AVISPA

www.avispa-project.org

IST-2001-39252

Automated Validation of Internet Security Protocols and Applications

AVISPA v1.0 User Manual

AVISPA Team

Document Version: 1.0
March 31, 2006

Information Society Technologies

Project funded by the European Community under the *Information Society Technologies* Programme (1998-2002)

Contents

1 Introduction

AVISPA (Automated Validation of Internet Security Protocols and Applications) is a push-button tool for the automated validation of Internet security-sensitive protocols and applications. It provides a modular and expressive formal language for specifying protocols and their security properties, and integrates different back-ends that implement a variety of state-of-the-art automatic analysis techniques.

The AVISPA tool has been realized thanks to the AVISPA shared cost RTD (FET open) project IST-2001-39252, by the *Artificial Intelligence Laboratory* at DIST (University of Genova, Genova, Italy), the *CASSIS* group at INRIA Lorraine (LORIA, Nancy, France), the *Information Security* group at ETHZ (Zürich, Switzerland), and *Siemens AG* (Munich, Germany).
For more information on this project and on the AVISPA Team, please consult

`http://www.avispa-project.org/`

or subscribe to the `avispa-users@avispa-project.org` mailing list by sending an empty e-mail to:

`avispa-users-join@avispa-project.org`.

1.1 Installation Procedure

The AVISPA tool is distributed with a copyright. Please take a look at the LICENSE file provided with the tool before proceeding with the installation.

This software is being developed as a research tool. We continue to make significant changes to it. This is an "alpha" release which we are doing primarily in order to get feedback. We are willing to know what you think of AVISPA, so please send comments to us at `avispa-users@avispa-project.org`.

In order to install AVISPA vX.Y, you need to extract the archive `avispa-package-X.Y_Linux-i686.tgz` in the desired directory, which will create a new sub-directory named `avispa_1.0` populated by a number of files and sub-directories. Then you need to set the environment variable `AVISPA_PACKAGE` to refer to the absolute path ending in `avispa-X.Y`, and to put the script called `avispa` in the execution path of your shell.

For example, if you want to install the AVISPA tool in a directory `/opt`, in a *bash* shell environment, the commands are:

```
cd /opt
```

```
tar -xzf /home/xyz/avispa-package-X.Y_Linux-i686.tgz
export AVISPA_PACKAGE=/opt/avispa-X.Y
export PATH=$PATH:$AVISPA_PACKAGE
```

Now you should be able to execute AVISPA, using the command `avispa`. Please see the `README` file for information about the command line options of AVISPA.

The AVISPA package provides a user-friendly mode for XEmacs to allow a simple interaction between the user and the modules of the AVISPA package. To set-up the XEmacs mode follow the instructions below:

```
cd $AVISPA_PACKAGE/others
tar -xzf avispa-mode.tgz
```

This command will create a directory `temporary-avispa` containing a makefile for installing the XEmacs mode. Follow the instruction in `temporary-avispa/help.txt`; when done, delete the temporary directory `temporary-avispa`.

The AVISPA package further provides the *h*lpsldoc tools for documenting HLPSL specifications in LATEXand HTML format. To set them up, follow the instructions below:

```
cd $AVISPA_PACKAGE/others/hlpsldoc
tar xzf hlpsldoc.tgz
```

Then follow the instructions in the local `INSTALL` file.
Usage of the hlpsldoc tools is explained in the local `README` file.

The current distribution has been tested on several Linux platforms (please refer to the global `README` file), but if you encounter any difficulty in installing and running AVISPA on your machine, please send us a message to `avispa-users@avispa-project.org`.

1.2 How to use the AVISPA tool?

The typical interaction with the AVISPA tool is as follows:

1. You start by specifying the protocol in HLPSL, the AVISPA specification language, including the properties that you want to check, then

2. you invoke AVISPA by issuing the `avispa` command at the prompt and by specifying which analyser (back-end) you want to use,

3. you look at the output and maybe shout "Yeahhhhh!" if you will see that the AVISPA tool has declared that your protocol is safe (maybe under some conditions), or "Ups!" when you will see that an attack has been found. In the latter event you may decide to modify your protocol specification and interaction continues at step 2.

Note that you can use the HLPSL XEmacs mode provided in the AVISPA package; its syntax highlighting and menus are very practical for writing a protocol specification and verifying it automatically.

1.3 About this Manual

This manual is divided in two part.

- **For non expert users**, the User Section describes the syntax of the specification language (HLPSL), and the different steps for analysing a HLPSL specification and interpreting the output.
- **For expert users**, the Advanced User Section describes how to use all the power of the AVISPA tool; it presents the translator of HLPSL specifications into IF (Intermediate Format) specifications; then it shows how to use each back-end; it also gives the common syntax of the output of the back-ends.

In the appendix, the XEmacs mode is described, and the semantics of the HLPSL and IF specification languages are detailed.

1.4 Contact

For contacting the AVISPA Team, either for questions, or for proposing improvements, or for bug reports, please do not hesitate to use the following mailing list:
`avispa-users@avispa-project.org`
You need to subscribe to the above list, either by sending an empty e-mail to
`avispa-users-join@avispa-project.org`,
or by following the link below:
`http://www.avispa-project.org/mailman/listinfo/avispa-users`
All registered AVISPA tool users will also receive your message.

2 User Section

This section describes the easiest way to use the AVISPA tool: to specify protocols in HLPSL, then to run the `avispa` script for analysing it.

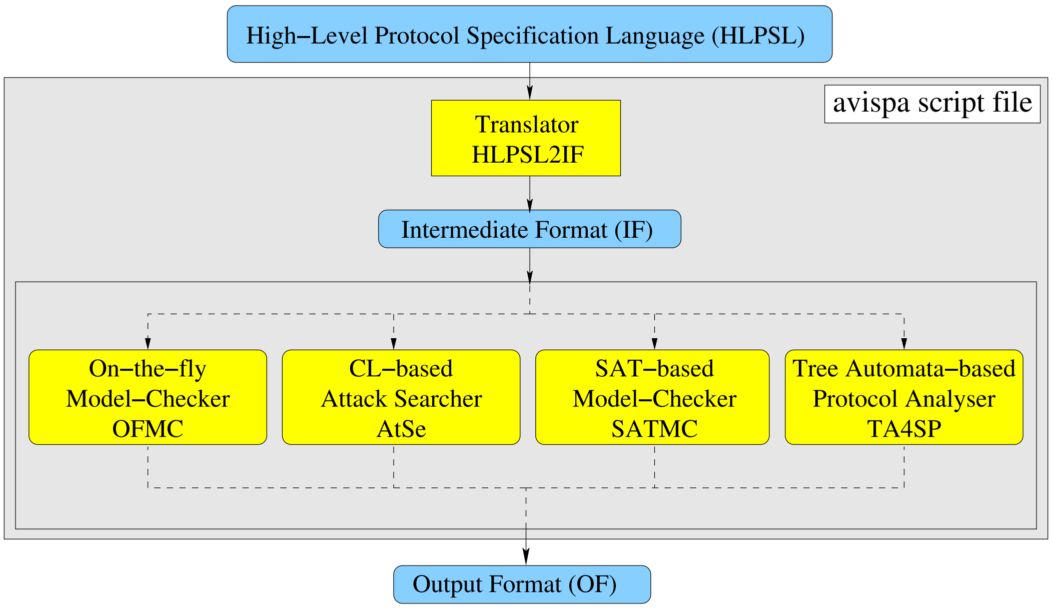

High−Level Protocol Specification Language (HLPSL)
avispa script file
Translator HLPSL2IF
Intermediate Format (IF)
On−the−fly Model−Checker OFMC
CL−based Attack Searcher AtSe
SAT−based Model−Checker SATMC
Tree Automata−based Protocol Analyser TA4SP
Output Format (OF)

Figure 1: Architecture of the AVISPA tool v.1.0

2.1 Specifying Protocols: HLPSL

Protocols to be studied by the AVISPA tool have to be specified in HLPSL (standing for *High Level Protocols Specification Language*), and written in a *file with extension* ***`hlpsl`***.
This language is based on roles: basic roles for representing each participant role, and composition roles for representing scenarios of basic roles. Each role is independent from the others, getting some initial information by parameters, communicating with the other roles by channels.

In this section, we present the syntax of HLPSL and some guidelines for HLPSL beginners.

2.1.1 HLPSL Syntax

The syntax of HLPSL is detailed in the following, using the standard BNF.
Before describing the syntax, we list the lexical entities used in the grammar, keywords and expressions being written as strings (i.e. arbitrary sequences of characters enclosed by two `"` characters).

a. Lexical entities. In HLPSL, all variables start with a capital letter, and constants start with a small letter; note that natural numbers can also be used as constants (without any specific interpretation).

```
var_ident: [A-Z][A-Za-z0-9_]*
const_ident: [a-z][A-Za-z0-9_]*
nat_ident: [0-9]+
```

The list of HLPSL keywords is the following:

```
accept, agent, authentication_on, bool, channel, composition, cons,
const, def=, delete, dy, end, exp, goal, hash, hash_func, iknows,
in, init, intruder_knowledge, inv, local, message, nat, new, not,
ota, owns, played_by, protocol_id, public_key, request, role,
secrecy_of, secret, set, start, symmetric_key, text, transition,
weak_authentication_on, witness, wrequest, xor.
```

Some other constant names are reserved and will be automatically renamed if used in a specification:

```
apply, attack_state, attack_states, contains, crypt, dummy_agent,
dummy_bool, dummy_chnl, dummy_chnl_dy, dummy_chnl_ota, dummy_hash,
dummy_msg, dummy_nat, dummy_nonce, dummy_pk, dummy_set, dummy_sk,
equal, equations, initial_state, inits, intruder, leq, pair,
properties, property, rules, scrypt, section, step, types.
```

In a HLPSL specification comments and separators (e.g. "white space", new line and tabulation characters) are ignored:

comments: `%[^\n]*`
spaces: `[\n\r\t ]`

b. Structure of a HLPSL specification. A HLPSL specification is composed of three parts: a list of definitions of roles, a list of declarations of goals (if any), and the instantiation (read *call*) of the main role (usually without arguments).

```
SpecHLPSL ::=
  Role_definition+
  Goal_declaration?
  % Call of the main role: (ex: environment() )
  Role_instantiation
```

c. Definition of roles. The roles in a specification are of two kinds: basic roles played by agents, and composition roles describing the scenario to consider during analysis (for example, describing what is a session of the protocol, or what instances of sessions should be used).

```
% Roles may be either basic or compositional:
Role_definition ::=
  Basic_role
| Composition_role
```

d. Definition of a role. Roles are independent processes: they have a name, receive information by parameters and contain local declarations.

Basic roles are played by an agent whose name is received as parameter. The actions of a basic role are transitions, describing changes in their state depending on events or facts.

```
% Basic roles must include a player definition and generally
% contain a transition declaration section.
Basic_role ::=
  "role"
  Role_header
  Player
  Role_declarations
  Transition_declaration
  "end" "role"

% Used to bind the role and the identifier of the agent playing the role.
Player ::=
  "played_by" var_ident
```

Note that all the information of a basic role (parameters and local variables) represents the knowledge of the player of the role.

A composition role combines other roles, either in parallel, or in sequence.

```
% Composition roles have no transition section, but rather
% a composition section in which they call other roles.
Composition_role ::=
  "role"
  Role_header
  Role_declarations
  Composition_declaration
  "end" "role"
```

e. Declarations in roles. The first element in a role is its header. It contains the role name (a constant) and its parameters (a list of declarations of variables with their type).

```
Role_header ::=
  const_ident "(" Formal_arguments? ")"

Formal_arguments ::=
  (Variable_declaration ",")* Variable_declaration
```

A role may contain numerous declarations:

- local declarations: declarations of variables with their type;
- constants declarations: declaring constants with their type is not local to the role; any constant in one role can be used in another one;
- initialisations: initialisation of local variables;
- accept declarations: conditions for which the role can be considered as done;
- intruder knowledge declaration: a set of information that is given to the intruder at the beginning of the role execution.

```
Role_declarations ::=
  "def="
  Local_declaration?
  Owns_declaration?      % not handled yet
  Const_declaration?
  Init_declaration?
  Accept_declaration?    % not handled yet
  IKnowledge_declaration?
```

f. Declaration of local variables. Declarations of different types are separated by a comma.

```
% Declaration of local variables.
Local_declaration ::=
  "local" Variables_declaration_list

Variables_declaration_list ::=
  (Variable_declaration ",")* Variable_declaration
```

Note that, even if variables are local to roles, the same variable declared in different roles has to have the same type (for avoiding automatic renaming that could bother the understanding of the result by the user).

g. Declaration of owned variables. Roles may declare ownership of variables.

```
% Declaration of owned variables.
Owns_declaration ::=
  "owns" Variables_list
```

Owned variables may change in *only* the way described by the owning role, even if they are visible from outside.
However, since shared variables are not fully handled in the current version of the AVISPA tool, ”owns” declarations are useless.

h. Declaration of constants. Constants are declared in roles, but are global. Multiple declarations of a constant do not raise an error, provided the type is the same.

```
Const_declaration ::=
  "const" Constants_declaration_list

Constants_declaration_list ::=
  (Constant_declaration ",")* Constant_declaration
```

i. Initialisation of local variables. The initialisation section is a conjunction of simple assignments to variables and of constant predicates. Expressions used for assignments and in the predicates have to used initialised variables; this is the meaning of a stutter expression.

```
Init_declaration ::=
  "init" Init_declarations

Init_declarations ::=
  (Init_declaration "/\")* Init_declaration

Init_declaration ::=
  var_ident ":=" Stutter_expression
| const_ident "(" Stutter_expressions_list? ")"
```

Let us give a simple example of initialisation:

```
init State := 0 /\ SetKeys := {} /\ iknows(K)
```

j. Declaration of the acceptance state. Acceptance is used for sequential composition to mark the stop states after which the following instantiation may begin.

```
Accept_declaration ::=
  "accept" Predicates
```

However, since sequential composition is not fully handled in the current version of the AVISPA tool, "accept" declarations are useless.

k. Declaration of intruder knowledge. The knowledge given to the intruder is defined by a list of constants, assigned variables (either parameters or initialised local variables) or messages build with constants and assigned variables.

```
IKnowledge_declaration ::=
  "intruder_knowledge" "=" "{" Stutter_expressions_list? "}"
```

This knowledge is given to the intruder in a role, but the full knowledge of the intruder is the union of all the knowledge given to him in all instances of basic and composition roles.

l. Transitions in basic roles. The transitions in a basic role are either spontaneous actions, enabled when the state predicates on the left-hand side are true, or immediate reactions fired immediately whenever the non-stutter events (that is events based on the change of some variables values) on the left-hand side are true.

```
Transition_declaration ::=
  "transition" Transition*

Transition ::=
  Label "." Predicates "--|>" Actions  % spontaneous action
| Label "." Events "=|>" Reactions     % immediate reaction

Label ::=
  const_ident
| nat_ident
```

Note that each transition starts with a label (a constant or a natural number followed by a dot).

