the modifications of the EVA translator.

The EVA translator, version 6

Florent Jacquemard

July 2, 2003

In this report we describe the last syntactic and semantic changes to the EVA protocol specification language [3] as well as the modifications of the EVA translator.

Contents

1 Translator package

The purpose of the translator of the EVA project [3] is to compile a high level specification of a security protocol written in the LAEVA language into a model that can be handled by all the verification tools of the project.

LAEVA has been designed [4] as a common protocol specification language for all the partners (and developers of verification tools) of the EVA project and follows the main characteristics of the *idealized* descriptions of protocols found in the papers concerned with security protocol verification, starting from the BAN logic paper [1]. This description language focuses on the form of the messages exchanged during the protocol, rather than on the actions of the processes sending and receiving the messages (we call such a presentation *a la BAN* below). While this presentation has the advantage of conciseness and is generally sufficient for discussions, a specification at a lower level, with description of the processes, is required for formal automatic verification of protocols. We call such a low level specification a *multi-process* presentation of a protocol.

The low level language for compiled protocols specifications is CPL ([4]) a syntax a la LISP which permits to describe (non-ambiguously) several communicating processes representing the principal of a protocol. CPL's syntax and semantics are formally defined in [4].

1.1 Main changes to the previous version

The version 6 of the EVA translator offers, to summarize, the following main changes:

- the LAEVA language, also called *concrete syntax* or *input syntax*, has been modified following in particular remarks and demands of the developers of Hermes [6, 7] (Section 2),
- an *abstract syntax* is defined in this version as a collection of abstract data types, presented in a programming interface, and used to store both the parsed "a la BAN" and the compiled "multi process" forms of a protocol specification (Section 3). Note that in [4], the term abstract syntax is used for CPL; here, we call CPL (the modified version presented in Section 5.2) an *output syntax*. However, extracting a CPL program for the abstract syntax is just a matter of pretty printing.
- the semantics, defined in [4] for CPL is adapted here and defined for the *abstract syntax* in Section 4.

The purpose of the "abstraction" of the abstract syntax (compared to CPL) and of its presentation in a programming interface is to improve the usability of the translator and its code (see Figure 1). From the user point of view, a command line which takes as input a specification written in LAEVA and prints its translation in CPL is still possible (see Section 1.3, but other output formats can be added as options, and at low development cost (Section 5).

From the developer point of view, it is possible to call the translator from the code of e.g. a protocol verification tool, using the data types of the abstract syntax, (see Section 1.4 for a short description of the translator modules), and this can prevent from having to write a parser for CPL.

From now on, a protocol specification in abstract syntax is called an *abstract specification*.

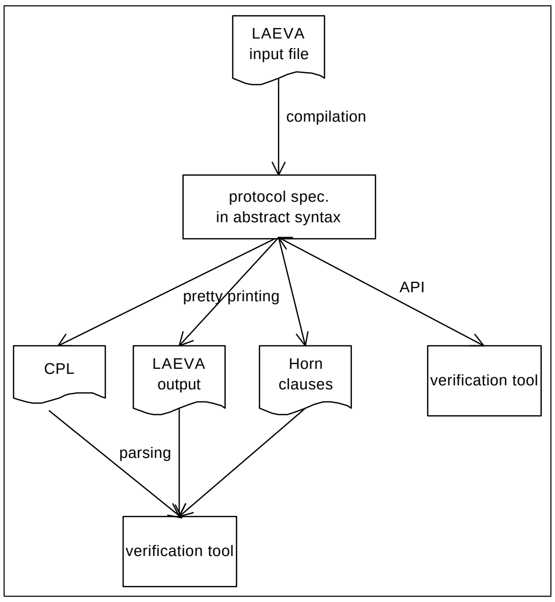

Figure 1: Architecture and usage of the EVA translator v.6

1.2 Installation

The package of the EVA translator contains:

- the documented source code of the translator and a makefile,
- the latex source of the translator documentation (in directory `doc`), and scripts to extract latex file from the some part of source code, like grammars (produce this document),
- some examples and test files in LAEVA.

To compile the translator and the documentations (refer to the file `README`) go to the directory where the translator source where unpacked, and type `make`. GNU make and OCAML v. 3.06 are required for the compilation

To produce the documentation (translator guide and html source documentation), in the same directory, type `make doc`. The guide, in postscript and pdf (`EVAtrans6.ps` and `EVAtrans6.pdf`) is left in directory `doc`, and the html files of the APIs in directory `html`. The guide and the APIs documentation can be produced separately by the respective commands `make rapport` and `make html`.

LaTeX2ε, dvips and dvipdfm are required in order to produces the translator's guide and ocamldoc (OCAML v. 3.06) is necessary for building the html documentation of the sources.

1.3 Command line

The compilation leaves an executable `evatrans` in the current directory[1] which can be called as follows:

[1]`evatrans` is actually a symbolic link to a command `eva2cpl`.

```
evatrans [options] input.eva
```

The options are the following:

- `-cpl` prints the compiled protocol in CPL syntax on the standard output
- `-eva` pretty-prints the compiled protocol in the LAEVA syntax (with additional syntax for the multi-process protocols) on the standard output
- `-horn` pretty-prints the compiled protocol as a set of horn clauses
- `-tptp` pretty-prints the compiled protocol as a set of horn clauses in TPTP format
- `-trace` verbose mode (prints debug information)
- `-help` prints an usage message and exit

The options `--eva` and `--cpl` are exclusive, only the last is taken into account. By default, the compiled protocol is not printed, and the translator only returns an error message or nothing if the compilation was successful.

1.4 Translator sources and API

We shall describe here briefly the organization of the sources of the translator and its main functionalities.

- a parser (module `evaparse`), generated by ocamlyacc, reads a protocol specification written in LAEVA (Section 2) and stores it in a structure of the abstract syntax (Section 3.3.1). Hence, parsing produces a new abstract specification.
- the different kinds of identifiers (e.g. parameters or aliases, see Section 3) are stored in different structures in the abstract syntax, whereas they can not be distinguished at parse time. Hence, an additional conversion (function `coerce_spec` of module `translator`) is necessary after parsing (see Section 3.3.2 for more details).
- the abstract syntax is a strongly typed language. The module `types` is for type checking. The type discipline of the LAEVA concrete syntax is slightly different from the one of the abstract syntax (see Section 4.1). Indeed, the tuples of elements of various types are allowed in LAEVA and not in the abstract syntax. Hence, during the compilation of an LAEVA specification in abstract syntax, like in previous versions [3], some coercion symbols (see Section 4.3.3) are added by the function `coerce_spec` (in module `translator`) to the terms to ensure their weel-typingness.
- some functions (in module `translator`) permit to check the non-syntactic restrictions on the LEVA language presented in Section 2.4.
- The function `translate` of the module `translator` converts an abstract specification of a protocol presented “a la BAN” (all the principal programs presented in one, see Section 3) into an abstract specification with a multi-process presentation of the protocol (list of separate programs, one for each principal). More details on this module are given in Section 3.3.5.

- several modules `*pp` are given for pretty printing an abstract specification into special formats (in particular CPL, see Section 5).

When they are chained in the above order (ending with the CPL pretty printer), these functions of this version of the translator, like in the other versions, print a CPL protocol specification which is the (multi-process) translation of a LAEVA specification given in an input file in LAEVA syntax.

1.5 Error messages

A protocol specification given as input to the translator must conform to the LAEVA syntax and to the additional restrictions described in Section 2.5.

Otherwise, the translator will fail with an error message. There is one specific error message for each of the restrictions (marked with a $\Rightarrow$) of Section 2.5.

1.6 Output syntaxes

Several pretty-printing formats have been implemented. The following ones are of particular interest:

- an extended LAEVA format (Section 5.1). This ouput format is proposed for information and debugging purposes (rather than for an input format for verification tools).
- a CPL format with the output syntax of Section 5.2,
- set of Horn clauses to be used by first order theorem provers.

The definition of the abstract syntax leaves many other possibilities, like for instance (conditional) rewrite systems etc.

However, the abstract syntax and the modular architecture of the translator have been designed to permit to the developers of tools in the EVA project to reuse the translators modules directly in their code, preventing them from the burden of parsing CPL files.

2 LAEVA Input Syntax

A protocol specifications in the LAEVA concrete syntax contains, to summarize:

- the declarations of the protocol identifiers,
- a description of the messages exchanges during an execution of the protocol,
- some other declarations more specific to the verification procedures:
  - the declarations of the identifiers representing values of domain of the protocol execution,
  - a description of the initial state of the system to be verified (number of sessions of the protocol in parallel, and initial local state of every principal in every session),
  - the hypothesizes (formulas) concerning the initial state of the of the hostile environment,
  - the security properties to verify.

2.1 Main changes to the previous version

The main syntactical changes to the LAEVA input language w.r.t. the previous version [3] are:

- the addition of a qualifier constant or parameter for the declaration of identifiers. See Section 4 for their respective meaning,
- extended syntax for the declaration of sessions,
- special and restricted form for the declaration of values which instantiate the initial states of principals in the sessions declarations,
- syntax of Horn clauses for the formulas, both for assume formulas describing the initial state of the hostile environment and for the claim formulas defining the security properties to verify.

2.2 Identifiers

ID An identifier is a non-empty sequence of letters ('a' to 'z' and 'A' to 'Z'), digits, and the underscore character '_', starting with a letter.

INT A integer literal is a non-empty sequence of digits.

LABEL A label is either a an identifier immediately followed by a dot character (ID.) or an integer literal immediately followed by a dot character (INT.).

2.3 Keywords

The following character sequences and identifiers are reserved:

(	)	{	}	[	]
,	;	:	=	^	*
_	->	=>	==	!=	:=
%	@	"			
alias	assume	asym_algo	axiom	basetype	case
claim	constant	everybody	false	fresh	hash
honest	keypair	knows	number	parameter	principal
secret	session	sym_algo	switch	value	

The characters ',' and ';' are right associative.

2.4 Extra restrictions

Some restrictions on the LAEVA language checked by the translator are not of a syntactical nature. They are marked with a $\Rightarrow$ in the LAEVA grammar given in Section 2.5.

If one of such conditions fails, the translator will stop with an appropriate error message on the standard output.

2.5 Grammar

2.5.1 Specification

An EVA specification, contains in this order:
a label (the protocol name)
the declaration of the specification identifiers, aliases, base types, principals knowledge, axioms,
a presentation a la BAN of the protocol messages,
the declaration of the identifiers used to construct the sessions values,
the declaration of sessions (principals initial state)
the formulas: assume formulas to define the initial state of the environment and claim formulas to be verified

spec ::=
  ID declarations block value_declarations sessions statements EOF

2.5.2 Type identifiers

type_id ::=
  ’principal’
| ’number’
| ’asym_algo’
| ’sym_algo’
| ID

2.5.3 Declarations

reversed list of declarations.
⇒ every symbol declared here must not be declared twice in the sections declarations and value_declarations

declarations ::=
  declarations declaration
| ’ ’

declaration

declaration ::=

typing_declaration	declaration of first order or functional identifiers
| ’alias’ ID ’=’ atomic_term_or_ciphertext	declaration of global alias ⇒ the ID must be distinct from ’I’ (intruder’s name)
| ids ’;’ ’alias’ ID ’=’ atomic_term_or_ciphertext	declaration of local alias ⇒ the ID must be distinct from ’I’ (intruder’s name)
| ’basetype’ ID	declaration of base type

⇒ the ID must be distinct from ’I’ (intruder’s name)

| ’everybody’ ’knows’ tuple — initial knowledge common to every principal

| ID ’knows’ tuple — initial knowledge of principal ID

⇒ ID must be declared as a first order identifier of type principal

| ’axiom’ atomic_term_or_ciphertext ’=’ atomic_term_or_ciphertext optional_quantification
global equational axiom

⇒ every leave in the terms must be an variable of the optional_quantification

⇒ both terms (members of the atom) must have the same type

quantification for the formulas (assume) and axiom

⇒ the symbols are bounded, they can be declared elsewhere in a declaration or value_declaration

⇒ every (variable) symbol must occur at most once in quantification

optional_quantification ::=

’[’ quantified_vars ’]’

| ’ ’

non-empty reversed list of universally quantified variables

quantified_vars ::=

quantified_var

| quantified_vars ’,’ quantified_var

universally quantified variable with optional type

⇒ the default type is number

quantified_var ::=

ID

| ID ’;’ type_id

type declaration for identifiers

typing_declaration ::=

scope ids_decl ’;’ type_id — declaration of first order identifiers

| scope ID ’(’ type_list ’)’ ’;’ type_id optional_hash optional_secret
declaration of functional identifier

⇒ the ID must be distinct from ’I’ (intruder’s name)

| scope ’keypair’ optional_encryption_algorithm ID ’,’ ID optional_type_list ’;’ type_id
declaration of a pair of asymmetric keys, both IDs can be first order or functional symbols.
note: no hash keyword, every keypair is assumed to be hash.

⇒ the two IDs (public and private keys) must be distinct
⇒ the two IDs must be distinct of 'I' (intruder's name)

non empty reversed list of identifiers

ids ::=
ID
| ids ',' ID

non-empty reversed list of identifiers for declaration

ids_decl ::=
id_decl
| ids_decl ',' id_decl

first order identifier declared with freshness

id_decl ::=

'fresh' ID	fresh first order identifier: ⇒ the ID must be distinct from 'I' (intruder's name) ⇒ the ID must be declared with the scope parameter
| ID	non fresh first order identifier: ⇒ the ID must be distinct from 'I' (intruder's name)

reversed list of types

type_list ::=
non_empty_type_list
| ' '

non-empty reversed list of types

non_empty_type_list ::=
type_id
| non_empty_type_list ',' type_id

list of types

optional_type_list ::=
'(' type_list ')'
| ' '

optional qualifier of one-way functions

optional_hash ::=
'hash'
| ' '

optional qualifier of secret functions

optional_secret ::=

’secret’

| ’ ’

quality of identifier

scope ::=

’constant’	global constant
| ’parameter’	session parameter

invocation of an encryption algorithm
⇒ must have an algorithm type

optional_encryption_algorithm ::=

’^’ atomic_term	the atomic_term describes a symmetric or asymmetric key algorithm
| ’ ’	default symmetric key algorithm

2.5.4 Protocol messages

protocol or sub-protocol

block ::=

’{’ messages ’}’

reversed list of instructions

messages ::=

messages message

| ’ ’

message label

label ::=

LABEL

protocol instruction
every term and atomic_term_or_ciphertext in the instruction:

- ⇒ can contain (at leaves positions) first order symbols, and functional symbols, and key symbols, declared either with scope constant or parameter
- ⇒ can contain declared alias symbols
- ⇒ must not contain located variables
- ⇒ must not contain identifiers declared as values
- ⇒ can contain ’%’ (Lowe’s notation)

message ::=

label ID ’->’ ID ’;’ term	sending/receiving message

⇒ the two ID's (sender and receiver) must be declared as first order identifiers with type principal

| ID ';' ID '==' atomic_term_or_ciphertext
equality test

⇒ the first ID must be declared as first order identifier with type principal

⇒ the second ID must not be known to the first ID (call it r), i.e. it must not have be assigned in the local state of r at this point.

| ID ';' ID ':=' atomic_term_or_ciphertext
local assignment

⇒ the first ID must be declared as first order identifier with type principal

⇒ the second ID must not be known to the first ID (call it r), i.e. it must not have be assigned in the local state of r at this point (reassignements are not allowed).

| block   sub-protocol

| 'switch' atomic_term_or_ciphertext '{' cases '}'
conditional

reversed list of cases

cases ::=

cases case

| ' '

conditional branching

case ::=
'case' atomic_term_or_ciphertext ';' block

2.5.5 Terms

term

term ::=

tuple

non-empty reversed list of terms

tuple ::=

atomic_term_or_ciphertext
| tuple ',' atomic_term_or_ciphertext

non-empty parenthesized reversed list of terms

non_empty_term_list ::=

’(’ tuple ’)’

parenthesized reversed list of terms

term_list ::=

’(’ ’)’

| non_empty_term_list

atomic term

atomic_term_or_ciphertext ::=

atomic_term

| ’{’ term ’}’ ’_’ atomic_term optional_encryption_algorithm
ciphertext

| ’[’ term ’]’ ’_’ atomic_term optional_encryption_algorithm
signature

| atomic_term_or_ciphertext ’%’ atomic_term_or_ciphertext
schizo-notation a la Lowe

⇒ ’%’ is not allowed inside the terms

| ID ’@’ ID
located term

⇒ for formulas claim only
⇒ the second ID must be declared as first order identifier with type principal
⇒ the first ID must be declared as first order or functional or key identifier
⇒ the type of the first ID only is considered in type evaluation

atomic term

atomic_term ::=

ID term_list
function call

⇒ ID must be declared as functional symbol or functional member of keypair
⇒ the length of the term_list must match the declared signature of ID (use parenthezing tuples to apply function to more arguments).
⇒ the types of terms in the term_list must match the declared signature of ID

| ID
identifier

⇒ ID must be declared as first order symbol or value

| non_empty_term_list
singleton list for optional parenthezing of tuples

2.5.6 Declaration of values (domain constructors)

reversed list of value declarations for instantiating the sessions
⇒ the value identifiers must not be declared otherwise

value_declarations ::=
 value_declarations value_declaration

| ' '

non-empty reversed list of values identifiers

values ::=
 ID
| values ',' ID

declaration of the symbols used to construct the terms of interpretation domain,
they can be used below to instanciate the sessions declared

value_declaration ::=

'value' values ';' type_id	declaration of nullary value symbols
| typing_declaration	declaration of constructor for values ⇒ it must be a declaration of functional or key identifier (not first order id) ⇒ the scope of declaration must be constant