The condition for applying a spontaneous action is a conjunction of predicates, representing a state of the role (all the information in those predicates is already known).

```
Predicates ::=
  (Predicate "/\")* Predicate

Predicate ::=
  Stutter_formula
| const_ident "(" Stutter_expressions_list? ")"
| var_ident "(" Stutter_expression ")"
```

A predicate of the last form has to correspond to the reception of a message in a channel (for example: `Rcv({M'}_K)`).

Contrarily to spontaneous actions, immediate reactions happen when the player of the role is in a given state and has to react to some events (a reception of a message, for example) that will change the value of some variables.

```
Events ::=
  ((Predicate|Event) "/\")* Event ("/\" (Predicate|Event))*

Event ::=
  Non_stutter_formula
| const_ident "(" Non_stutter_expressions_list? ")"
| var_ident "(" Non_stutter_expression ")"
| var_ident "(" "start" ")"   % Dummy start message for Dolev-Yao models
```

The `start` message is used as a signal sent to the player of the role, for asking him to start a session of the protocol. In the last two cases, the variable has to be a channel.

m. Actions and reactions. Actions and reactions are syntactically similar; they differ only by the context in which they are used.
An action can be the assignment of a variable, possibly with a fresh value (e.g.: `Na':=new()`), the call of a user-defined predicate (such as sending a message in a channel), or the call of predefined goal predicates. The expression assigned to a variable can be either a stutter one, or a non-stutter one.
In any transition, the old value and the new value of a variable are syntactically distinguished: the *prime* symbol (') has to be attached to the name of a variable for considering its new value. Examples: `M':={M}_K` and `Snd(M'.M)`.

```
Reactions ::=
  Actions

Actions ::=
  (Action "/\")* Action

Action ::=
  var_ident "'" ":=" Expression
| var_ident "'" ":=" "new" "(" ")"
| const_ident "(" Expressions_list? ")"
| var_ident "(" Expression ")"
| "secret" "(" Expression "," const_ident "," Expression ")"
| "witness" "(" Expression "," Expression "," const_ident "," Expression ")"
| "request" "(" Expression "," Expression "," const_ident "," Expression ")"
| "wrequest" "(" Expression "," Expression "," const_ident "," Expression ")"
```

The goal predicates contain the following information:

- `secret(E,id,S)`: declares the information `E` as secret shared by the agents of set `S`; this secret will be identified by the constant `id` in the goal section;
- `witness(A,B,id,E)`: for a (weak) authentication property of `A` by `B` on `E`, declares that agent `A` is witness for the information `E`; this goal will be identified by the constant `id` in the goal section;
- `request(B,A,id,E)`: for a strong authentication property of `A` by `B` on `E`, declares that agent `B` requests a check of the value `E`; this goal will be identified by the constant `id` in the goal section;
- `wrequest(B,A,id,E)`: similar to `request`, but for a weak authentication property.

n. Composition of roles. Roles, basic and/or composition, can be composed in parallel or sequentially. Such scenarios are described in the composition section of so called *composition roles.*[1]

```
% Definition of the composition section (for composed roles)
Composition_declaration ::=
  "composition" Compositions_list?

Compositions_list ::=
  Composition
| Composition "/\" Bracketed_par_compositions_list  % parallel
| Composition ";" Bracketed_seq_compositions_list   % sequential
| "(" Compositions_list ")"

Composition ::=
| "/\" "_" "{" Parameters_instance "}" Bracketed_compositions_list
| Role_instantiation

Parameters_instance ::=
  "in" "(" Concatenated_variables_list "," Stutter_expression ")"
```

[1]Note that in the current version of the AVISPA tool, sequential composition of roles is not handled yet.

```
Concatenated_variables_list ::=
  Concatenated_variables
| "(" Concatenated_variables ")"

Concatenated_variables ::=
  (var_ident ".")* var_ident

Bracketed_par_compositions_list ::=
  Composition
| Composition "/\" Bracketed_par_compositions_list
| "(" Compositions_list ")"

Bracketed_seq_compositions_list ::=
  Composition
| Composition ";" Bracketed_seq_compositions_list
| "(" Compositions_list ")"

Bracketed_compositions_list ::=
  Composition
| "(" Compositions_list ")"
```

An example of composition of roles is:

```
server(S,Ks) /\
/\_{in(A.B.Ka.Kb,Instances)} (alice(A,Ka) /\ bob(B,Kb))
```

Note that in this case, `Instances` has to be a set whose elements are of the compound type `agent.agent.public_key.public_key`, provided `A` and `B` are variables of type `agent`, and `Ka` and `Kb` are variables of type `public_key`. For example, `{a.b.ka.kb, a.i.ka.ki}` could be the value of `Instances`.

o. Instantiation of a role. To create an instantiation of a role is like calling a procedure, giving values to each argument. Of course, the number of arguments has to be the same as the number of formal parameters, and the type of each argument has to be compatible with the type of the corresponding formal parameter.

```
Role_instantiation ::=
  const_ident "(" Expressions_list? ")"
```

p. Declaration of goals. Goals are declared in a specific section. Such declarations are done either by using predefined macros, or by using a LTL formula[2].

[2]LTL formulas are handled by the translator, but in the current version of the AVISPA tool, no backend does use them.

The available macros correspond to:

- the secrecy of some information,
- the strong authentication of agents on some information,
- the weak authentication of agents on some information.

Each goal is identified by a constant, referring to predefined predicates (`secret`, `witness`, `request` and `wrequest`) added in transitions by the user. For more details on those predicates, see the description of actions, page 15.

```
Goal_declaration ::=
  "goal" Goal_formula+ "end" "goal"

Goal_formula ::=
  "secrecy_of" Constants_list
| "authentication_on" Constants_list
| "weak_authentication_on" Constants_list
| "[]" LTL_unary_formula

LTL_unary_formula ::=
  LTL_unary_predicate
| "<->" LTL_unary_formula
| "(-)" LTL_unary_formula
| "[-]" LTL_unary_formula
| "~" LTL_unary_formula
| "(" LTL_formula ")"

LTL_formula ::=
  LTL_predicate
| "<->" LTL_unary_formula
| "(-)" LTL_unary_formula
| "[-]" LTL_unary_formula
| LTL_formula "/\" LTL_formula
| LTL_formula "\/" LTL_formula
| LTL_formula "=>" LTL_formula
| "~" LTL_unary_formula
| "(" LTL_formula ")"

LTL_unary_predicate ::=
  const_ident "(" Stutter_expressions_list? ")"
| "in" "(" Stutter_expression "," Variable_or_constant ")"

LTL_predicate ::=
  LTL_unary_predicate
| Stutter_expression "=" Stutter_expression
| Stutter_expression "<=" Stutter_expression
| Stutter_expression "/=" Stutter_expression
```

In the temporal formula, "`<->`" means "sometimes in the past", "`(-)`" means "one time instant in the past", "`[-]`" means "globally in the past". The other symbols are standard logical connectives: conjunction (`/\`), disjunction (`\/`), implication (`=>`) and negation(`~`).
Note that LTL formulas always start by "`[]`", the "always" temporal operator.

An example of goal section is the following:

```
  goal
    authentication_on nb
    weak_authentication_on na
    secrecy_of na, nb
    [] (<-> has_seen(A,B,M) => ((has_seen(B,A,M) /\ ~iknows(M)) \/ B=i))
  end goal
```

Note that the last LTL formula is just an example, without any serious meaning. It is given only to show that user-defined predicates (such as `has_seen`) can be used, so as `iknows` for representing intruder's knowledge.

q. Declaration of types of variables. All variables must be declared, with the most specific type possible (or at least with the generic type `message`).

```
  Variable_declaration ::=
    Variables_list ":" Type_of

  Variables_list ::=
    (var_ident ",")* var_ident

  Type_of ::=
    (Subtype_of "->")* Subtype_of

  Subtype_of ::=
    Simple_type
  | "(" Subtype_of ")"
  | Compound_type
```

Note that a variable can be of type function, using "`->`" for separating the types of the arguments and the type of the result.
More generally, types are either simple or compound.

r. Declaration of types of constants. To declare constants is similar to declaring variables, except that the type of a constant cannot be compound.

```
Constant_declaration ::=
    Constants_list ":" Simple_type_of

Constants_list ::=
    (const_ident ",")* const_ident

Simple_type_of ::=
    (Simple_subtype_of "->")* Simple_subtype_of

Simple_subtype_of ::=
    Simple_type
  | "(" Simple_type_of ")"
```

s. Types and compound types. The types proposed are the standard ones (agent, key, text, channel), and also more advanced ones, like hash functions and enumerations. Types `nat` and `bool` do not have any predefined semantics. The generic type, compatible with all the others, is `message`. Variables representing channels[3] have an additional attribute indicating the level of protection: `dy` for no protection; `ota` for forbidding divert actions of the intruder.

```
Simple_type ::=
    "agent"
  | "channel"
  | "channel" "(" "dy" ")"
  | "channel" "(" "ota" ")"
  | "public_key"
  | "symmetric_key"
  | "text"            % used for nonces
  | "message"         % generic type
  | "protocol_id"     % kind of label
```

[3]In the current version of the AVISPA tool, only Dolev-Yao channels are supported.

```
| "nat"
| "bool"
| "hash_func"
| "{" Constants_or_nat_list "}"  % enumeration

Constants_or_nat_list ::=
  const_ident
| nat_ident
| const_ident "," Constants_or_nat_list
| nat_ident "," Constants_or_nat_list
```

HLPSL also allows for the specification of compound types. Compound types allow the protocol designer to declare HLPSL variables of sorts restricted and specialised in a particular way, and permit to provide a detailed description of the contents of a variable, using concatenation, sets, encryption, inverse of keys and result of the application of a hash function.

```
Compound_type ::=
  Subtype_of "." Subtype_of
| Subtype_of "set"
| "{" Subtype_of "}" "_" Bracketed_subtype_of
| "inv" "(" Subtype_of ")"
| "hash" "(" Subtype_of ")"

Bracketed_subtype_of ::=
  Simple_type
| "inv" "(" Subtype_of ")"
| "(" Subtype_of ")"
```

t. Stutter and non-stutter formulas. A stutter formula is a formula that does not use the new value of a variable; such formulas can be comparisons, set membership tests.

```
Stutter_formula ::=
  Stutter_expression "=" Stutter_expression
| Stutter_expression "<=" Stutter_expression
  % Inclusion test: in(Elt,Set)
| "in" "(" Stutter_expression "," Stutter_expression ")"
| "in" "(" Non_stutter_expression "," Stutter_expression ")"
  % Syntactic sugar for inequality:
| Stutter_expression "/=" Stutter_expression
| "not" "(" Stutter_formula ")"
| "(" Stutter_formula ")"
```

A non-stutter formula uses the new value of at least one variable, represented by "priming" the variable (example: `Na'`).

```
Non_stutter_formula ::=
  Non_stutter_expression "=" Stutter_expression
| Stutter_expression "=" Non_stutter_expression
| Non_stutter_expression "=" Non_stutter_expression
| Non_stutter_expression "<=" Stutter_expression
| Stutter_expression "<=" Non_stutter_expression
| Non_stutter_expression "<=" Non_stutter_expression
  % Inclusion test: in(Elt,Set)
| "in" "(" Non_stutter_expression "," Non_stutter_expression ")"
| "in" "(" Stutter_expression "," Non_stutter_expression ")"
  % Syntactic sugar for inequality:
| Non_stutter_expression "/=" Stutter_expression
| Stutter_expression "/=" Non_stutter_expression
| Non_stutter_expression "/=" Non_stutter_expression
| "not" "(" Non_stutter_formula ")"
| "(" Non_stutter_formula ")"

Stutter_expressions_list ::=
  (Stutter_expression ",")* Stutter_expression

Non_stutter_expressions_list ::=
  Non_stutter_expression
| Non_stutter_expression "," Expressions_list
| Stutter_expression "," Non_stutter_expressions_list
```

u. Stutter and non stutter expressions. Expressions are composed with variables and constants, combined by concatenation or encryption, or used with functions or sets.

```
Stutter_expression ::=
  "(" Stutter_expression ")"
| Variable_or_constant_or_nat
| "inv" "(" Stutter_expression ")"
  % Concatenation, right-associative:
| Stutter_expression "." Stutter_expression
  % Function application:
| Variable_or_constant "(" Stutter_expressions_list ")"
  % Set:
| "{" "}"
| "{" Stutter_expressions_list "}"
  % Encryption : {Na.A}_inv(Ka)
| "{" Stutter_expression "}" "_" Bracketed_stutter_expression

Non_stutter_expression ::=
  "(" Non_stutter_expression ")"
| var_ident "'"
| "inv" "(" Non_stutter_expression ")"
  % Concatenation, right-associative:
| Non_stutter_expression "." Stutter_expression
| Stutter_expression "." Non_stutter_expression
| Non_stutter_expression "." Non_stutter_expression
  % Function application:
| Variable_or_constant "(" Non_stutter_expression_list ")"
  % Insertion of an element in a set: cons(Elt,Set)
| "cons" "(" Expression "," Expression ")"
  % Deletion of an element in a set: delete(Elt,Set)
| "delete" "(" Expression "," Expression ")"
  % Set:
| "{" Non_stutter_expressions_list "}"
  % Encryption: {Na'.A}_(Ka.Kb')
| "{" Non_stutter_expression "}" "_" Bracketed_expression
| "{" Stutter_expression "}" "_" Bracketed_non_stutter_expression

Expressions_list ::=
  (Expression ",")* Expressions

Expression ::=
  Stutter_expression
| Non_stutter_expression

Bracketed_stutter_expression ::=
  "inv" "(" Stutter_expression ")"
| Variable_or_constant "(" Stutter_expressions_list ")"
| Variable_or_constant_or_nat
| "(" Stutter_expression ")"

Bracketed_non_stutter_expression ::=
  var_ident "'"
| "inv" "(" Non_stutter_expression ")"
| Variable_or_constant "(" Non_stutter_expressions_list ")"
| "(" Non_stutter_expression ")"

Bracketed_expression ::=
  Bracketed_stutter_expression
| Bracketed_non_stutter_expression

Variable_or_constant ::=
  var_ident
| const_ident

Variable_or_constant_or_nat ::=
  var_ident
| const_ident
| nat_ident
```

v. Predefined equational theories. In HLPSL specifications, several specific operators have predefined equational properties:

- **concatenation:** this operator, "." is associative:

$$\texttt{(A.B).C} = \texttt{A.(B.C)}$$

- **exclusive or:** this operator, "`xor(A,B)`", is associative, commutative and nilpotent:

$$\begin{aligned} \texttt{xor(xor(A,B),C)} &= \texttt{xor(A,xor(B,C))} \\ \texttt{xor(A,B)} &= \texttt{xor(B,A)} \\ \texttt{xor(xor(A,A),B)} &= \texttt{B} \end{aligned}$$

- **exponential:** this operator, "`exp(E,N)`" representing E^N, can commute exponents, and admits an inverse for exponents:

$$\begin{aligned} \texttt{exp(exp(E,N),M)} &= \texttt{exp(exp(E,M),N)} \\ \texttt{exp(exp(E,N),inv(N))} &= \texttt{E} \end{aligned}$$

Those operators can be used in any expression: concatenation is explicitly recognised in the grammar; `xor` and `exp` are part of the function applications in expressions.

2.1.2 HLPSL Guidelines

This section will guide you for writing in a "good" way an HLPSL specification.

a. Variable/constant names.
Do not use the same variable/constant name in different roles with different types.

b. Arithmetic.
Do not use arithmetic operators/relations (e.g.'+', '=<'). They are not supported by the translator.

c. Old/new values of variables.
A primed variable (eg. `X'`) represents the new value of a variable in a transition: this new value has been either learned in the left-hand side of the transition (received on a channel, or found by decomposing a message or searching in a set), or assigned in the right-hand side of the transition. Assigning the new value of a variable with `new()` means assigning it with a fresh value (i.e. a nonce).

A primed variable must not appear in the initialisation section, or in the intruder knowledge declaration of a role.

d. Channels.

- Constants should not be declared of type channel;
- When only DY channels are used, then it is possible to express the transition relation of honest agents by means of immediate reactions. In such cases the used DY channels should be each one different from another. This can be easily achieved by declaring a different HLPSL variable for each DY channel.
  Channels should be declared as local variables of the `Session` role or in the `Environment` role and never in the basic roles.

e. Goal specification.
Goals are specified as macros representing pre-defined safety temporal formulae built on top of the goal predicates `witness`, `wrequest` (for weak authentication), `request` (for strong authentication), and `secret`. These goal predicates are explicitly declared in right-hand sides of HLPSL transitions and are translated into corresponding IF facts (`request` and `wrequest` facts will be augmented with the role ID). These predicates are used to specify secrecy and different forms of authentication.

- **Secrecy** is modelled by means of the goal predicate `secret(T,id,{A,B})` standing for "the value of term `T` is a secret shared only between agents `A` and `B`". The secrecy property is violated everytime the intruder learns a value that is considered a secret and that he is not allowed to know. (Note that if in a certain session the intruder plays the role of an honest agent that is allowed to know the secret value, then the intruder is allowed to know it and

no attack is reported for this value.)
The label `id` (of type `protocol_id`) is used to identify the goal. In the HLPSL goal section the statement `secrecy_of id` should be given to refer to it.
The set of agents sharing the secret has to be written as a constant set, and not by using a variable of type `agent set`.

The secrecy events should be given as early as possible, i.e. right when the secret term has been created in the respective role(s), because the secrecy check takes effect only after the events have been issued and it will stay in effect till the end of the protocol run.

If a value `T` that should be kept secret is determined by a single role (in particular, if it is an atomic value like a nonce produced by `new()`), then the secrecy statement should be given in — and only in — the role introducing the value.

If the secret is a combination of ingredients from several roles, then secrecy predicates should be given in all roles contributing to the non-atomic secret value. Unfortunately, if the intruder plays one of these roles in one session and legitimately learns the “secret”, then he can re-use this value in some other session (where he does not play the role of an honest agent) to masquerade as one of the honest agents, while the other agents believe that the value is a shared secret between honest agent only, and this attack cannot be detected. Still, this should not be a serious problem, since it is indicative of an authentication attack, which should be found nevertheless.

If a role played by `A` shares a secret `T` with some player `U` of another role, and the identity of `U` is not accessible for `A` (e.g. because of anonymity), the predicate `secret(T,t,{U})` cannot be given in the role of `A`. In this case, it should be given in the role of `U` instead, right after the transition that sends `T` to `U` has been authenticated.

- **Authentication** is modelled by means of several goal predicates: `witness(A,B,id,T1)`, `request(B,A,id,T2)` and `wrequest(B,A,id,T3)`. The protocol designer should respect the following criteria:

  – suppose you want to express that agent `X`, playing role `AX`, (weakly) authenticates agent `Y`, playing role `AY`, on some information `T`; then it is expected that:

    ∗ in the HLPSL goal section, this property is written: `authentication_on id` (resp. `weak_authentication_on id`), where `id` is a label (of type `protocol_id`) for uniquely representing this goal;

    ∗ in role `AX`, agent `X` states a `request(X,Y,id,T1)` (resp. `wrequest(X,Y,id,T1)`) predicate in the right-hand side of some of its transitions;

    ∗ in role `AY`, agent `Y` states a `witness(Y,X,id,T2)` predicate in the right-hand side of some of its transitions.
    Note that `T1` and `T2` may be different terms but they should have the same value such that the two events match.

- the protocol ID that appears in the third position of `witness`, `request`, and `wrequest` facts *must* be declared of type `protocol_id` in a `const` declaration. For example,
  `authentication_on nb`
  should use `witness/request` facts with `nb` in the third slot and the following declaration should appear in the specification:
  `const nb: protocol_id`
- you should not use variables for protocol identifiers inside goal predicates; otherwise this would be impossible to run the analysis on one specific goal, one option of the AVISPA tool when used by an expert user. This means that for instance you should not write
  `witness(A,B,ID,Term)`
  even if the constant value `term` is passed to the role as value for the variable `ID`. You should write directly
  `witness(A,B,term,Term)`
  The same applies for `request`, `wrequest` and `secret`.

f. Transitions.

- In the left-hand side of a transition, primed variables should occur only inside a receiving channel and they will be assigned to the received message (or part of it). For instance, `Rcv(X')` means to assign `X'` to the value sent on the channel `Rcv`.
- Variables intended to be fresh must be written as primed in the right-hand side of a transition (assigned to `new()`) and they should not occur as primed in the left-hand side. For instance the following transition

```
1. State   = 0 /\ Rcv(start) =|>
   State' := 1 /\ Nb':=new() /\ Snd(Nb'.Text1)
```

  shows that `Nb'` is a fresh term.

g. Initial value.
In every role, a variable such that:

- it occurs inside the `local` declaration (therefore it does not occur in the parameter list of the role), and
- it is not of type `channel`, and
- there does not exist a left-hand side of a transition in which the variable occurs primed, and
- it is not assigned in a right-hand side of a transition with a fresh value,

should be given an **initial value**. For summarising, a local variable has to be initialised if its first use is unprimed.

h. Constants.

- In HLPSL it is not mandatory to declare the types for constants. However, not all the types of constants can be uniquely inferred by the translator. (For instance, suppose you declare a constant intended to be of type `text`, but without to specify its type and suppose you use such a constant only inside a message. Then the translator can only infer that this constant is of type `message`.) Hence, for more precise specifications, it is better to specify the type of each constant used.
- The type of a constant cannot be a compound type. For instance, suppose you declare:
  ```
  X : text.agent
  ```
  then you **cannot** declare
  ```
  x : text.agent
  ```
  and trying to instantiate `X` with `x`. What you should do is to declare two constants:
  ```
  x1 : text,
  x2 : agent
  ```
  and then you can instantiate `X` with `x1.x2`.

i. Messages.

- Please try to avoid variables of type `message`. For variables, please use compound types as much as possible. Namely:
  - do not use compound types when the variable is assigned with a term that makes use of algebraic equations;
  - use compound types in the other cases. E.g. when `Na_Nb` is a message that would represent a pair of texts, declare
    ```
    Na_Nb : text.text
    ```
    instead of
    ```
    Na_Nb : message
    ```
- When the form of the message is not important, use the type `protocol_id` instead of `message`. This is for instance the case in those protocols in which control messages like `Failure`, `Success`, etc. are sent over the channels. In this case it is useless to declare `Failure` and `Success` of type `message` since they will be used merely as constant messages instantiated in the topmost-level role. Please declare them of type `protocol_id`. For instance, in a protocol like
  ```
  A -> S: A, B, KeyRequest
  ```

```
S -> A: B, Kb
```

where `KeyRequest` is just a predefined constant control message for distinguishing between different server requests, it is useless to declare `KeyRequest` of type `message`. You can use a constant `keyrequest` of type `protocol_id` in the topmost-level role (e.g. `Environment`) and, accordingly, you can declare a variable `KeyRequest` of type `protocol_id` in the appropriate roles.

j. Knowledge.
The knowledge of an honest agent `A` playing the role `alice` is intended:

1. to contain all the parameters of the role `alice`,
2. to contain all the local variables of the role `alice`,
3. to be sufficient to execute all its transitions. For instance, if you declare for `alice` a transition:
   ```
   St=0 /\ RCV({M'}_Ka) =|> St':=1 /\ SND(M')
   ```
   then every time the event in the left-hand side is fired, then it is assumed that `A` has enough information to get `M'` (e.g. `A` may know the inverse key of `Ka`).

Suppose the intruder is playing the role `alice`, then the intruder's knowledge is supposed to contain all the terms given as parameter of the corresponding instance of the role `alice`. But this knowledge is not automatically given to the intruder; all **intruder's knowledge** will have to be declared as a set of terms in the `intruder_knowledge` declaration of roles. As a rule of thumb, the whole **intruder's knowledge** should be put in one single `intruder_knowledge` declaration in composition roles. In case there are more than one `intruder_knowledge` declarations (e.g. one per basic role), the total intruder knowledge is intended to be the union of the sets defined in those declarations.

k. Sessions generation.
Each HLPSL specification should have a special role, called `session` for example, which represents a single session of the protocol. This role is parametrised by all variables necessary for one session (BTW: channels can be declared as `local` variables inside a "session" role instead of being in the parameters list). For instance, in NSPK, a session might look like this:

```
role session(A, B: agent,
             Ka, Kb: public_key) def=
  composition
    alice(A,B,Ka,Kb) /\
    bob(A,B,Ka,Kb)
end role
```

2.1.3 Example As illustration of HLPSL, we describe in this section the specification of the well-known Needham-Schröder Public Key (NSPK) protocol. This example is usually considered as very simple and far away from real protocols. But here we will consider a more complex variant of the NSPK protocol: the NSPK Key Server (NSPK-KS). This protocol is given as follows, using an Alice&Bob-based notation:

$$
\begin{array}{l}
\textit{if } A \textit{ does not know } K_B, \\
\qquad A \rightarrow S \;:\; A, B \\
\qquad S \rightarrow A \;:\; \{B, K_B\}_{K_S^{-1}} \\
A \rightarrow B \;:\; \{N_A, A\}_{K_B} \\
\textit{if } B \textit{ does not know } K_A, \\
\qquad B \rightarrow S \;:\; B, A \\
\qquad S \rightarrow B \;:\; \{A, K_A\}_{K_S^{-1}} \\
B \rightarrow A \;:\; \{N_A, N_B\}_{K_A} \\
A \rightarrow B \;:\; \{N_B\}_{K_B}
\end{array}
$$

The main difference to NSPK is that agents A and B, needing to know the public key of each other for running the protocol, may ask the server S to supply the key if they do not already know it. This means that some steps of the protocol are conditional.