2.5.7 Sessions

reversed list of parallel sessions
(initial state of the principals)

sessions ::=
 sessions session
| ' '

Session definition

session ::=

'session' '*'	arbitrary number of sessions over arbitrary domains
| 'session' '*' '[' session_constraints ']'	arbitrary number of sessions with domain satisfying a constraint
| 'session' '*' '(' session_assignments ')'	arbitrary number of copies of the given session
| 'session' '(' session_assignments ')'	fixed session
| label 'session' '(' session_assignments ')'	fixed session with label

non empty reversed list of constraints on domains for instantiating sessions

session_constraints ::=

session_constraint
| session_constraints ';' session_constraint

constraint on domains for session instances

session_constraint ::=

ID '==' ID	equality constaint ⇒ both IDs must be declared with scope parameter
| ID '!=' ID	disequality constaint ⇒ both IDs must be declared with scope parameter
| ID ':' '{' tuple '}'	membership constraint: the parameter ID belongs to a finite set of values. ⇒ ID must be declared with scope parameter ⇒ the tuple is a sequence of terms can contain only at leaves positions: values symbols, first order, functional and key symbols declared with scope constant.
| ID ':' predicate	domain constraint: the first ID is in the model of the predicate. ⇒ ID must be declared with scope parameter

predicate symbol for membership constraints in session declarations

predicate ::=

'secret'	reserved unary predicate "secret"
| 'honest'	reserved unary predicate "honest"
| ID	user predicate ⇒ ID, the name of the predicate, must not be a declared identifer. it can only occur as a predicate symbol in an atom of a statement.

non-empty list of assignments for the parameters of a session

session_assignments ::=

session_assignment
| session_assignments ';' session_assignment

instantiation of a parameter for the definition of a session

session_assignment ::=

ID '=' atomic_term_or_ciphertext
⇒ the ID must have been declared as a parameter

⇒ the atomic_term_or_ciphertext can contain (at leaves positions) either: value symbols, first order, functional and key symbols declared with scope constant, 'I', the name of the intruder, or aliases to terms of the above form

⇒ the ID and atomic_term_or_ciphertext must have the same type

2.5.8 Formulas

reversed list of hypotheses and properties

statements ::=
  statements assumption
| statements claim
| ' '

hypothesis on the initial state of the environment

assumption ::=
  'assume' formula optional_quantification

property to prove

claim ::=
  'claim' formula

Horn clause

formula ::=
  optional_label atom
| optional_label atoms '=>' 'false'
| optional_label atoms '=>' atom

optional formula name

optional_label ::=
  label
| ' '

reversed list of atoms

atoms ::=
  non_empty_atom_list
| ' '

non-empty reversed list of atoms

non_empty_atom_list ::=

atom
| non_empty_atom_list ',' atom

atomic proposition.

Every term in the folloing atoms must be such that:

- ⇒ in assume: every leave in the term must be a protocol constant, a declared value, or a quantified variable.
- ⇒ in claim: every leave in the term must be a protocol constant, a declared value, or a located variable
- ⇒ in both: the term must not contain '%' (Lowe's notation)

atom ::=
atomic_term_or_ciphertext '==' atomic_term_or_ciphertext
term equality
| atomic_term_or_ciphertext '!=' atomic_term_or_ciphertext
term disequality
| 'secret' '(' atomic_term_or_ciphertext ')'
the intruder ignores term
| 'honest' '(' atomic_term_or_ciphertext ')'
the term is the identifier of an honest principal.
| ID '(' tuple ')' user defined predicate

- ⇒ ID, the name of the predicate, must not be an identifer declared elsewhere
- ⇒ there must be exactly one argument.

3 Abstract syntax

The *abstract syntax* is a collection of abstract data types used to store a protocol specification. It is defined exhaustively in Section 3.2.

3.1 a la BAN and multi-process specifications

The structures used to store the declarations (dcl in Section 3.2), the sessions (session) and the hypotheses and claims (statement) mimic the corresponding definitions of the concrete syntax LAEVA described in section 2.

Concerning the protocol instructions and messages (type protocol) there are two alternatives in the abstract syntax:

1 the first option is the same as in the concrete syntax, i.e. a presentation called *a la BAN* of all the programs of the different principals in a single list of instructions and messages.

2 the second option, called *multi-process protocol* is a list of programs, one for each principal. Every program contains a list of instructions and of send or received messages. The received messages are patterns which may contain fresh variables which are not declared in the specification (they correspond to cipher or hashed text that a receiver cannot read). These variables (called *private*) are declared in the program.

Note that a protocol specification in LAEVA syntax (Section 2) can be straightforwardly stored in a type spec, with a component protocol of the kind 1 (a la BAN) above. The compilation procedure of the translator consists in converting this a la BAN abstract specification into a multi-process abstract specification (see Section 3.3).

3.2 Data types

We give below the complete description of the data types of the abstract syntax. The types ID and LABEL are defined in Section 2.2.
Top level container for a protocol specification in EVA syntax.

spec =

spec(ID, (dcl list), protocol, (dcl list), (session list), (statement list))

contents:

- protocol label
- the declarations of the spec
- the messages of the spec
- declarations of values (for the session domains) in the spec
- the session defined in the spec
- the hypotheses and formulas in the spec

The label of messages, claims or sessions.

label =

nolabel	empty label
| label(LABEL)	other label

EVA types

type =

void	NULL type
| principal	predefined type "principal"
| number	predefined type "number"
| aalgo	predefined type for asymmetric encryption algo
| salgo	predefined type for symmetric encryption algo
| talgo	union of the two above
| usertype(ID)	type of the specification = subtype of "number"
| basetype(ID)	user type of the specification declared as basetype

quality of identifiers

scope =

cst	global constant
| param	session parameter
| private	local variable in some principal 's program

value to be assigned to a identifier in a session declaration

value =

intruder	special value for the intruder
| value(ID)	arbitrary value

term =

term_id(ID)	identifier declared as parameter or private variable, can be a first order symbol or a function symbol or a keypair symbol
| term_cst(ID)	identifier declared as constant, can be a first order symbol or a function symbol or a keypair symbol
| term_value(value)	first order identifier, declared as value constant
| quantified(ID, type)	quantified variable, only for axioms and assume formulas
| located(ID, ID)	located variable var@role, only for claim formulas – the first ID is a parameter – the second ID is a role
| term_alias(ID, term)	alias symbol – name of the identifier – type declared in quantification (default is number)
| cons(term list)	tuple of terms
| app(ID, (term list))	application of function symbol – root symbol – arguments
| crypt(algo, term, term)	cipher text – algo – key – contents
| sign(algo, term, term)	signature – algo – key – contents
| p(term)	coercion from principal to number
| a(term)	coercion from talgo to number
| sa(term)	coercion from salgo to talgo
| aa(term)	coercion from aalgo to talgo
| u(ID, term)	coercion from user type τ to number
| pcent(term, term)	Lowe's notation

name of an encryption algorithm

algo =

vanilla	generic symmetric key algorithm
| algo(term)	other algorithm, the term is restricted to be a constant

description of the initial knowledge of a principal

knowledge =

know_id(ID)	first order or functional identifier
| know_term(term)	well formed term

declarations in the EVA protocol specification

dcl =

dcl_id((ID, bool) list, scope, type)	declaration of first order identifiers – list of pairs of (identifier, flag) where the flag is true iff the identifier is declared to be fresh – scope common to all the identifiers of the list – type common all the identifiers of the list
| dcl_fun(ID, scope, type, (type list), bool, bool)	declaration of functional identifier – name of the function symbol – scope of the identifier – domain type – list of the respective types of the arguments of the function symbol – flag true iff the function symbol is declared to be hash (one-way) – flag true iff the function symbol is declared to be secret
| dcl_keypair(algo, ID, ID, scope, type, (type list))	declaration of a pair of asymmetric keys – associated encryption algorithm – name of first key – name of second key – scope of both keys – (domain) type of both keys – list of the respective types of the arguments of both keys (can be empty)
| dcl_alias(ID, term)	declaration of an alias – name of the alias – term for replacement
| dcl_localalias((ID list), ID, term)	declaration of local alias
| dcl_basetype(ID)	declaration of user base type, specific to verifying tool

| dcl_know(ID, (knowledge list))
declaration of initial knowledge of a principal,
for initialization of the components of its local state

- principal name
- list of initial knowledge

| dcl_every(term list) declaration of local state of every principal

| dcl_axiom(term, term, ((ID, type) list))
equational axiom

- left term
- right term
- quantified variables with types

| dcl_value((value list), type)
values (for session instantiation)

- list of value identifiers declared
- type common all the identifiers of the list

protocol instruction

instr =

skip	do nothing
| msg(label, ID, ID, term)	protocol message – label – sender – receiver – body
| assign(ID, ID, term)	assignment of a principal's local variable – role – variable – value
| comp(ID, ID, term)	comparison between two local variables in a principal's state – role – left variable – right variable
| block(instr list)	block of instructions
| switch(term, ((term, (instr list)) list))	switch case branching

specification of the messages of the protocol

protocol =

mp(instr list)	presentation a la BAN of all the programs in one block
| program(ID, (ID list), (instr list)) list	list of programs presented separately. For each program:

- the principal to which the program is associated
- list of variables (in messages) private to the program
- list of messages, which contain protocol constants and parameters and program variables

constraints on the session instances

constraint =

eq_constraint(ID, ID)	id == id
| neq_constraint(ID, ID)	id != id
| membership_constraint(ID, term list)	id in val list
| domain_constraint(ID,ID)	first id in the interpration domain of the predicate (second id)

declaration of a system assignment for verification

session =

BANG
| constrained_sessions(constraint list)

| sessions((ID, value) list)	copies of a single session - association list (var = value)
| session(label, ((ID, value) list))	single session – label – association list (var = value)

atom =

true

| false

| eq(term, term)

| neq(term, term)

| honest(term)

| secret(term)

| user_predicate(ID, term)

statement =

assume(((ID, type) list), (atom list), atom)	hypothesis – quantification – formula – tail of Horn clause – head of Horn clause
| claim(label, (atom list), atom)	formula to verify – optional label – tail of Horn clause – head of Horn clause

3.3 The translation procedure

We summarize here the role w.r.t. abstract specifications of the main functions of the translator, which are presented in Section 1.4.

3.3.1 Parsing

The *parsing* stores a protocol specification given in a file in LAEVA syntax into a new element of type spec of abstract specification. The third component of the spec, of type protocol, has the form mp(instr list) (aka presentation a la BAN).

During parsing, some decorations are added to the data types of the abstract syntax (Section 3.2); they contain location information (in the original file containing the LAEVA spec) and are used for the output error messages to the user.

3.3.2 Conversions

After parsing, every identifier i occurring in a term is stored in a $\mathsf{term\_id}(i)$, whatever its declaration. The translator converts these subterms, according to the declaration of the identifier i, as described in the following table:

	declaration of i	conversion
parameter	$\mathsf{dcl\_id}([\ldots,(i,b),\ldots],\mathsf{parameter},\tau)$	$\mathsf{term\_id}(i)$
constant symbol	$\mathsf{dcl\_id}([\ldots,(i,b),\ldots],\mathsf{constant},\tau)$	$\mathsf{term\_cst}(i)$
value constant symbol	$\mathsf{dcl\_value}([\ldots,i,\ldots],\tau)$	$\mathsf{term\_value}(\mathsf{value}(i))$
intruder symbol	I	$\mathsf{term\_value}(\mathsf{value}(\mathsf{intruder}))$
alias	$\mathsf{dcl\_alias}(i,t)$	$\mathsf{term\_alias}(i,t)$
quantification in axiom	$\mathsf{dcl\_axiom}(t_1,t_2,[\ldots,(i,type),\ldots])$	$\mathsf{quantified}(i,type)$
quantification in formula	$\mathsf{assume}([\ldots,(i,type),\ldots],\ldots)$	$\mathsf{quantified}(i,type)$
user type	type occ. in a dcl	ERROR

3.3.3 Coercion symbols

As outlined in Section 4.1.3, the type discipline for lists of terms, in terms of the form $\mathsf{cons}(\ldots)$ and $\mathsf{app}(\ldots)$, is not the same in the EVA concrete syntax and in the abstract syntax, and the translator adds some coercions symbols to cast the terms of these lists to the type number.

While adding coercion symbols, the translator checks whether in the terms of the form $\mathsf{app}(f,(\mathsf{term\ list}))$, the types of the arguments of the term list conform to the signature in the declaration $\mathsf{dcl\_fun}(f,\ldots)$ of the symbol f.

3.3.4 Basic verifications, typing

After this, the translator performs some additional tests on the abstract specification obtained, including the conformity to the extra restrictions described Section 2.5 (marked with $\Rightarrow$) and typing, following the definition of Section 4.1.

3.3.5 Compilation

Then, compilation consists in converting the protocol of the spec from the form mp(instr list) (presentation “a la BAN” of the messages) into a list of $\mathsf{program}(r,[x_1,\ldots,x_n],[\mathsf{instr}_1,\ldots,\mathsf{instr}_n])$

(multi-process presentation), where r is a role (as defined in Section 4.2.2), $x_1,\ldots,x_n$ are the private variables of the program.

The main functions of the *translation* module are similar to the function compose and expect of CASRUL [2].

Moreover, the Lowe's constructors pcent are eliminated. In every program (of the protocol) a $\mathsf{pcent}(m_1, m_2)$ in a message sent is replaced by m_1 and a $\mathsf{pcent}(m_1, m_2)$ in a message received is replaced by m_2.

3.3.6 Printing

The translator can then dump the abstract specification obtained in a required abstract syntax.

3.4 Restrictions

As explained above, a protocol specifications in LAEVA syntax given as input to the translator, must conform to the restrictions described in Section 2.4.

After translation, the abstract multi-process specification obtained fulfills the same restrictions (translated from concrete to abstract syntax) and additional ones:

- all the restrictions described in Section 2.5 for the terms in a protocol a la BAN are still valid for the terms in a multi-process protocol,

- in the terms of the form $\mathsf{cons}(\mathsf{term\ list})$ and $\mathsf{app}(f, \mathsf{term\ list})$, all the components of the term lists must have the type number (as defined in Section 4.1). Hence, the declared signature of the function symbols is obsolete in the abstract syntax, but it is checked at compilation (see Section 3.3.3).

- the terms in a multi-process protocol contain no constructor pcent.

These conditions can be assumed safely by each program which uses an abstract specification produced by the translator functions.

4 Semantics

We propose in this section an adaptation of the semantics defined in [3] to the new version of the abstract syntax. This operational semantics is defined by a infinite states / transitions model for a given protocol w.r.t. declared values.

We assume given a protocol specification in abstract syntax in multi-process form (see Section 3), conforming to the restrictions of Section 3.4.

We shall define first a type discipline (Section 4.1) and a domain of interpretation for the specified protocol (Section 4.3.4), as a multi-sorted term algebra defined essentialy with the function symbols presented in Sections 3.2. The states and transitions of our model are presented in the respective Sections 4.4 and 4.5. Finally, in Section 4.6 we define the satisfiability of a claim formula in a given state.

4.1 Types

We shall define here a type system for an interpretation of the protocol specified.

4.1.1 Reserved types

The types principal, number, sym_algo, asym_algo and algo are predefined in the abstract syntax.

4.1.2 User and base types

The other types occurring in the protocol specification (type 'type_id' in Section 3.2) are called *user types*. Every user type is a subtype of number. The *base types* are the user types declared with the special declaration dcl_basetype.

4.1.3 Types of lists

The type of lists (tuples) of numbers is number. No other tuples are allowed in the abstract syntax. Note though that polymorphic tuples are allowed in the LAEVA syntax. The translator adds some coercions functions symbols (given in Section 3.2) in order to transform these tuples into tuples of numbers (see also Sections 1.4, 3.3.4 and 4.3.6).

4.1.4 Types of functions

We assume that every function symbol f declared in the protocol abstract specification with a declaration (see Section 3.2):

$$\mathsf{dcl\_fun}(f, scope, \tau, type\_list, h\_flag, s\_flag)$$

where τ is a type of Sections 4.1.1 and 4.1.2, takes a single argument which is a tuple of numbers. Hence, the function f has signature $\mathsf{number} \to \tau$. This type is abbreviated by τ^{number}, when f is declared has not one-way and not secret ($h\_flag = s\_flag = false$). The types τ^{hnumber}, τ^{snumber}, τ^{hsnumber} are for functions declared respectively as one-way and not secret ($h\_flag = true$, $s\_flag = false$), not one-way and secret ($h\_flag = false$, $s\_flag = true$), one-way and secret ($h\_flag = s\_flag = true$). The distinction between these types is used below to define intruder decomposition rules (Section 4.5.2).

4.1.5 Types of keys

We define another type denoted τ^{kp} for the two function symbols f_1, f_2 declared with type τ in a key pair declaration of the protocol abstract specification:

$$\mathsf{dcl\_keypair}(algo, f_1, f_2, scope, \tau, type\_list)$$

The role of this type is explained in Section 4.3.2 below.

4.2 Protocol variables

Let $\mathcal{X}$ be the set of protocol variables defined in Sections 4.2.1 and 4.2.1 and 4.2.4 below.