The specification is therefore decomposed into three basic roles: `alice`, `bob` and `server`. In addition, two composition roles are specified: `nspk` representing the classical composition of roles `alice` and `bob`, and `environment` representing the composition of several instances of `nspk` with one instance of `server`.

```
role alice (A, B: agent,
            Ka, Ks: public_key,
            KeyRing: (agent.public_key) set,
            SND, RCV: channel(dy))
played_by A def=

  local State : nat,
        Na, Nb: text,
        Kb: public_key

  init State := 0

  transition

   % Start, if alice must request bob's public key from key server
   ask.    State  = 0 /\ RCV(start) /\ not(in(B.Kb', KeyRing))
       =|> State':= 1 /\ SND(A.B)
   % Receipt of response from key server
   learn.  State  = 1 /\ RCV({B.Kb'}_inv(Ks))
       =|> State':= 0 /\ KeyRing':=cons(B.Kb', KeyRing)

   % Start/resume, provided alice already knows bob's public key
   knows.  State  = 0 /\ in(B.Kb', KeyRing)
       =|> State':= 4 /\ Na':=new() /\ SND({Na'.A}_Kb')
                      /\ secret(Na',na,{A,B})
                      /\ witness(A,B,bob_alice_na,Na')

   cont.   State  = 4 /\ RCV({Na.Nb'}_Ka)
       =|> State':= 6 /\ SND({Nb'}_Kb)
                      /\ request(A,B,alice_bob_nb,Nb')

end role
```

```
role bob(A, B: agent,
         Kb, Ks: public_key,
         KeyRing: (agent.public_key) set,
         SND, RCV: channel(dy))
played_by B def=

  local State: nat,
        Na, Nb: text,
        Ka: public_key

  init State := 2

  transition

   % Start if bob must request alice's public key from key server
   ask.    State  = 2 /\ RCV({Na'.A}_Kb) /\ not(in(A.Ka', KeyRing))
       =|> State':= 3 /\ SND(B.A)

   % Receipt of response from key server
   learn.  State  = 3 /\ RCV({A.Ka'}_inv(Ks))
       =|> State':= 2 /\ KeyRing':=cons(A.Ka', KeyRing)
   % Start/resume, provided if bob knows alice's public key
   knows.  State  = 2 /\ RCV({Na'.A}_Kb) /\ in(A.Ka', KeyRing)
       =|> State':= 5 /\ Nb':=new() /\ SND({Na'.Nb'}_Ka')
                      /\ secret(Nb',nb,{A,B})
                      /\ witness(B,A,alice_bob_nb,Nb')

   cont.   State  = 5 /\ RCV({Nb}_Kb)
       =|> State':= 7 /\ request(B,A,bob_alice_na,Na)

end role
```

```
role server(S: agent,
            Ks: public_key,
            KeyMap: (agent.public_key) set,
            SND, RCV: channel(dy))
played_by S def=

  local State : nat,
        A, B: agent,
        Kb: public_key

  init State := 8

  transition
   loop.   State  = 8 /\ RCV(A'.B') /\ in(B'.Kb', KeyMap)
       =|> State':= 8 /\ SND({B'.Kb'}_inv(Ks))

end role
```

```
% The role representing a partial session between alice and bob

role nspk(SND, RCV: channel(dy),
          Ks: public_key,
          Instances: (agent.agent.public_key.public_key) set,
          KeySet: agent -> (agent.public_key) set)
def=

  local A, B: agent,
        Ka, Kb: public_key

  composition
     /\_{in(A.B.Ka.Kb,Instances)}
       (alice(A,B,Ka,Ks,KeySet(A),SND,RCV)
       /\ bob(A,B,Kb,Ks,KeySet(B),SND,RCV))

end role
```

```
role environment() def=

  local KeyMap: (agent.public_key) set,
        SND, RCV: channel(dy)

  const a,b,s,i: agent,
        ka, kb, ki, ks: public_key,
        na, nb, alice_bob_nb, bob_alice_na: protocol_id

  init KeyMap := {a.ka, b.kb, i.ki}

  intruder_knowledge = {a, b, ks, ka, kb, ki, inv(ki)}

  composition
        server(s,ks, KeyMap, SND, RCV)
     /\ nspk(SND, RCV,                  % channels
             ks,                        % public key of server
             {a.b.ka.kb,                % session instances
              a.i.ka.ki},
             {a.{a.ka,b.kb},            % initial KeyRings
              b.{b.kb},
              i.{i.ki}})
end role
```

```
goal

  secrecy_of na, nb
  authentication_on alice_bob_nb
  authentication_on bob_alice_na

end goal
```

```
environment()
```

2.2 Analyzing a HLPSL Specification

Analyzing a protocol is entirely automatic: once the HLPSL specification has been written, the script `avispa` can be used to evaluate it, and will print the result of the analysis.
For a basic use of this script, you do not need to know many options. For a more expert use, please read the Advanced User Section (Section 3).

The AVISPA tool is composed of several modules:

- a translator for transforming HLPSL specifications (written by the user) to IF specifications (intermediate format, see Section 3.1.2), called `hlpsl2if` (see Section 3.1.1);
- four different verification tools (back-ends) that can analyze IF specifications:
  - CL-AtSe: the Constraint-Logic-based Attack Searcher (see Section 3.2.1);
  - OFMC: the On-the-Fly Model-Checker (see Section 3.2.2);
  - SATMC: the SAT-based Model-Checker (see Section 3.2.3);
  - TA4SP: the Tree Automata tool based on Automatic Approximations for the Analysis of Security Protocols (see Section 3.2.4).

2.2.1 Running `avispa`

The syntax of the `avispa` command is one of the following:

```
avispa [OPTION]
avispa FILE [OPTIONS] [MODULE [MOPTIONS]]
```

where:

- `OPTION` can be:

`-h`, `--help`	display this help and exit.
`-v`, `--version`	output version information and exit.

- `OPTIONS` can be:

`--typed_model=TM`	IF specifications can be generated both in an untyped variant (set `TM` to `no`), in a typed variant (set `TM` to `yes`), and in a strongly-typed one (set `TM` to `strongly`). (Default: `yes`)
`--output=DIR`	set the output directory to `DIR`. Namely the IF file, and any other file generated by the selected back-end will be written in such a directory. (Default: `$AVISPA_PACKAGE/testsuite/results`)
`--nowarnings`	do not display warnings in executing the `hlpsl2if` translator.

- `MODULE`: selects a specific AVISPA sub-module. Accepted values are:

`--hlpsl2if`	runs only the translator.
`--ofmc`	runs the translator (no particular option), then OFMC.
`--satmc`	runs the translator (no particular option), then SATMC.
`--cl-atse`	runs the translator (no particular option), then CL-AtSe.
`--ta4sp`	runs the translator (no particular option), then TA4SP.

  (Default: `--ofmc`)

- `MOPTIONS` are the options given to the selected AVISPA sub-module.
  (Default: no particular option)

- `FILE` is the HLPSL file to be analyzed.

Not all combination of the above settings are allowed. The allowed combinations are indicated in the following table:

	`--typed_model=`		
	`yes`	`no`	`strongly`
`--ofmc`	$\surd$	$\surd$	
`--satmc`	$\surd$		$\surd$
`--cl-atse`	$\surd$	$\surd$	
`--ts4sp`	$\surd$		

All options for backends are detailed in the files `cl.bash`, `ofmc.bash`, `satmc.bash` and `ta4sp.bash`, in the directory `$AVISPA_PACKAGE/bin/backends/`. For some backends, a file (`satmc.config`, `ta4sp.config`) lists the default options.

2.2.2 Generated Errors If some errors are detected during the execution of the `avispa` script, a message will indicate the encountered problem. The most standard errors are:

- HLPSL specification problems: the name of a log file is given (usually in the directory `$AVISPA_PACKAGE/logs`); this file contains information about the location and the cause of the errors;
- Back-end execution problems: the printed message should be explicit.

2.2.3 Interpreting the Output When the analysis of a protocol has been successful (by finding an attack or not), the output describes precisely what is the result, and under what conditions it has been obtained.

The first printed section is `SUMMARY`; it indicates if the protocol is safe, unsafe, or if the analysis is inconclusive.

In any case, a second section titled `DETAILS` will either explain under what conditions the protocol is declared safe, or what conditions have been used for finding an attack, or finally why the analysis was inconclusive.

The next sections, `PROTOCOL`, `GOAL` and `BACKEND` recall the name of the protocol, the goal of the analysis and the name of the back-end used, respectively.

After some possible comments and statistics, the trace of the attack (if any) is printed in an Alice&Bob notation.

More information on the format of the output produced by the AVISPA Tool is given in Section 3.3.

2.2.4 Example For running the AVISPA tool on the NSPK Key Server protocol, with the back-end CL-AtSe, the command is:

```
avispa NSPK-KS.hlpsl --cl-atse -ns -lr
```

Note that we have given two options to CL-AtSe: `-ns` for no simplifications (the output prints all the steps); `-lr` for getting one of the shortest attacks.

The output is listed hereafter, and shows that a secrecy attack has been found on this protocol.

```
SUMMARY
  UNSAFE

DETAILS
  ATTACK_FOUND
  TYPED_MODEL
  BOUNDED_SEARCH_DEPTH

PROTOCOL
  NSPK-KS.if

GOAL
  Secrecy attack on (n23Nb)

BACKEND
  CL-AtSe

STATISTICS
  Analysed   : 2660 states
  Reachable  : 1567 states
  Translation: 0.10 seconds
  Computation: 4.50 seconds

ATTACK TRACE
  i -> (a.6):  start               & TestNotInSet({i,X32Kb}_(set_93))
  (a.6) -> i:  a,i
  i -> (s.2):  X1A,i
  (s.2) -> i:  {i,ki}_(inv(ks))    & TestInSet({i,ki}_(set_91))
  i -> (s.2):  X2A,a
  (s.2) -> i:  {a,ka}_(inv(ks))    & TestInSet({a,ka}_(set_91))
  i -> (a.6):  {i,ki}_(inv(ks))
  (a.6) -> i:  {n33Na,a}_(ki)      & Secret(n33Na,set_124)
                                   & AddToSet({i,ki}_(set_93))
                                   & AddToSet({a}_(set_124))
                                   & AddToSet({i}_(set_124))
  i -> (b.5):  {n33Na,a}_(kb)      & TestNotInSet({a,X22Ka}_(set_94))
  (b.5) -> i:  b,a
  i -> (b.5):  {a,ka}_(inv(ks))
  (b.5) -> i:  {n33Na,n23Nb}_(ka)  & Secret(n23Nb,set_117)
                                   & Witness(b,a,nb,n23Nb)
                                   & AddToSet({a,ka}_(set_94))
```

```
                                  & AddToSet({a}_(set_117))
                                  & AddToSet({b}_(set_117))
    i -> (a.6):  {n33Na,n23Nb}_(ka)
    (a.6) -> i:  {n23Nb}_(ki)
```

The description of the attack is not very difficult to understand, but a detailed study of the output of the AVISPA tool is given in Section 3.3.

3 Advanced User Section

AVISPA users who want to do more than just using the `avispa` script file will find in the following sections important details, guiding them for exploiting all the possible options at each step of the AVISPA tool architecture (Figure 2).