4.2.1 Variables identifiers

A variable symbol is associated to each identifier x, f or k, k' declared in the abstract protocol specification with one of the following declarations (see Section 3.2):

$$\begin{array}{l}\mathsf{dcl\_id}((\ldots,(x, fresh\_flag),\ldots), scope, \tau_x)\\ \mathsf{dcl\_fun}(f, scope, \tau_f, type\_list, h\_flag, s\_flag)\\ \mathsf{dcl\_keypair}(algo, k, k', scope, \tau_k, type\_list)\end{array}$$

where $scope \neq \mathsf{CST}$. Indeed, see Section 4.3.1, the identifiers of scope cst are considered as values of the interpretation domain. The types of the above protocol variables are τ_x for x, τ_f^{number}, $\tau_f^{\mathsf{hnumber}}$, $\tau_f^{\mathsf{snumber}}$ or $\tau_f^{\mathsf{hsnumber}}$ for f, according to the respective values of *h_flag* and *s_flag* (see section 4.1.4), τ_k^{kp} for k, k' (see section 4.1.5).

A variable symbol is also associated to each of the *private* variables $x_1,\ldots,x_n$ in a program $\mathsf{program}(r,[x_1,\ldots,x_n],[\mathsf{instr}_1,\ldots,\mathsf{instr}_n])$ of the protocol part of the spec. The type of every private variable is number.

4.2.2 Roles

Let us consider the following declaration of a protocol in an abstract specification (see Section 3.2):

$$\big[\mathsf{program}(r_1,[x_{1,1},\ldots],[instr_{1,1},\ldots]),\ldots,\mathsf{program}(r_n,[x_{n,1},\ldots],[instr_{n,1},\ldots])\big]$$

Each identifier of $r_1,\ldots,r_n$ is called a *role* of the protocol. Each element $(r_j,[x_{j,1},\ldots],[instr_{j,1},\ldots])$ $(1 \leq j \leq n)$ is called the *program* of the role r_j (it is unique thanks to a restriction of Section 3.4).

4.2.3 Located variables

For each protocol variable x of Section 4.2.1, and each role r of Section 4.2.2, we consider a unique new protocol variable denoted '*x@r*'.

4.2.4 Fresh variables

Finally we assume a infinite set $\mathcal{X}$*_fresh* or fresh variables disjoint from the variables of Section 4.2.1 and 4.2.1.

4.2.5 Aliases

The alias symbols are not protocol variables.

An alias symbol a occurring in a subterm is understood as its syntactic replacement by the aliased term (as defined in the corresponding declaration $\mathsf{dcl\_alias}(a,t)$). More precisely, a the subterm $\mathsf{term\_alias}(a,t)$ in an abstract specification is interpreted as t (see Section 4.3.6).

4.3 Interpretation domain

The execution domain in our operational semantics is defined as a set of ground terms built with the following function symbols. Almost all these symbols are declared in the abstract specification, except two constructors for lists and coercions symbols.

4.3.1 Declared constants function symbols

We associate a set $\mathcal{C}$ of constants (nullary function symbols) to the abstract specification.

For each predefined or user type τ (see Sections 4.1.1 and 4.1.2), let $\mathit{values}(\tau)$ be the set of identifiers i declared as values of type τ in the abstract specification, with a declaration (the fourth component of protocol specification spec is reserved for values declarations, see Section 3.2):

$$\mathsf{dcl\_value}((\ldots, i, \ldots), \tau)$$

Note that dcl_value allow only to declare first order identifiers of a predefined or user type (Sections 4.1.1 and 4.1.2), but no functions, or keys symbols (Sections 4.1.4,4.1.5). Every constant of $\mathit{values}(\tau)$ has type τ.

Moreover, for each type τ, let $\mathit{constants}(\tau)$ be the set of identifiers i (and i') declared with type τ by one of the declarations (kind dcl in Section 3.2):

$$\begin{array}{l} \mathsf{dcl\_id}((\ldots, (i, \mathit{fresh\_flag}), \ldots), \mathsf{cst}, \tau) \\ \mathsf{dcl\_fun}(f, \mathsf{cst}, \tau, \mathit{type\_list}, \mathit{h\_flag}, \mathit{s\_flag}) \\ \mathsf{dcl\_keypair}(\mathit{algo}, i, i', \mathsf{cst}, \tau, \mathit{type\_list}) \end{array}$$

(the scope component must be equal to cst).

Note that the restrictions (Section 3.4) on the specification ensure that the sets $\mathit{values}(\tau)$ and $\mathit{constants}(\tau)$ are pairwise disjoint. The constants of $\mathit{constants}(\tau)$ may have type τ (dcl_id) or τ^{number}, τ^{hnumber}, τ^{snumber}, τ^{hsnumber} (dcl_fun, see section 4.1.4) or τ^{kp} (dcl_keypair, section 4.1.5).

The set $\mathcal{C}$ of constants associated to the abstract specification is the union of the above sets, plus a reserved constant I, of type principal, for the intruder's name:

$$\mathcal{C} := \biguplus_{\tau} \mathit{values}(\tau) \uplus \mathit{constants}(\tau) \uplus \{\mathsf{I}\}$$

4.3.2 Other function symbols

We have the following other function symbols:

empty_list, nullary, of type number

cons, binary, of type number $\rightarrow$ number $\rightarrow$ number

one app_τ, binary, of type $\tau^{\mathsf{number}} \rightarrow$ number $\rightarrow \tau$, for each predefined or user type (of sections 4.1.1 and 4.1.2). Similarly, we have happ_τ of type $\tau^{\mathsf{hnumber}} \rightarrow$ number $\rightarrow \tau$, sapp_τ of type $\tau^{\mathsf{snumber}} \rightarrow$ number $\rightarrow \tau$, hsapp_τ of type $\tau^{\mathsf{hsnumber}} \rightarrow$ number $\rightarrow \tau$,

one appkey_τ, binary, of type $\tau^{\mathsf{kp}} \rightarrow$ number $\rightarrow \tau$,

crypt, ternary, of type algo $\rightarrow$ number $\rightarrow$ number $\rightarrow$ number. The first argument is the encryption algorithm, the second is the encryption key (note that it can be an arbitrary term) and the third is the plain text to be encrypted.

For sake of simplicity, all the symbols app_τ, happ_τ, sapp_τ, hsapp_τ, appkey_τ are denoted app below.

4.3.3 Coercions symbols

In addition we have some unary symbols to perform types coercions (see remark in Section 4.1.3):

p of type principal $\longrightarrow$ number,

a of type algo $\longrightarrow$ number,

sa of type sym_algo $\longrightarrow$ algo,

aa of type asym_algo $\longrightarrow$ algo,

u_τ of type $\tau \longrightarrow$ number for each user type τ of the abstract specification.

4.3.4 Domain

Lets call $\mathcal{F}$ the set of the above function symbols (of Sections 4.3.1, 4.3.2, 4.3.3). The domain of interpretation of the given protocol (abstract specification), w.r.t. to the given set of values (see Section 4.3.1) is the set $\mathcal{T}(\mathcal{F})$ of well typed ground terms build on the above signature.

We shall also consider below the terms of $\mathcal{T}(\mathcal{F},\mathcal{X})$ with containing variables of Section 4.2.1.

4.3.5 Axioms

The domain is considered modulo the axioms declared with dcl_axiom (see Section 3.2),

Note that the coercions functions are added by the translator to the terms of the axioms too.

4.3.6 Terms of domain and abstract syntax

The translation between *terms* of the abstract syntax (i.e. elements of the data type term in Section 3.2) and terms of the above domain$\mathcal{T}(\mathcal{F},\mathcal{X})$ is straithforward. One only need to encode lists of terms by terms of the domain using the constructors empty_list and cons, and interprete located variables and aliases as expected.

More precisally, this coding is performed by the following recursive interpretation

$\bar{t}$ of an abstract term t into a term of $\mathcal{T}(\mathcal{F},\mathcal{X})$:

$$
\begin{array}{rcl}
\overline{\mathsf{term\_id}(x)} & := & x \in \mathcal{X} \\
\overline{\mathsf{term\_cst}(c)} & := & c \in \mathcal{C} \\
\overline{\mathsf{term\_value}(\mathsf{intruder})} & := & \mathsf{I} \in \mathcal{C} \\
\overline{\mathsf{term\_value}(\mathsf{value}(v))} & := & v \in \mathcal{C} \\
\overline{\mathsf{quantified}(x,t)} & := & x_{fresh} \in \mathcal{X}_{fresh} \\
\overline{\mathsf{located}(x,r)} & := & x@r \in \mathcal{X} \\
\overline{\mathsf{alias}(x,t)} & := & \bar{t} \\
\overline{\mathsf{cons}(t_1,\ldots,t_n)} & := & \mathsf{cons}(\overline{t_1},\mathsf{cons}(\ldots\mathsf{cons}(\overline{t_n},\mathsf{empty\_list}))) \\
\overline{\mathsf{app}(f,(t_1,\ldots,t_n))} & := & \mathsf{app}(f,\mathsf{cons}(\overline{t_1},\mathsf{cons}(\ldots\mathsf{cons}(\overline{t_n},\mathsf{empty\_list})))) \\
\overline{\mathsf{crypt}(a,k,t)} & := & \mathsf{crypt}(\bar{a},\bar{k},\bar{t}) \\
\overline{\mathsf{sign}(a,k,t)} & := & \mathsf{crypt}(\bar{a},\bar{k},\bar{t}) \\
\overline{\mathsf{p}(t)} & := & \mathsf{p}(\bar{t}) \\
\overline{\mathsf{a}(t)} & := & \mathsf{a}(\bar{t}) \\
\overline{\mathsf{sa}(t)} & := & \mathsf{sa}(\bar{t}) \\
\overline{\mathsf{aa}(t)} & := & \mathsf{aa}(\bar{t}) \\
\overline{\mathsf{u}_\tau(t)} & := & \mathsf{u}_\tau(\bar{t})
\end{array}
$$

Note that there is no interpretation of the terms of the form $\mathsf{pcent}(t_1,t_2)$ since it is assume (Section 3.4) that there is no such terms in the abstract specifications.

4.4 States

4.4.1 Substitutions

Due to some syntactic constructions in LAEVA, we shall use a special notion of substitutions in the definition of states below. Indeed, we call a substitution a mapping from terms of $\mathcal{T}(\mathcal{F},\mathcal{X})$ to terms of $\mathcal{T}(\mathcal{F},\mathcal{X})$ (and not only from variables of $\mathcal{X}$ to terms). The reason is that terms are allowed in the declarations dcl_know and that these declarations are used in the construction of initial states (see Section 4.4.6). The definition domain of a substitution σ is denoted $dom(\sigma)$. The application $t\sigma$ of a substitution σ to a term $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ is recursively defined by:

- $t\sigma := \sigma(t)$ if $t \in dom(\sigma)$,
- $f(t_1,\ldots,t_n)\sigma := f(t_1\sigma,\ldots,t_n\sigma)$ otherwise.

Moreover, we consider only substitution which respect types, in the sens that t and $t\sigma$ must have the same type, according to the definitions of Sections 4.1, 4.2 and 4.3.

4.4.2 Processes

A *process* is a tuple made of:

- a unique session identifier, which is a LABEL in the sense of the LAEVA syntax (see Section 2),
- a role,
- a program counter, which is an index in the list of instructions ((instr list) in Section 3.2) of the program associated to the role in the abstract specification,

- a substitution (well typed) from a finite subset of $\mathcal{T}(\mathcal{F},\mathcal{X})$ into the protocol interpretation domain $\mathcal{T}(\mathcal{F})$ (as defined in Section 4.3).

4.4.3 Predicates

The *predicates* of the protocol are the predicates identifiers occuring in the formulas (aka statement) of the abstract specification. They can be either eq, neq, honest, secret, or a user predicate symbol p occurring in an atom of the form $\mathsf{user\_predicate}(p,t)$.

4.4.4 States

A protocol *state* is a pair $(\mathcal{S}, I)$ where:

- $\mathcal{S}$ is a set of processes,
- I is an interpretation of the predicates of the protocol: to each predicate p, it associates a subset of the interpretation domain $p^I \subseteq \mathcal{T}(\mathcal{F})$.

4.4.5 Initial state: environment

We define the second part of an initial state as an interpretation I_0 which satisfies every assume formula of the abstract specification. Such an interpretation defines in particular a set of honest principals I_0^{honest}, and a set of data initially known to the intruder $\mathcal{T}(\mathcal{F}) \setminus I_0^{\mathsf{secret}}$.

In order to fix a unique I_0 from a given abstract specification, we make the following assumptions on the assume formulas:

- every equality atom has the form $\mathsf{eq}(x,y)$ or $\mathsf{eq}(x,a)$ or $\mathsf{eq}(x, f(y,z))$ where x, y, z are protocol variables and a, f are function symbols of $\mathcal{F}$ (respectively nullary and binary),
- every disequality atom has the form $\mathsf{neq}(x,t)$ where x is a protocol variable and t is a ground term.

With these restrictions, the set $\mathcal{A}$ of assume formulas can be transformed (by elimination of $\mathsf{eq}()$ and $\mathsf{neq}()$) into a equivalent set $\mathcal{A}'$ of Horn clauses all the atom of which are build with unary predicates. Hence, I_o is defined as the smallest Herbrand model of $\mathcal{A}'$.

4.4.6 Initial state: processes

Let $\mathcal{P}$ be a fixed infinite set of processes such that:

- for all $(i,r,pc,\sigma) \in \mathcal{P}$, r is a role of the protocol, pc is 0 (the index of the first instruction (instr in Section 3.2) in the program of r), the substitution σ takes its values in the protocol interpretation domain $\mathcal{T}(\mathcal{F})$ and its definition domain is exactly the set of terms of the list of terms ℓ in the initial knowledge declaration for the role r, $\mathsf{dcl\_know}(r,\ell)$, in the abstract specification. According to the restrictions on the specification (Section 3.4), there is at most one such declaration. If there is no such declaration, then the domain of σ is empty.
- $\mathcal{P}$ contains an infinite number of processes (i,r,pc,σ) (with different session identifiers) for each triple (r,pc,σ) as above,

- every two distinct processes $(i_1, r_1, pc_1, \sigma_1), (i_2, r_2, pc_2, \sigma_2) \in \mathcal{P}$ have different session identifiers ($i_1 \neq i_2$).
- freshness constraint: given any two distinct processes of $\mathcal{P}$, of respective substitutions σ_1 and σ_2, and for each protocol variable x declared as *fresh* in the specification, with a declaration of the form: $\mathsf{dcl\_id}((\ldots,(x,\mathsf{true}),\ldots),\mathsf{parameter},\tau)$, if $x \in dom(\sigma_1) \cap dom(\sigma_2)$, then $x\sigma_1 \neq x\sigma_2$ (the "nonce generator" is collision free).

To each declaration d of kind session in the abstract specification (see Section 3.2) we associate a set of processes $proc(d)$, defined as follows:

$$
\begin{array}{rcl}
proc(\mathsf{bang}) & := & \mathcal{P} \\
proc(\mathsf{constrained\_sessions}(\ell)) & := & \bigcap_{c\in\ell} constrained(c) \\
constrained(\mathsf{eq\_constraint}(x_1,x_2)) & := & \\
\multicolumn{3}{r}{\{(i,r,pc,\sigma) \in \mathcal{P} \mid \text{if } x_1\sigma, x_2\sigma \text{ are defined then } x_1\sigma = x_2\sigma\}} \\
constrained(\mathsf{neq\_constraint}(x_1,x_2)) & := & \\
\multicolumn{3}{r}{\{(i,r,pc,\sigma) \in \mathcal{P} \mid \text{if } x_1\sigma, x_2\sigma \text{ are defined then } x_1\sigma \neq x_2\sigma\}} \\
constrained(\mathsf{membership\_constraint}(x,v)) & := & \\
\multicolumn{3}{r}{\{(i,r,pc,\sigma) \in \mathcal{P} \mid \text{if } x\sigma \text{ is defined then } x\sigma \in v\}} \\
constrained(\mathsf{domain\_constraint}(x,pred)) & := & \\
\multicolumn{3}{r}{\{(i,r,pc,\sigma) \in \mathcal{P} \mid \text{if } x\sigma \text{ is defined then } x\sigma \in I_0^{pred}\}} \\
proc(\mathsf{sessions}(\sigma) & := & \{(i,r,pc,\sigma') \in \mathcal{P} \mid \sigma' \subseteq \sigma\} \\
proc(\mathsf{session}(label,\sigma)) & := & \{(i_1,r_1,pc,\sigma_1),\ldots,(i_n,r_n,pc,\sigma_n) \mid \\
\multicolumn{3}{r}{\text{every } (i_j,r_j,pc,\sigma_j) \in \mathcal{P},\ \sigma_j \subseteq \sigma,\ r_1,\ldots,r_n \text{ are the roles of the protocol}\}}
\end{array}
$$

In these definitions, $\sigma' \subseteq \sigma$ means $dom(\sigma') \subseteq dom(\sigma)$ and $\sigma|_{dom(\sigma')} = \sigma'$.
Note that $proc(\mathsf{session}(label,\sigma))$ is finite, whereas $proc(\mathsf{sessions}(\sigma))$ is infinite, it is an infinite set of copies of the processes in $proc(\mathsf{session}(label,\sigma))$.

The initial set of processes $\mathcal{S}_0$ (first component of the initial state of the system) is defined as the union of all the sets $proc(d)$ for every session declaration d in the abstract specification.