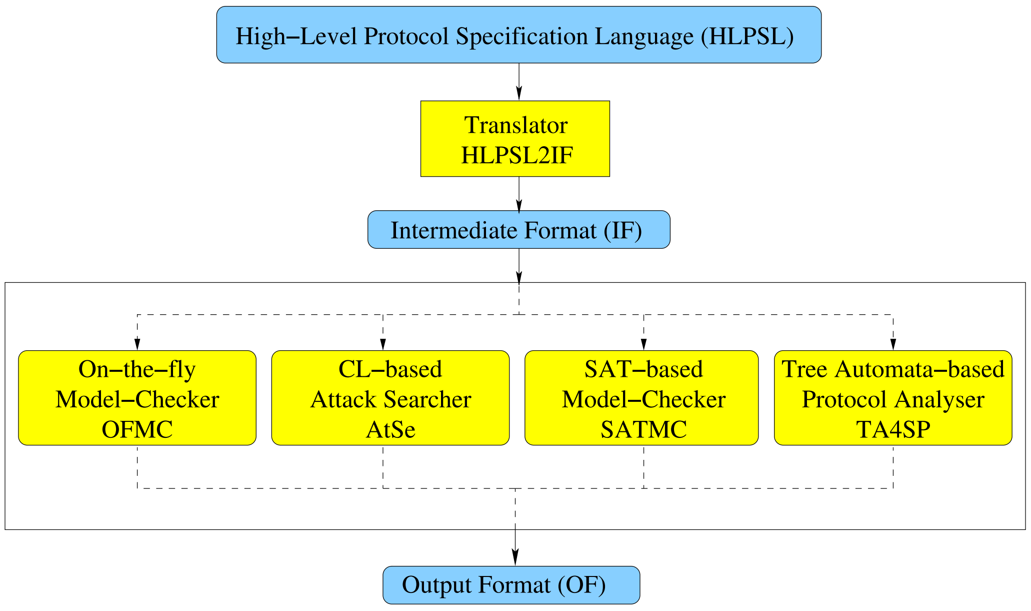

High−Level Protocol Specification Language (HLPSL)
Translator HLPSL2IF
Intermediate Format (IF)
On−the−fly Model−Checker OFMC
CL−based Attack Searcher AtSe
SAT−based Model−Checker SATMC
Tree Automata−based Protocol Analyser TA4SP
Output Format (OF)

Figure 2: Architecture of the AVISPA tool v.1.0

3.1 Generating an IF Specification

Given a HLPSL specification written in a file with extension `hlpsl`, the first step is to translate it into a lower level specification. This is automatically done by the translator `hlpsl2if`, generating a specification in an intermediate format, IF. The resulting file has the same name, with extension `if`.

3.1.1 Automatic Translation from HLPSL to IF

This section lists the possible parameters of the `hlpsl2if` translator. `hlpsl2if` is a translator that maps security protocol specifications written in HLPSL into rewriting systems written in IF. It is written in Objective Caml which is an implementation of the ML language, based on the Caml Light dialect.

```
USAGE:
  hlpsl2if [options] [file.hlpsl]
```

```
Options:
  --types       Print identifiers and their types
  --init        Print initial state
  --rules       Print protocol rules
  --goals       Print goals
  --all         Print everything (default)
  --split       Split goals in different IF files
  --stdout      Print on the standard output
  --output dir  Set the output directory (default: same as input)
  --nowarnings  Do not display warnings
  -help         Display this list of options
  --help        Display this list of options
```

3.1.2 The IF Specification Language

In the following we give the entire BNF of the IF (with the usual conventions).

a. Lexical entities. In IF, the conventions are the same as in HLPSL: all variables start with a capital letter, and constants start with a small letter; natural numbers can also be used as constants (without any specific interpretation).

```
var_ident: [A-Z][A-Za-z0-9_]*
const_ident: [a-z][A-Za-z0-9_]*
nat_ident: [0-9]+
```

Comments start by the symbol `%`, and finish at the end of the line.

b. Prelude and IF files. The grammar has two start symbols, `Prelude` and `IF_File`. The first one permits to describe the pre-defined file `prelude.if`, containing general information about protocols specifications in IF: available types, super types, signature of functions and predicates, variables and constants declaration, equations, and the intruder behaviour. For more information on this file, see Section 3.1.3.
The second start symbol corresponds to the description of a protocol specification in IF: signature of role states, variables and constants declaration, initialisation, transition rules, properties to satisfy, and attack states to reach.

```
Prelude ::=
  TypeSymbolsSection
  SignatureSection
  TypesSection
```

```
    EquationsSection
    IntruderSection

  IF_File ::=
    SignatureSection
    TypesSection
    InitsSection
    RulesSection
    PropertiesSection
    AttackStatesSection
```

c. Section for type symbols. This section contains the list of constant names representing the basic types available, such as `agent`, `public_key`, `symmetric_key`, ...

```
  TypeSymbolsSection ::=
    "section typeSymbols:" ConstantList

  ConstantList ::=
    const_ident ("," const_ident)*
```

d. Section for signature. This section is decomposed in three parts: the declaration of super types (an ordered precedence between some types); the declaration of function symbols; the declaration of predicate symbols

```
  SignatureSection ::=
    "section signature:" SignatureSection0

  SignatureSection0 ::=
    SuperTypeDeclaration*
  | FunctionDeclaration*
  | PredicateDeclaration*
```

One super type (`message` for example) is very useful for avoiding to have as many copies of functions and predicates as there are types.

```
  SuperTypeDeclaration ::=
    IF_Type ">" IF_Type

  IF_Type ::=
    const_ident
```

The type of a function is declared in a very standard way. The only difference with the type of a predicate is that the result of a predicate is of type `fact`.

```
FunctionDeclaration ::=
  IF_Operator ":" TypeStar "->" Type

PredicateDeclaration ::=
  IF_Operator ":" TypeStar "->" "fact"

IF_Operator ::=
  const_ident

TypeStar ::=
  Type
| Type "*" TypeStar

Type ::=
  IF_Type
| IF_Operator "(" TypeList ")"
| "{" ConstantNatList "}"

TypeList ::=
  Type ("," Type)*

ConstantNatList ::=
  (const_ident | nat_ident) ("," (const_ident | nat_ident))*
```

Note that types may be compound, using functions representing pairing, encryption, ...

e. Section for variables and constants declaration. Variables and constants are all declared using types and functions declared in the prelude file.

```
TypesSection ::=
  "section types:" TypeDeclaration*

TypeDeclaration ::=
  AtomicTermList ":" Type

AtomicTermList ::=
  AtomicTerm ("," AtomicTerm)*
AtomicTerm ::=
  const_ident
| nat_ident
| var_ident
```

f. Section for equations. This section represents the equational theory that has to be considered for some specific function operators, such as `pair`, `xor` and `exp`.

```
EquationsSection ::=
 "section equations:" Equation*

Equation ::=
  Term "=" Term

Term ::=
  AtomicTerm
| ComposedTerm

ComposedTerm ::=
  IF_Operator "(" TermList ")"

TermList ::=
  Term ("," Term)*
```

g. Section for initialisation. This section contains all the initial information needed before starting the analysis of the protocol: the intruder initial knowledge, the instantiated roles states, and some additional information needed for the two previous information.

```
InitsSection ::=
  "section inits:" ("initial_state" const_ident ":=" State)+

State ::=
  Fact ("." Fact)*

Fact ::=
  IF_Fact "(" TermList ")"

IF_Fact ::=
  "state_"const_ident
| const_ident
```

h. Section for transition rules. Contrarily to a HLPSL specification, all the transitions of the protocol are listed in the same section, independently of the concerned role.

```
RulesSection ::=
  "section rules:" RuleDeclaration*
```

A transition starts with a unique label and the list of variables involved; the left-hand side is a list of facts (containing mostly a role state fact), plus maybe some positive or negative conditions; the right-hand side is a list of facts.

```
RuleDeclaration ::=
  "step" const_ident "(" VariableList ")" ":="
  CState ExistsVar? "=>" State

CState ::=
  State ConditionList

ConditionList ::=
  ("&" Condition)*

Condition ::=
  PositiveCondition
| NegativeCondition

PositiveCondition ::=
  "equal" "(" Term "," Term ")"
| "leq" "(" Term "," Term ")"

NegativeCondition ::=
  "not" "(" Condition ")"
| "not" "(" Fact ")"

ExistsVar ::=
  "=[exists" VariableList "]"

VariableList ::=
  var_ident ("," var_ident)*
```

Variables that have to receive a fresh value for this transition are listed in the implication, after the `exists` keyword.

i. Section for properties. This section contains properties described by LTL formulas. They have to be satisfied all along the execution trace, provided the analysers can handle them.
Those properties come either directly from LTL formulas written in the HLPSL specification, or from the HLPSL macro goals for secrecy and authentication.

```
PropertiesSection ::=
  "section properties:" PropertyDeclaration*

PropertyDeclaration ::=
  "property" PropertyID "(" VariableList ")" ":="
  "[]" LTL_Formula

PropertyID ::=
  AttackStateID
| "ltl_"nat_ident

LTL_Formula ::=
  LTL_predicate
| "<->" LTL_formula
| "(-)" LTL_formula
| "[-]" LTL_formula
| LTL_formula "/\" LTL_formula
| LTL_formula "\/" LTL_formula
| LTL_formula "=>" LTL_formula
| "~" LTL_formula
| "(" LTL_formula ")"

LTL_predicate ::=
  Fact
| PositiveCondition
```

In such temporal formulas, "`<->`" means “sometimes in the past”, "`(-)`" means “one time instant in the past”, "`[-]`" means “globally in the past”. The other symbols are standard logical connectives (conjunction, disjunction, implication and negation, respectively).

j. Section for attack states. This is a second goal section, containing descriptions of states representing attacks. The contents of this section comes from the HLPSL macro goals for authentication and secrecy.

```
AttackStatesSection ::=
  "section attack_states:" AttackStateDeclaration*
```

```
AttackStateDeclaration ::=
  "attack_state" AttackStateID "(" VariableList ")" ":=" CState

AttackStateID ::=
  "secrecy_of_"const_ident
| "authentication_on_"const_ident
| "weak_authentication_on_"const_ident
```

k. Section for intruder behaviour. This section contains the description of the intruder behaviour, represented by transition rules.

```
IntruderSection ::=
  "section intruder:" RuleDeclaration*
```

In the current version of the AVISPA tool, this section is unique because only the Dolev-Yao model is supported. In a close future, it should be parametrised by the model to consider.

3.1.3 IF Prelude File The IF prelude file (`prelude.if`) contains some information about the IF syntax for back-ends. It also contains the description of equational properties of some operators (such as `pair`, `exp` and `xor`) and the intruder's behaviour (in the Dolev-Yao model).

```
section typeSymbols:

  agent, text, symmetric_key, public_key, hash_func,
  message, fact, nat, protocol_id, bool, set

section signature:

   message > agent
   message > text
   message > symmetric_key
   message > public_key
   message > hash_func
   message > nat
   message > protocol_id
   message > bool
   message > set

  % concatenation: pair(Msg1,Msg2)
  pair     : message * message -> message
  % asymmetric encryption: crypt(Key,Message)
  crypt    : message * message -> message
  % inverse of a public key (=private key): inv(Key)
  inv      : message -> message
  % symmetric encryption: scrypt(Key,Message)
  scrypt   : message * message -> message
  % exponentiation: exp(Base,Exponent)
  exp      : message * message -> message
  % exclusive or: xor(N1,N2)
  xor      : message * message -> message
  % application of a hash function: apply(F,Arg)
  apply    : message * message -> message

  % intruder knowledge: iknows(ki)
  iknows   : message -> fact
  % set element: contains(a,set_1)
  contains : message * set -> fact
  % witness for authentication: witness(A,B,id,Msg)
  witness  : agent * agent * protocol_id * message  -> fact
  % request for strong authentication: request(B,A,id,Msg,RoleID)
  request  : agent * agent * protocol_id * message * nat -> fact
  % request for weak authentication: wrequest(B,A,id,Msg,RoleID)
  wrequest : agent * agent * protocol_id * message * nat -> fact
  % secrecy: secret(Msg,id,AgentSet)
  secret   : message * protocol_id * set(agent) -> fact

section types:

  % declaration of the pre-defined constants:
  true, false: bool
  % declaration of the variables used in this file:
  PreludeK,PreludeM,PreludeM1,PreludeM2,PreludeM3 : message

section equations:

  % associativity of concatenation:
  pair(PreludeM1,pair(PreludeM2,PreludeM3))
    = pair(pair(PreludeM1,PreludeM2),PreludeM3)

  % identity of double inverse:
  inv(inv(PreludeM)) = PreludeM

  % commutation of exponents:
  exp(exp(PreludeM1,PreludeM2),PreludeM3) = exp(exp(PreludeM1,PreludeM3),PreludeM2)
  % cancellation of inverse exponents:
  exp(exp(PreludeM1,PreludeM2),inv(PreludeM2)) = PreludeM1

  % associativity of xor:
  xor(PreludeM1,xor(PreludeM2,PreludeM3)) = xor(xor(PreludeM1,PreludeM2),PreludeM3)
  % commutativity of xor:
  xor(PreludeM1,PreludeM2) = xor(PreludeM2,PreludeM1)
  % nilpotency of xor:
  xor(xor(PreludeM1,PreludeM1),PreludeM2) = PreludeM2

section intruder:  % for the Dolev-Yao model

  % generating rules:
  step gen_pair (PreludeM1,PreludeM2) :=
    iknows(PreludeM1).iknows(PreludeM2) => iknows(pair(PreludeM1,PreludeM2))
  step gen_crypt (PreludeM1,PreludeM2) :=
    iknows(PreludeM1).iknows(PreludeM2) => iknows(crypt(PreludeM1,PreludeM2))
  step gen_scrypt (PreludeM1,PreludeM2) :=
    iknows(PreludeM1).iknows(PreludeM2) => iknows(scrypt(PreludeM1,PreludeM2))
  step gen_exp (PreludeM1,PreludeM2) :=
    iknows(PreludeM1).iknows(PreludeM2) => iknows(exp(PreludeM1,PreludeM2))
  step gen_xor (PreludeM1,PreludeM2) :=
    iknows(PreludeM1).iknows(PreludeM2) => iknows(xor(PreludeM1,PreludeM2))
  step gen_apply (PreludeM1,PreludeM2) :=
    iknows(PreludeM1).iknows(PreludeM2) => iknows(apply(PreludeM1,PreludeM2))

  % analyzing rules:
  step ana_pair (PreludeM1,PreludeM2) :=
    iknows(pair(PreludeM1,PreludeM2)) => iknows(PreludeM1).iknows(PreludeM2)
  step ana_crypt (PreludeK,PreludeM) :=
    iknows(crypt(PreludeK,PreludeM)).iknows(inv(PreludeK)) => iknows(PreludeM)
  step ana_scrypt (PreludeK,PreludeM) :=
    iknows(scrypt(PreludeK,PreludeM)).iknows(PreludeK) => iknows(PreludeM)

  % generating fresh constants of any type:
  step generate (PreludeM) :=
    =[exists PreludeM]=> iknows(PreludeM)
```

3.1.4 Example The IF specification given in the following has been automatically generated from the HLPSL specification of the Needham-Schröder Public Key Protocol with Key Server (Section 2.1.3).

Note that in the initial state, arguments of a role state that have not been initialised are assigned to a generic constant, for example `dummy_agent` for an argument of type `agent`.

In the transition rules, in general, when an argument of a role state is modified, its old value (in the left-hand side) is denoted with a generic name, for example `Dummy_Na` if the argument is called `Na`.

The description of sets is given by a list of `contains` facts, one per element. The set itself is identified by a constant (`set_94` for example). Reading, adding or removing elements in a set is possible only if the set identifier is known.

```
%% IF specification of NSPK-KS.hlpsl

section signature:

state_bob: agent * agent * public_key * public_key
  * set(pair(agent,public_key)) * nat * text * text * public_key
  * set(agent) * set(agent) * nat -> fact
state_alice: agent * agent * public_key * public_key
  * set(pair(agent,public_key)) * nat * text * text * public_key
  * set(agent) * set(agent) * nat -> fact
state_server: agent * public_key * set(pair(agent,public_key))
  * agent * agent * public_key * nat -> fact


section types:

MGoal, start: message
snb, nb, na, sna: protocol_id
ASGoal, Set_45, Set_44, Set_27, Set_23: set(agent)
set_94, set_93, set_91, KeyMap, KeyRing: set(pair(agent,public_key))
Na, Nb, Dummy_Nb, Dummy_Na, dummy_nonce: text
set_124, set_123, set_117, set_116, set_106, set_105: set
dummy_pk, ka, kb, ki, ks, Kb, Ks, Ka, Dummy_Ka, Dummy_Kb: public_key
dummy_agent, A2Goal, A1Goal, a, b, s, A, B, S, Dummy_B, i, Dummy_A: agent
SID, SID2, SID1, 3, 1, 6, 2, 5, 0, 4: nat


section inits:
```

```
initial_state init1 :=
 iknows(start).
 iknows(ki).
 iknows(inv(ki)).
 iknows(a).
 iknows(b).
 iknows(ks).
 iknows(ka).
 iknows(kb).
 iknows(i).
 state_server(s,ks,set_91,dummy_agent,dummy_agent,dummy_pk,2).
 state_alice(a,b,ka,ks,set_93,0,dummy_nonce,dummy_nonce,dummy_pk,
   set_105,set_106,4).
 state_bob(b,a,kb,ks,set_94,0,dummy_nonce,dummy_nonce,dummy_pk,
   set_116,set_117,5).
 state_alice(a,i,ka,ks,set_93,0,dummy_nonce,dummy_nonce,dummy_pk,
   set_123,set_124,6).
 contains(pair(a,ka),set_91).
 contains(pair(b,kb),set_91).
 contains(pair(i,ki),set_91).
 contains(pair(a,ka),set_93).
 contains(pair(b,kb),set_93).
 contains(pair(b,kb),set_94)


section rules:

step step_0 (S,Ks,KeyMap,Dummy_A,Dummy_B,Dummy_Kb,SID,A,B,Kb) :=
 state_server(S,Ks,KeyMap,Dummy_A,Dummy_B,Dummy_Kb,SID).
 iknows(pair(A,B)).
 contains(pair(B,Kb),KeyMap)
 =>
 state_server(S,Ks,KeyMap,A,B,Kb,SID).
 iknows(crypt(inv(Ks),pair(B,Kb))).
 contains(pair(B,Kb),KeyMap)

step step_1 (A,B,Ka,Ks,KeyRing,Dummy_Na,Nb,Dummy_Kb,Set_23,Set_27,SID,Na,Kb) :=
 state_alice(A,B,Ka,Ks,KeyRing,0,Dummy_Na,Nb,Dummy_Kb,Set_23,Set_27,SID).
 iknows(start).
 contains(pair(B,Kb),KeyRing)
=[exists Na]=>
 state_alice(A,B,Ka,Ks,KeyRing,2,Na,Nb,Kb,Set_23,Set_27,SID).
 iknows(crypt(Kb,pair(Na,A))).
 witness(A,B,na,Na).
 secret(Na,sna,Set_23).
 contains(A,Set_23).
 contains(B,Set_23).
 contains(pair(B,Kb),KeyRing)

step step_2 (A,B,Ka,Ks,KeyRing,Na,Nb,Dummy_Kb,Set_23,Set_27,SID) :=
 state_alice(A,B,Ka,Ks,KeyRing,0,Na,Nb,Dummy_Kb,Set_23,Set_27,SID).
 iknows(start) &
 not(contains(pair(B,Kb),KeyRing))
 =>
 state_alice(A,B,Ka,Ks,KeyRing,1,Na,Nb,Dummy_Kb,Set_23,Set_27,SID).
 iknows(pair(A,B))

step step_3 (A,B,Ka,Ks,KeyRing,Dummy_Na,Nb,Dummy_Kb,Set_23,Set_27,SID,Na,Kb) :=
 state_alice(A,B,Ka,Ks,KeyRing,1,Dummy_Na,Nb,Dummy_Kb,Set_23,Set_27,SID).
 iknows(crypt(inv(Ks),pair(B,Kb)))
=[exists Na]=>
 state_alice(A,B,Ka,Ks,KeyRing,2,Na,Nb,Kb,Set_23,Set_27,SID).
 iknows(crypt(Kb,pair(Na,A))).
 witness(A,B,na,Na).
 secret(Na,sna,Set_27).
 contains(pair(B,Kb),KeyRing).
 contains(A,Set_27).
 contains(B,Set_27)

step step_4 (A,B,Ka,Ks,KeyRing,Na,Dummy_Nb,Kb,Set_23,Set_27,SID,Nb) :=
 state_alice(A,B,Ka,Ks,KeyRing,2,Na,Dummy_Nb,Kb,Set_23,Set_27,SID).
 iknows(crypt(Ka,pair(Na,Nb)))
 =>
 state_alice(A,B,Ka,Ks,KeyRing,3,Na,Nb,Kb,Set_23,Set_27,SID).
 iknows(crypt(Kb,Nb)).
 request(A,B,nb,Nb,SID)

step step_5 (B,A,Kb,Ks,KeyRing,Dummy_Na,Dummy_Nb,Dummy_Ka,Set_44,Set_45,
             SID,Na,Nb,Ka) :=
 state_bob(B,A,Kb,Ks,KeyRing,0,Dummy_Na,Dummy_Nb,Dummy_Ka,Set_44,Set_45,SID).
 iknows(crypt(Kb,pair(Na,A))).
 contains(pair(A,Ka),KeyRing)
=[exists Nb]=>
 state_bob(B,A,Kb,Ks,KeyRing,2,Na,Nb,Ka,Set_44,Set_45,SID).
 iknows(crypt(Ka,pair(Na,Nb))).
 witness(B,A,nb,Nb).
 secret(Nb,snb,Set_44).
 contains(A,Set_44).
 contains(B,Set_44).
 contains(pair(A,Ka),KeyRing)

step step_6 (B,A,Kb,Ks,KeyRing,Dummy_Na,Nb,Dummy_Ka,Set_44,Set_45,SID,Na) :=
 state_bob(B,A,Kb,Ks,KeyRing,0,Dummy_Na,Nb,Dummy_Ka,Set_44,Set_45,SID).
 iknows(crypt(Kb,pair(Na,A))) &
 not(contains(pair(A,Ka),KeyRing))
 =>
 state_bob(B,A,Kb,Ks,KeyRing,1,Na,Nb,Dummy_Ka,Set_44,Set_45,SID).
 iknows(pair(B,A))

step step_7 (B,A,Kb,Ks,Dummy_KeyRing,Na,Dummy_Nb,Dummy_Ka,Set_44,Set_45,
             SID,Nb,Ka) :=
 state_bob(B,A,Kb,Ks,KeyRing,1,Na,Dummy_Nb,Dummy_Ka,Set_44,Set_45,SID).
 iknows(crypt(inv(Ks),pair(A,Ka)))
=[exists Nb]=>
 state_bob(B,A,Kb,Ks,KeyRing,2,Na,Nb,Ka,Set_44,Set_45,SID).
 iknows(crypt(Ka,pair(Na,Nb))).
 witness(B,A,nb,Nb).
 secret(Nb,snb,Set_45).
 contains(pair(A,Ka),KeyRing).
 contains(A,Set_45).
 contains(B,Set_45)

step step_8 (B,A,Kb,Ks,KeyRing,Na,Nb,Ka,Set_44,Set_45,SID) :=
 state_bob(B,A,Kb,Ks,KeyRing,2,Na,Nb,Ka,Set_44,Set_45,SID).
 iknows(crypt(Kb,Nb))
 =>
 state_bob(B,A,Kb,Ks,KeyRing,3,Na,Nb,Ka,Set_44,Set_45,SID).
 wrequest(B,A,na,Na,SID)


section properties:
```

```
property authentication_on_nb (A1Goal,A2Goal,MGoal,SID,SID1,SID2) :=
 [] (((request(A1Goal,A2Goal,nb,MGoal,SID) /\ ~ equal(A2Goal,i))
      => witness(A2Goal,A1Goal,nb,MGoal))
     /\ ((request(A1Goal,A2Goal,nb,MGoal,SID1)
          /\ request(A1Goal,A2Goal,nb,MGoal,SID2)
          /\ ~ equal(A2Goal,i))
         => equal(SID1,SID2)))

property weak_authentication_on_na (A1Goal,A2Goal,MGoal,SID) :=
 [] ((wrequest(A1Goal,A2Goal,na,MGoal,SID) /\ ~ equal(A2Goal,i))
     => witness(A2Goal,A1Goal,na,MGoal))

property secrecy_of_sna (MGoal,ASGoal) :=
 [] ((secret(MGoal,sna,ASGoal) /\ iknows(MGoal))
     => contains(i,ASGoal))

property secrecy_of_snb (MGoal,ASGoal) :=
 [] ((secret(MGoal,snb,ASGoal) /\ iknows(MGoal))
     => contains(i,ASGoal))


section attack_states:

attack_state authentication_on_nb (A1Goal,A2Goal,MGoal,SID) :=
 request(A1Goal,A2Goal,nb,MGoal,SID) &
 not(witness(A2Goal,A1Goal,nb,MGoal)) &
 not(equal(A2Goal,i))
attack_state replay_protection_on_nb (A2Goal,A1Goal,MGoal,SID1,SID2) :=
 request(A1Goal,A2Goal,nb,MGoal,SID1).
 request(A1Goal,A2Goal,nb,MGoal,SID2) &
 not(equal(SID1,SID2)) &
 not(equal(A2Goal,i))

attack_state weak_authentication_on_na (A1Goal,A2Goal,MGoal,SID) :=
 wrequest(A1Goal,A2Goal,na,MGoal,SID) &
 not(witness(A2Goal,A1Goal,na,MGoal)) &
 not(equal(A2Goal,i))

attack_state secrecy_of_sna (MGoal,ASGoal) :=
 iknows(MGoal).
 secret(MGoal,sna,ASGoal) &
```

```
   not(contains(i,ASGoal))

  attack_state secrecy_of_snb (MGoal,ASGoal) :=
   iknows(MGoal).
   secret(MGoal,snb,ASGoal) &
   not(contains(i,ASGoal))
```

3.2 Analysing a IF Specification

This section contains the description of the use of each back-end provided by the AVISPA tool.

3.2.1 Using `CL-AtSe` The CL-based Model-Checker (CL-Atse) provides a translation from any security protocol specification written as transition relation in the IF, into a set of constraints which can be effectively used to find attacks to protocols. Both translation and checking are fully automatic and internally performed by CL-Atse, i.e. no external tool is used.

In this approach, each protocol step is modeled by constraints on the adversary's list of knowledges. For example, a message received by an honest participant is a forgeability constraint for the adversary. Conditions like equality, inequality, element or non-element of a list are also constraints. To interpret the IF transition relation, each role is partially pre-executed to extract an exact and minimal list of constraints modeling it. The participants's states and knowledges are eliminated thanks to the use of global variables. Any protocol step is executed by adding new constraints to the system and reduce/eliminate other constraints accordingly. Finally, at each step the system state is tested against the provided set of security properties.

The analyze algorithm used by CL-Atse is designed for a bounded number of loops, i.e. a bounded number of protocol steps in any trace. That is, if the protocol specification is loop-free, then the whole specification is analyzed, otherwise the user must provide an integer bound on the maximal number of loop iterations. With a bounded number of loop iterations, the search for attacks is correct and complete, and corresponds to an optimized deterministic implementation of the NP-Completeness result from [14].