4.5 Transitions

4.5.1 Inverse key

The inverse k^{-1} of a term k (representing a cryptographic key) is defined as follows:

if $k = \mathsf{app}(i,\ell)$ and i is a symbol of type τ^{kp}, for some type τ, which was declared by: $\mathsf{dcl\_keypair}(algo, i, i', scope, \tau, type\_list)$, then $k^{-1} = \mathsf{app}(i',\ell)$.

if $k = \mathsf{app}(i',\ell)$ and i' is a symbol of type τ^{kp}, for some type τ, which was declared by: $\mathsf{dcl\_keypair}(algo, i, i', scope, \tau, type\_list)$, then $k^{-1} = \mathsf{app}(i,\ell)$.

if $k = \mathsf{cons}(t, \mathsf{empty\text{-}list})$, then $k^{-1} = \mathsf{cons}(t^{-1}, \mathsf{empty\text{-}list})$.

otherwise, $k^{-1} = k$ (k is a symmetric key).

4.5.2 Intruder's deductions

We assume that in each reachable state $\mathcal{S}, I)$ of the protocol, the intruder's knowledge $\mathcal{T}(\mathcal{F}) \setminus I^{\mathsf{secret}}$ is saturated with the classical deduction rules of the Dolev-Yao model, formalized by the following inferences. Each inference means that the conclusion, if well typed, is added to E provided that the premises belong to E and are well-typed terms.

$$\frac{a, m, k}{\mathsf{crypt}(a,k,m)}\ \mathsf{Encrypt} \qquad \frac{}{\mathsf{empty\text{-}list}}\ \mathsf{Empty} \qquad \frac{m_1, m_2}{\mathsf{cons}(m_1,m_2)}\ \mathsf{Pair} \qquad \frac{m}{\mathsf{app}(f,m)}\ \mathsf{Apply}$$

In the rule Apply, the term f must have a type of the form τ^{number} or τ^{hnumber} (but not τ^{snumber} or τ^{hsnumber}). Hence, like in the previous version [3], the intruder is assumed to be able to perform every non secret functions (including one way functions).

$$\frac{\mathsf{cons}(m_1,m_2)}{m_1}\ \mathsf{Unpair\_L} \qquad \frac{\mathsf{cons}(m_1,m_2)}{m_2}\ \mathsf{Unpair\_R}$$

$$\frac{\mathsf{crypt}(a,k,m), a, k^{-1}}{m}\ \mathsf{Decrypt} \qquad \frac{\mathsf{app}(f,m)}{m}\ \mathsf{Invert}$$

The rule Invert applies only if the term f has a type of the form τ^{number} or τ^{snumber} (not τ^{hnumber} or τ^{hsnumber}). Hence, the intruder is assumed to be able to invert every non-hash functions.

Extensions. Hence, the EVA project is restricted to the model of Dolev and Yao but we may consider in the future some inference systems which entends the one above to new capacities of the intruder.

4.5.3 Transitions

A transition from a state $(\mathcal{S}, I)$ of the protocol to a state $(\mathcal{S}', I')$ is possible iff there is a process $(i,r,pc,\sigma) \in \mathcal{S}$, and we are in one of the following cases concerning the instruction (instr in Section 3.2) at index pc in the program of the principal r in the abstract specification:

skip: $I' = I$, $\mathcal{S}'$ is obtained from $\mathcal{S}$ by incrementing pc in the process (i,r,pc,σ) (see below the procedure)

msg(l,r,x,m) (r sends a message): I' is defined as $\mathcal{T}(\mathcal{F}) \setminus \mathcal{K}'$ where $\mathcal{K}'$ is obtained from $\mathcal{K} := (\mathcal{T}(\mathcal{F}) \setminus I^{\mathsf{secret}}) \cup \{m\sigma\}$ by saturation with the rules of Section 4.5.2. $\mathcal{S}'$ is obtained from $\mathcal{S}$ by incrementing pc in the process (i,r,pc,σ).
The translator ensures that if the abstract specification is obtained by a successful compilation of a protocol in LAEVA syntax, then $m\sigma$ is ground.

msg(l,x,x,m) (r receives a message) if there exists a ground term $t \in \mathcal{T}(\mathcal{F}) \setminus I^{\mathsf{secret}}$ and a ground matching σ' of $m\sigma$ by t ($m\sigma\sigma' = t$): $\mathcal{S}'$ is obtained from $\mathcal{S}$ by replacing (i,r,pc,σ) by $(i,r,pc,\sigma \uplus \sigma')$ and incrementing pc in this process, $I' = I$.

assign(r,x,m): $I' = I$, $\mathcal{S}'$ is obtained from $\mathcal{S}$ by replacing (i,r,pc,σ) by $(i,r,pc,\sigma \uplus \{x \mapsto m\sigma\})$ and incrementing pc in this process.
The translator ensures that if the abstract specification is obtained by a successful compilation of a protocol in LAEVA syntax, then $m\sigma$ is ground and $x \notin dom(\sigma)$.

comp(r,x,m): $I' = I$, if $x\sigma = m\sigma$ then $\mathcal{S}'$ is obtained from $\mathcal{S}$ by incrementing the *pc* in the process (i,r,pc,σ). Otherwise, the process is deleted from §, giving $\mathcal{S}'$.

block(ℓ): $I' = I$, $\mathcal{S}'$ is obtained from $\mathcal{S}$ by incrementing the *pc* in the process (i,r,pc,σ) to the index of the first instruction of the list ℓ.

switch$(t,((t_1,\ell_1),\dots,(t_n,\ell_n)))$: $I' = I$, $\mathcal{S}'$ is obtained from $\mathcal{S}$ as follows. Let i be the smallest integer $\leq n$ such that $t\sigma = t_i\sigma$. If such i exists, then *pc* is incremented in the process (i,r,pc,σ) to the first instruction of the list ℓ_i. Otherwise, *pc* is changed to the first instruction following the switch, if any, otherwise, the process (i,r,pc,σ) is simply deleted from §.

Incrementation of the program counter: In a process (i,r,pc,σ), *incrementing the counter* means:

- if the instruction at index *pc* is the last instruction of the program of r, then the process is deleted (from § to obtain $\mathcal{S}'$).
- if the instruction at index *pc* in the program of r is the last instruction of an instruction list in a block, then *pc* is changed to the index of the first instruction following the block, if any. If there is no such instruction (the block is the last instruction of the program), then the process is deleted.
- if the instruction at index *pc* in the program of r is the last instruction of a case of a switch, then *pc* is changed to the first instruction following the switch, if any. If there is no such instruction (the switch is the last instruction of the program), then the process is deleted.
- in any other case, the *pc* is incremented to the index of the next instruction in the containing list (which can be an instruction list of a block or an instruction list in a case of a switch, or the program itself).

Note that in these transitions, we do not care whether it is possible for a principal to compose a message to send or to read a received message, according to the knowledge he has gain in former steps and to the "one-way" or "secret" characteristics of functions etc. All these issues are treated by the translator and at compile time and incorporated in the programs of the abstract specification of the protocol (see [2]).

Extensions. In the EVA semantics, every transition $(\mathcal{S}, I) \to (\mathcal{S}', I')$ only affects the interpretation of the predicate secret (complement of intruder's knowledge) in I. In the future, we may wish to define semantics in which other predicates are modified in transition. In particular, changes on honest may be usefull for the verification of tripartite electronic protocols.

4.6 Correctness of claim formulas

We define first the satisfaction of a formula of type claim by a process (i,r,pc,σ) and an interpretation I.

According to Section 2.5, the leaves of terms in literal of claim formulas can be protocol identifiers declared as constants, or declared values (hence in both cases constant function symbols of the protocol interpretation domain, according to Section 4.3.1), or located variables (located(x,r) in Section 3.2). The idea to define satisfiability w.r.t.

process (i,r,pc,σ) and interpretation I. is to instanciate first the located variables in the terms in the literal of the formula using σ, and, if the terms obained are all ground, to check the satisfaction of the formula obtained in I. But, as defined in Section 4.3.6, the located variables have the form '*x@r*' and do not belong to the domain of any process substitution. Hence, given a process (i,r,pc,σ), we define a subtitution σ_r whose domain is the set of located variables of Section 4.2.3: for each located variable '*x@p*' such that $p = r$ and $x \in dom(\sigma)$, $\sigma_r(x@p) = \sigma(x)$.

We say that a process (i,r,pc,σ) is compatible with a claim formula iff for every literal A of the formula, of the form atom_eq(t_1,t_2), or atom_neq(t_1,t_2), or $pred(t_1)$ for some other unary predicate symbol *pred*:

- every located variable $x@p$ in $\overline{t_1}$ or $\overline{t_2}$ belongs to $dom(\sigma_r)$,
- and both $\overline{t_1}\sigma_r$ and $\overline{t_2}\sigma_r$ are ground.