While reading the IF file, CL-Atse tries by default to simplify the protocol specification. The goal of that is to reduce the total number of protocol steps that need to be checked. Since most of the execution time is consumed in testing all possible interleaving of the protocol steps, this simplification can be very important for the biggest protocols. The idea is to identify and mark the protocol steps that can be executed as late, or as soon, as possible. This information is then used to reduce the step interleaving.

CL-Atse is able to take advantage of the algebraic properties of the XOR operator, and most properties of the exponential. In order to model algebraic properties of certain operators, CL-Atse implements a variant of the Baader and Shoultz unification algorithm, optimized for XOR and usable for the exponential.

Finally, CL-Atse tries to produce nice human-readable attack description (when one is found). In particular, choice points are identified by keyword and step indent; both a short and detailed attack are given; and the simplified protocol specification analyzed by CL-Atse is provided on demand. The user can also compare the simplified and non-simplified protocol specification by using the "-ns" and "-noexec" options.

Usage. CL-Atse can be invoked by typing on the command-line

```
cl-atse <filename> [-nb n] [-v] [-ns] [-noexec] [-notype]
                   [-short] [-light] [-out] [-help]
```

in any order, where each option is described as follow:

- `nb n`: Maximum number of loop iteration in any trace. Only used when the protocol specification contains loops. The default is 3.
- `v`: Verbose mode. In this mode, CL-Atse displays the analyzed (maybe simplified) protocol specification and a bit more details in the attack trace. Otherwise, CL-Atse only displays a compact attack trace with the summary.
- `ns`: No simplification. Not recommended, unless used with `noexec`. With this option, the simplification step is skipped.
- `noexec`: Do not analyze the protocol. Usually only used to see the protocol specification (`-v`) without spending time in the analysis.
- `notype`: No type mode. Is this mode, CL-Atse considers all variables to be of generic type (or "no type"), i.e. the typing constraints described in the IF file are not checked. This option is very useful to discover type-flaw attacks.
- `light`: Force the use of non-algebraic pairing operator in the unification algorithm. Otherwise, the unification algorithm tries to consider the associativity property of the pairing.
- `short`: Ask CL-AtSe to output a minimal attack if possible, i.e. an attack with a minimal number of protocol steps. Same effect as `-lr`. It usually need more memory, and a bit more cpu time, that finding any attack. It performs a breath-first search instead of depth-first in the protocol step interleaving tree.

  (*
- `td`: Perform depth-first search in the protocol step interleaving tree. (By default, uses less memory)

- `lr`: Perform breath-first search in the step interleaving tree. (Find minimal attacks at the cost of memory use.) Usually slower than `-td` due to a lot of memory access, garbage collector, etc... *)
- `out`: Write the attack to "filename.atk" instead of using stdout.
- `dir d`: Chose `d` as the output directory.
- `help`: Write a summary of all options, including debugging options.

The cl-atse verbose output format Using the *-v* or *-noexec* option(s) of cl-atse, we get a detailed view of the starting point of cl-atse. This is called the "Initial System State" in the output, and represents both the intruder and honest participant's states in cl-atse just after reading (and interpreting) the if file. While the intruder state is just represented by a list of knowledges ("Intruder knowledge:" line), the honest participants are described by a set of instantiated roles, so called "Instantiated protocol specification". While the syntax of this protocol specification is quite simple, it needs a few explanations.

Role instances : In cl-atse, each protocol role must be independent from the others and use different variable names. Therefore, each role in the IF file is instantiated into a set of roles, one for each "state_" fact in the initial state. This instantiation actually "runs" the participant role, generate new variable names, and extract a (minimal) set of constraints representing this role. For example, a variable GX in role "Server" might become GX(1) and GX(2) in two role instances of this "Server". Also, any nonce (say, *Na*) generated in the specification is replaced by a different constant in each role instance (say, $n1(Na)$ and $n2(Na)$). For readability the set of constraints representing each role instance is displayed in the following Send/Receive syntax:

$$\textit{Step} \;:\quad \text{Received Msg} \;\Rightarrow\; \text{Sent Msg} \quad [\textit{sigma}] \quad \&\ \text{Inequalities} \;\; \&\ \text{IF Facts}$$

where *sigma* is a unification constraint (i.e. a set of equalities). For conciseness, the & symbol is only written for new lines. Any non-determinism in the role execution is represented by a choice point on a set of roles. When a choice point is executed, the system (in fact, the intruder) chooses what branch will be run. Finally, a role consists in a tree where unary nodes are protocol steps and n-ary nodes are choice points. An execution of a role is a path in this tree. An output example follows.

```
Intruder state
--------------

Intruder Knowledge : start i {i,ki}_(inv(kca)) inv(ki) ki kca
```

List of initial intruder knowledges.

```
 Unforgeable terms : inv(ks) inv(kca)
```

Computed list of term that the intruder cannot forge.

```
Interpreted protocol specification
----------------------------------

Role server played by (s,7):
```

First instance of the role "server".

```
| start => s, ks, n26(Ns)
```

First step: receives *start* and send a nonce $n26(Ns)$.

```
| Choice Point
```

Second step: chose one branch or the other.

```
|  | Csus(27), {i,ki}_(inv(kca)) => n27(SeID)
```

Third step: assumes $\{i, ki\}_{inv(kca)}$ was received.

```
|  | .....
```

Other steps.

```
| Or
|  | Csus(31), {s,ks}_(inv(kca)) => n31(SeID), n31(Ns)
```

Third step (other choice): assumes $\{s, ks\}_{inv(kca)}$ was received.

```
|  | .....
```

Other steps.

IF Facts Each protocol step contains all the facts found for this step in the IF file (except the state). While their syntax might look a bit different than original IF facts due to cl-atse's internal fact representation, their semantic is identical. Major differences are :

- *contains*(*term*, *set*) facts are changed into "Test *term* in *set*", "Test *term* not in *set*", "Add *term* to *set*", and "Remove *term* from *set*" depending on the position of the *contains*(..) fact in the rule, and following the semantic of *contains*(..) facts. Tests are preconditions for the protocol step containing them.
- *secret*(*term*, *mark*, *set*) becomes "Secret (*term*,*set*)".

3.2.2 Using `OFMC` The On-the-Fly Model-Checker OFMC builds the infinite tree defined by the protocol analysis problem in a demand-driven way, i.e. on-the-fly, hence the name of the back-end. It uses a number of symbolic techniques to represent the state-space as described in more detail in [9, 8]. Thanks to these techniques, OFMC can be employed not only for efficient falsification of protocols (i.e. fast detection of attacks), but also for verification (i.e. proving the protocol correct) for a bounded number of sessions — without bounding the messages an intruder can generate.

The most significant new feature of OFMC in this release is that the user can specify an algebraic theory on message terms, modulo which the analysis of the protocol is performed. For more information on using algebraic theories, see the file ”user-guide-algebraic.pdf” in the `docs/ofmc` directory included in this distribution. Example theories and sample IF specifications that employ algebraic theories can be found in the directory `testsuite/algebraic`.

Usage. OFMC is invoked by typing on the command-line

```
ofmc <filename> [-theory <theoryfile>] [-sessco] [untyped]
                [-d <number>] [-p <number>*]
```

- `<filename>` is an IF file to be checked.
- The `-theory` option allows the user to specify a custom algebraic theory given in the file `<TheoryFile>`. See the file ”user-guide-algebraic.pdf” in the docs/ofmc directory for more information, including details about the default algebraic theory used by OFMC. See also the examples in the directory `testsuite/algebraic`. Note that user-defined theories are currently only supported for protocols specified directly in IF. The `--no-hlpsl2if` option can be useful when passing IF specifications directly to the AVISPA Tool.
- When using the `-sessco` option, OFMC will first perform a search with a passive intruder to check whether the honest agents can execute the protocol, and then give the intruder the knowledge of some “normal” sessions between honest agents. In the case certain steps cannot be executed by any honest agent, OFMC reports that the protocol is not executable and stops. If the executability check is successful, then the normal search with an active intruder is started, with the only difference that the intruder initially knows all the messages exchanged by the honest agents in the passive intruder phase.
- The `untyped` option forces OFMC to ignore all types specified in an IF file. (This is equivalent to specifying no types at all in IF or to give atom and variable the type `message`).
- Using the `-d` option one can specify a depth bound for the search (the default being unbounded depth). In this case, OFMC uses a depth-first search (while the standard search strategy is a combination of breadth-first search and iterative deepening search).

- Using the `-p` option, one can "manually browse" the search tree, e.g.:

  `-p` is the root node,

  `-p 0` is the first (left-most) successor of the root node,

  `-p 0 1` is the second successor (next to left-most) successor of the node obtained by `-p 0`.

  An exception is raised if a path to a non-existing node is specified.

3.2.3 Using `SATMC`

The SAT-based Model-Checker (SATMC, [5]) builds a propositional formula encoding a bounded unrolling of the transition relation specified by the IF, the initial state and the set of states representing a violation of the security properties. (The SAT compilation of IF specifications results from the combination of a reduction of security problems to planning problems and SAT-encoding techniques developed for planning.) The propositional formula is then fed to a state-of-the-art SAT solver and any model found is translated back into an attack.

In implementing SATMC, we have given a great deal of care on design issues related to flexibility, modularity, and efficiency. The result of such an effort is an open and flexible platform for SAT-based bounded model checking of security protocols. For instance, improvements of SAT technology can be readily exploited by integrating in a plug and play manner state-of-the-art SAT-solvers (e.g. the best performers in the SAT competition, [15]). Similarly, advancements and novelties in AI planning SAT-reduction techniques can be promptly implemented in SATMC.

SATMC can be employed not only for discovering attacks on protocols, but also for verification (i.e. proving the protocol satisfies its security requirements) of a bounded number of sessions, a problem that has been proved (see [14]) to belong to the same complexity as SAT i.e. NP-complete.

Usage. SATMC can be invoked by typing on the command-line

```
satmc <filename> --prelude=<fileprelude>
                 [--max=<number>] [--encoding=<encoding>] [--mutex=<number>]
                 [--solver=<solver>] [--ct=<bool>] [--oi=<bool>]
```

where `<filename>` and `<fileprelude>` are, respectively, the IF problem to be analysed and the prelude file, and each option is described as follow:

- `max`: maximum depth of the search space up to which SATMC will explore (the parameter `max` can be set to `-1` meaning *infinite*, but in this case the procedure is not guaranteed to terminate); by default it is set to `11`.
- `encoding`: the selected SAT reduction encoding technique (currently implemented are the linear encoding [1] and two graphplan-based encodings, one using the backward chaining

schema [3] and the other one applying the explanatory frame schema); it can be set to either `linear`, `gp-bca` or `gp-efa` (default value).

- `mutex`: level of the mutex relations to be used during the SAT-reduction; if set to `0`, then the abstraction/refinement strategy provided by SATMC (see [2] for more details) is enabled; otherwise the abstraction/refinement strategy is disabled and the static mutexes are generated; moreover if `mutex` is set to `2` and the encoding `gp-bca` has been selected, then also the dynamic mutexes are computed.

- `solver`: the selected state-of-the-art SAT solver (Chaff [13], SIM[11], and SATO [17] are currently supported); it ranges over the values `chaff` (default value), `sim`, and `sato`.

- `ct`: a Boolean parameter for enabling or disabling the *compound typing* assumption presented in Deliverable 3.2 [6] (see also section 2.1.1); by default it is set to `true`.

- `oi`: a Boolean parameter for enabling or disabling the *optimised intruder model* presented in [4]; by default it is set to `true`. Disabling such an option can be useful to experiment the effectiveness of the optimised intruder model.

Notice that expert users can change the default values associated to the above options by acting on the `bin/backends/satmc.config` configuration file.

3.2.4 Using `TA4SP` Given an initial state, the `TA4SP` tool computes either an over-approximation or an under-approximation of the intruder knowledge by means of rewriting on tree languages in a context of unbounded number of sessions. The `TA4SP` tool uses the tree automata library `Timbuk 2.0` (developed by Th. Genet IRISA, Rennes France and available at `http://www.irisa.fr/lande/genet/timbuk/`) to perform the computation of the intruder knowledge (over or under approximated).

An over-approximation may lead to positive proofs of secrecy properties on the studied protocol for an unbounded number of sessions, but `TA4SP` requires a special initial state and abstractions presented in Paragraph 3.2.4. Otherwise, in the over-approximation context, `TA4SP` can only conclude that secrecy properties are safe for the given initial state.

In an under-approximation context, without any optional abstractions, the tool may show that the protocol is flawed for a given secrecy property.

To verify a protocol with `TA4SP`, the empirical strategy to apply is the following:

1. The user computes an over-approximation and check secrecy properties.

2. If the first step does not allow to ensure secrecy then the user successively computes under-approximations until obtaining an attack in a reasonable time.

However, this empirical strategy does not always lead to the expected result. Indeed, an inconclusive result using an over-approximation does not imply that there exists a real attack.

Up to now, `TA4SP` does not handle sets and conditions and verifies only secrecy properties with a typed model.

The following paragraph describes the `TA4SP` options which are very useful to specify precisely the kind of verification a user wants to do.