The satisfaction of the literals by a compatible process (i,r,pc,σ) and an interpretationt I is defined as follows:

$(i,r,pc,\sigma), I \models \text{atom\_eq}(t_1,t_2)$ iff $\overline{t_1}\sigma_r = \overline{t_2}\sigma_r$

$(i,r,pc,\sigma), I \models \text{atom\_neq}(t_1,t_2)$ iff $\overline{t_1}\sigma_r \neq \overline{t_2}\sigma_r$

$(i,r,pc,\sigma), I \models pred(t_1)$ iff $\overline{t_1}\sigma_r \in I^{pred}$

We say that an abstract specification of a protocol is correct iff for all claim formula ϕ of the specification, for all state $(\mathcal{S}, I)$ reachable (by transitions defined in Section 4.5.3) from the initial state (defined in Sections 4.4.6 and 4.4.5), for all process $(i,r,pc,\sigma) \in \mathcal{S}$ compatible with ϕ, we have $(i,r,pc,\sigma), I \models \phi$.

5 Output syntaxes

5.1 Extended LAEVA syntax

This syntax reuses the syntax defined in Section 2, with an additional syntax for the multi-process presentations of protocols.

5.2 CPL

The new version of CPL [3] proposed here can be seen as a concrete syntax for the abstract syntax of Section 3, restricted to multi-process presentation of protocols (list of repeat-process in CPL syntax).

We give below the grammar for this new version.

Une spécification CPL, d'abord, est la donnée de déclarations d'identificateurs (Types), d'alias globaux (Values), d'axiomes (Axioms), d'hypothèses sur les connaissances initiales de l'intrus (Assumptions), de formules à prouver (Claims), et d'un ensemble de processus (System).

Spec = protocole EVA, syntaxe CPL

compiled-spec(Types, Values, Axioms, System, Assumptions, Claims)

Types =

types(Type-declaration*)	déclarations d’identificateurs

Type-declaration =

type(id, Scope, Type)	déclaration d’identificateur
type(id, Scope, Type, "fresh")	déclaration d’identificateur frais
| type-shared(id, Scope, Type)	déclaration d’identificateur partagé (not used)
| value-type(id, Type)	déclaration de valeur pour instances de sessions

Scope =

"constant"

| "parameter"

| "private"

| "quantified"

Values =

values(Alias*)	déclarations d’alias

Alias =

alias(id, Term)	déclaration d’alias

Axioms =

axioms(Axiom*)	listes d’axiomes

Axiom =

axiom(Term, Term, Type, Type-declaration*)	axiome, (lhs, rhs, type, variables quantifiées)

System =

system(Process*)	Le système (composition paralléle de processus)

Assumptions =

assume(Formula*)	Hypothèses

Claims =

claim(Formula*)	Formules à prouver

Type =

"principal"	type d’ordre 1
| "number"	type d’ordre 1
| "asym_algo"	type d’ordre 1
| "sym_algo"	type d’ordre 1
| "*algo*"	type d’ordre 1
| id	user type or basetype d’ordre 1
| function(id, id*)	type de fonction (inversible, non secrete)

| one-way-function(id, id*)	type de fonction one-way
| secret-function(id, id*)	type de fonction secrète
| secret-one-way-function(id, id*)	type de fonction secrète et one-way

Note: there is no type for the keypairs declared. Instead, as described in [3], we declare a unique identifier associated to both keys of the pair, and we refer to aliases using functions lambda-pubk and lambda-privk for the application of the keys.

Un processus PEVA peut être soit un processus simple, décrit par un graphe de transitions, soit un processus en multi-session parallèle, qui est un nombre non borné de processus simples décrits par le même graphe de transitions (les *copies*), mis en parallèle. Les transitions sont données comme une liste de triplets (source, cible, action). Les actions sont décrites plus bas. Dans le cas de processus en multi-session parallèle, Privates contient la liste des identificateurs qui contiennent des valeurs possiblement différentes d'une copie à l'autre; les autres identificateurs sont partagés entre les copies. Finalement, chaque processus vient avec une liste de connaissances (Know), qui sert à faire le lien entre expressions LAEVA et identificateurs correspondants à ces expressions dans les processus PEVA.

Process =

process(id, state, Know, Transition*)	processus simple (nom, label de départ, connaissances, transitions)
| repeat-process(Privates, id, state, Know, Transition*)	processus en multi-session parallèle (vars privées, role, départ, connaissances, transitions)

Transition =

trans(state, state, Action)	transition (source, cible, action)

state =

label	état visible
| '%' label	état interne

Know =

knows(As*)	connaissances

As =

as(Term, Term)	terme connu sous la forme: terme

Privates =

private(id*)	liste de variables privées

Action =

new(Pattern)	création de nonce (unused in this version)
| let(Pattern, Term)	pattern-matching (unused in this version)
| recv(Pattern)	réception de message
| send(Term)	émission de message

| comp()	
| assign()	
| skip()	action nulle

Term =

id	variable
| crypt(Term, Term, Term)	chiffrement (algo, texte, clé)
| tuple(Term*)	*n*-uplet
| p(Term)	coercion principal $\rightarrow$ number
| a(Term)	coercion *algo* $\rightarrow$ number
| sa(Term)	coercion sym_algo $\rightarrow$ *algo*
| aa(Term)	coercion asym_algo $\rightarrow$ *algo*
| vanilla()	algorithme symétrique par défaut
| apply(id, Term*)	application de fonction définie
| hash-apply(id, Term*)	application de fonction one-way
| secret-apply(id, Term*)	application de fonction secrète
| hash-secret-apply(id, Term*)	application de fonction secrète one-way
| apply-pubk(Term, id, Term*)	appl. de constructeur de clé publique (algo, constructeur, arguments) (not used, all key functions are assumed one-way)
| hash-apply-pubk(Term, id, Term*)	appl. de constructeur de clé publique one-way (algo, constructeur, arguments)
| apply-privk(Term, id, Term*)	appl. de constructeur de clé privée (algo, constructeur, arguments) (not used, all key functions are assumed one-way)
| hash-apply-privk(Term, id, Term*)	appl. de constructeur de clé privée one-way (algo, constructeur, arguments)
| lambda-pubk(Term, id)	partie publique de clé (algo, clé)
| lambda-privk(Term, id)	partie privée de clé (algo, clé)
| located(label, id, Term)	terme vu par session/principal (label not significant in this version)

Pattern =

id	variable
| exact(Term)	constante littérale
| crypt(Term, Pattern, Term)	déchiffrement (algo, texte, clé)
| apply(id, Pattern*)	application de constructeur

| secret-apply(id, Pattern*)	application de constructeur secret
| tuple(Pattern*)	*n*-uplet
| p(Pattern)	extraction number $\rightarrow$ principal
| a(Pattern)	extraction number $\rightarrow$ *algo*
| sa(Pattern)	extraction *algo* $\rightarrow$ sym_algo
| aa(Pattern)	extraction *algo* $\rightarrow$ asym_algo
| vanilla()	algorithme symétrique par défaut
| apply-pubk(Term, id, Pattern*)	appl. de constructeur de clé publique (algo, constructeur, arguments)
| apply-privk(Term, id, Pattern*)	appl. de constructeur de clé privée (algo, constructeur, arguments)

Formula =

clause(Atom*, Atom)	Horn clause
| negative(Atom*)	negative Horn clause

Atom =

secret(Term)

| honest(Term)

| eq(Term, Term)

| neq(Term, Term)

References

[1] Michael Burrows, Martin Abadi, and Roger Needham. A logic of authentication. Technical Report 39, Digital Systems Research Center, february 1989.

[2] Florent Jacquemard, Michael Rusinowitch, and Laurent Vigneron. Compiling and verifying security protocols. In *7th International Conference on Logic for Programming and Automated Reasoning*, Lecture Notes in Artificial Intelligence. Springer Verlag, november 2000.

[3] Jean Goubault-Larrecq. Les syntaxes et la sémantique du langage de spécification EVA. *Rapport Technique EVA No 3* novembre 2001.

[4] Florent Jacquemard and Daniel Le Métayer. Langage de spécification de protocoles de EVA: syntaxe concrète. Technical Report EVA-1, Trusted Logic S.A., July 2001. Version 3.14.

[5] Jean Goubault-Larrecq. Langage de spécification de protocoles de EVA: syntaxe abstraite et sémantique. Technical Report EVA-2, LSV/CNRS UMR 8643 & ENS Cachan, July 2001.

[6] Liana Bozga, Yassine Lakhnech and Michael Périn. L'outil de vérification HERMES. Technical Report EVA-6, laboratoire VERIMAG, mai 2002.

[7] Liana Bozga, Yassine Lakhnech and Michael Périn. Hermes,a tool verifying secrecy properties of unbounded security protocols. In *15th international conference on Computer-Aided Verification* (CAV'03), Lecture Notes in Computer Science, Springer Verlag, july 2003.