TA4SP Options The options below are used by the binary `ta4spv2` (at `bin/backend/TA4SP`). However, to use `TA4SP` from the `avispa` script, the user will have to set these options in the *ta4sp.config* file.

- `--level <integer>` (`level=<integer>` in *ta4sp.config*): When this option is initialised to **0**, an over-approximation will be computed. With a number greater than 0, an under-approximation is computed and this number corresponds to the number of times that rewriting is applied on the tree languages computed by `TA4SP`.
- `--2AgentsOnly` (`abstractions=<boolean>` in *ta4sp.config*): This option is very useful to improve time computations. This option provides a specification in which there are only two agents (the intruder and an honest agent). If secrecy properties are verified in this model then they are verified in the specified model (IF specification). However, if there is an attack then it may be a false one due to the abstractions done. Another interesting point is when an initial state specify:
  - a session between honest agents and
  - all possible sessions where the intruder plays at least one of the role (for example in NSPK, (Alice played by a, Bob played by b), (Alice played by a, Bob played by i) and (Alice played by i, Bob played by b)),

  and when the given secrecy properties are verified with `TA4SP`. In this context, the given properties will be verified for any sessions.

TA4SP Outputs As seen in the previous paragraph, several outputs are possible depending on the options used and the protocol to check. These following examples illustrate the following cases:

1. Secrecy verified in an over-approximation context;
2. Secrecy not verified in an over-approximation context;
3. Secrecy violated in an under-approximated context;
4. Attack not yet found in an under-approximated context.

These examples about `ta4spv2` runs concern the two protocols: Needham Schroeder Public Key protocol (NSPK.if) and its corrected version (NSPK-fix.if).

1. `./ta4spv2 --2AgentsOnly --level 0 NSPK-fix.if`:

```
SUMMARY
   SAFE

DETAILS
   TYPED_MODEL
   OVER_APPROXIMATION
   UNBOUNDED_NUMBER_OF_SESSIONS

PROTOCOL
   NSPK-fix.if
...
COMMENTS
   TA4SP uses abstractions '2AgentsOnly'
   For the given initial state, an over-
   approximation is used with an unbounded
   number of sessions.
   Terms supposed not to be known by the
   intruder are still secret.
...
```

2. `./ta4spv2 --2AgentsOnly --level 0 NSPK.if`:

```
SUMMARY
   INCONCLUSIVE
DETAILS
   OVER_APPROXIMATION
   UNBOUNDED_NUMBER_OF_SESSIONS
   TYPED_MODEL

PROTOCOL
   NSPK.if
...
COMMENTS
   TA4SP uses abstractions '2AgentsOnly'
   Use an under-approximation in order to
   show a potential attack.
   The intruder might know some critical
   information
...
```

3. `./ta4spv2 --level 7 NSPK.if`:

```
SUMMARY
   UNSAFE
DETAILS
   UNDER_APPROXIMATION
...
PROTOCOL
   NSPK.if
...
COMMENTS
   In our model, there is a potential attack.
   The intruder may know some critical
   information
...
```

4. `./ta4spv2 --level 3 NSPK.if`:

```
SUMMARY
   INCONCLUSIVE
DETAILS
   UNDER_APPROXIMATION
...
PROTOCOL
   NSPK.if
...
COMMENTS
   Use a greater bound or check the protocol in
   an over-approximated context.
...
```

3.3 The Standard Output Format

All back-ends of the AVISPA tool have the same output format. Based on this format a tool may be used for graphically representing an attack as a sequence of message exchanges. Such a graphical tool is not distributed in this light package, but will soon be supplied in a more complete version.

As a consequence, if you plan to add your own verification tool to the AVISPA tool, we recommend to follow the following output syntax.

```
% AVISPA output format BNF
%
```

```
% ------------- the following symbols are assumed:
% ident                  a string
% int                    an integer number
% float                  a floating-point number
% msg                    a string describing a message
% goalDescription        a string describing a goal
% msg_ident              a constant (initial lower case letter)
%                        or a variable (initial capital letter)
% -------------

comment ::=
  "%" msg

Output ::=
  Summary Protocol Goal BackEnd Comments? Statistics Trace

Summary ::=
  "SUMMARY" Result

Result ::=
  Conclusive
| Inconclusive

Conclusive ::=
  "SAFE" ConclusiveDetails
| "UNSAFE" ConclusiveDetails

ConclusiveDetails ::=
  "DETAILS" (ConclusiveExplanation)+

ConclusiveExplanation ::=
  "ATTACK_FOUND"
| "STRONGLY_TYPED_MODEL"
| "TYPED_MODEL"
| "UNTYPED_MODEL"
| "BOUNDED_NUMBER_OF_SESSIONS"
| "BOUNDED_NUMBER_OF_SYMBOLIC_SESSIONS"
| "BOUNDED_SEARCH_DEPTH"
| "BOUNDED_MESSAGE_DEPTH"

Inconclusive ::=
  "INCONCLUSIVE" InconclusiveDetails

InconclusiveDetails ::=
  "DETAILS" (InconclusiveExplanation)+

InconclusiveExplanation ::=
  "TIME_OUT"
| "MEMORY_OUT"
| "NOT_SUPPORTED"
| "OVER_APPROXIMATION"
| "UNDER_APPROXIMATION"

Comments ::=
  "COMMENTS" msg*

Protocol ::=
  "PROTOCOL" ident

Goal ::=
  "GOAL" ident

BackEnd ::=
  "BACKEND" ident

Statistics ::=
  "STATISTICS" LabeledStat+

LabeledStat ::=
  StatLabel ShortStat UnitLabel
StatLabel ::=
  ident
UnitLabel ::=
  ident
ShortStat ::=
  float

Trace ::=
  "ATTACK TRACE" Step+

Step ::=
  StepNumber? Agent -> Agent : Msg
```

```
StepNumber ::=
  int "."

Agent ::=
  "(" ident "." SessionNumber ")"
| "i"

SessionNumber ::=
  int

Msg ::=
  Msg ("," Msg)*
| Msg ("." Msg)*
| ident "(" Msg ")"
| "(" Msg ")"
| msg_ident
```

An example of output is given in Section 2.2.4, for the protocol NSPK-KS analysed by CL-AtSe.

A XEmacs mode

A mode for editing, compiling and analyzing protocol specifications written in HLPSL is available for XEmacs. This mode can either be installed directly by the avispa package or an archive file can be downloaded separately at `http://www.avispa-project.org/software.html`.

A.1 Installation

If the XEmacs mode is installed through the avispa package, one only has to specify repositories for the different type of files:

- protocol specification files in HLPSL edited by the user;
- Intermediate Format files generated by the compiler;
- ATK files that are output of the avispa backends.

If a separate installation of the mode is scheduled, one should download the tar archive. The procedure is then:

```
tar -xzvf avispa-mode.tgz
cd temporary-avispa
make install
```

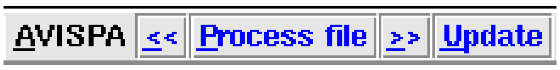
If the shell variable `AVISPA_PACKAGE` is not set, the different target repositories for the avispa files are asked. Otherwise the repositories for the global files are automatically inferred from the value of this variable, as `$AVISPA_PACKAGE/emacs` for the XEmacs mode files and `$AVISPA_PACKAGE/bin` for the path to the tools (the backends and the `hlpsl2if` translator).

It is assumed that one has a working version of XEmacs to compile the mode files. This means that it is currently not possible to install the mode on a server that does not have XEmacs. The `init.el` file of the user is changed to auto-load the *avispa mode* when opening an avispa related file (suffixed by `hlpsl`, `atk` or `if`). The changes also automatically add the path to the Emacs repository where the mode files are. This repository does not have to be a global one.

A.2 Usage

A.2.1 First steps... The Avispa XEmacs mode permits to specify and analyse protocol specifications in an integrated environment.

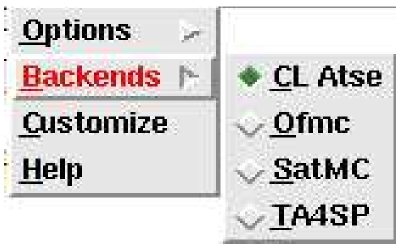
The usual starting point of analysis is the opening of a hlpsl file. The Avispa mode, once correctly installed, automatically detects this kind of files on account of its suffix "`.hlpsl`". Several buttons then appear on the menubar:



These buttons have the following role:

- **AVISPA** permits to open the Avispa menu to customize the mode and change the backend as well as its options;
- << and >> permit to navigate among the different files (".if", ".atk") related to the specification of a protocol;
- **Process file** permits to launch either the **hlpsl2if** compiler on the current HLPSL buffer or the current backend on the current buffer.
- When a tool is launched asynchronously with XEmacs, the **Update** permits to refresh the content of the current buffer once the tool has terminated.

A.2.2 The Avispa Menu The Avispa menu permits to change the behavior of the tools when analyzing a specification. Before describing the possible options, its first use is to select a specific backend for the analysis of a IF specification (button **Backends**):



The **Customize** item permits to access to the customization of the variables in the 'avispa-tools group (the most useful ones). The customization permits to change some default values permanently, see Section A.3 for more information on this topic.

The **Help** permits to launch a small help file.

A.2.3 The options This submenu is accessed *via* the **Options** of the Avispa menu. There are two kinds of options:

- The "Avispa" options which relate to all backends and have to be handled by all tools;
- The tool-specific options, each related to a specific backend.

The former are immediately accessible in the Options submenu, while the latter are accessible *via* the $\boxed{\textbf{More Options}}$ button. Both kind of options are described in previous sections of this manual, but for the $\boxed{\textbf{Verbose}}$ option which, if unselected, suppresses the hlpsl2if compiler warnings on the type of constants. Note that setting an option through this menu lasts a whole session (*i.e.* until XEmacs is closed).

A.2.4 Navigation The *avispa mode* keeps internally a *current state* that takes into account both the current file and the current tool that will be applied on the current file through the **Process file** button. This permits to navigate among the different files using the $\boxed{<<}$ and $\boxed{>>}$ buttons.

The $\boxed{<<}$ button. If the current file is an ATK file, it changes to the corresponding IF file. If the current file is an IF file, it changes to the corresponding HLPSL file.

The $\boxed{>>}$ button. If the current file is an HLPSL file, it changes to the corresponding IF file. If the current file is an IF file, it changes to the ATK file corresponding to the current backend.

The action in other cases depends on the value of the **Navigation Button Wrap** variable. The possible behaviors are explained in Section A.3.2.

A.3 Customization

The *avispa mode* may be customized through the command:

```
M-x customize
```

You may then enter one of the following:

- **avispa-project** permitting access to all other variables;
- **avispa-tools** containing generic options that are explained in Subsection A.3.2;
- **avispa-hlpsl2if**, **avispa-atse**, **avispa-ofmc**, **avispa-satmc** and **avispa-ta4sp** provide another way to change the options of the backends and of the translator. It also permits to save options from one session to another. It is also possible to change the names of the executable launched upon calling a tool (the translator or a backend).

Note that the $\boxed{\textbf{AVISPA}} \rightarrow \boxed{\textbf{Customize}}$ button permits to directly access the **avispa-tools** group of variables.

A.3.1 The *avispa-project* group This is the global group of variables related to the *avispa mode*. Its main use is to present the Avispa Project, since most variables are member of the *avispa-tools* group. It permits to set the *Use Abbrev Mode* that controls whether abbreviations should be launched when accessing a new buffer with the Avispa mode.

The description of this variable also contains the abbreviations defined by default.

A.3.2 The *avispa-tools* group A first subset of the variables in the **avispa-tools** group concerns the directories where the files are to be found.

- **Tools Path**: This is the repository where the tools of the Avispa Project are. This variable defaults to `$(AVISPA_PACKAGE)/bin` or to the directory entered in the installation of the XEmacs mode. Note that the full path to the executable used is constructed dynamically from the value of this variable and from the name of the executable.
- **Protocols Hlpsl Repository Path**: This variable specifies the repository containing the HLPSL specifications. It is used when calling the **hlpsl2if** compiler to construct the full path to the protocol specification. It is also used when navigating among files (see Section A.2.4). This path *should not* end with a slash, otherwise XEmacs may get confused during the navigation, and opens the same file twice (which is unnecessary).
- **Protocols If Repository Path**: This is the equivalent of the **Protocols Hlpsl Repository Path** variable with regards to the Intermediate format files. It is also used by the **hlpsl2if** compiler to determine the full path to the output file. compiler.
- **Protocols Results Repository Path**: This is the equivalent of the **Protocols Hlpsl Repository Path** variable with regards to the resulting result files (suffixed by ".atk"). It is also used to determine the full path to the auxiliary output files of a backend. Currently, only samtc and ta4sp generate such files that contain statistics on the analysis.

The second subset of variables in the **avispa-tools** group concerns the behavior of XEmacs when calling the translator or a backend.

- **Synchronous Compilation**: This variable controls whether XEmacs shall spawn a new, concurrent process to compile an HLPSL specification or if it should wait the result of the compilation before resuming execution. If it is set to a non-nil value (*e.g.* t), it enables the automatic jump to the corresponding Intermediate Format buffer if **Fetch Result** is also non-nil;
- **Synchronous Validation** Set this value to true if you want XEmacs to wait for the result of the backend before resuming execution. This enables the automatic jump to the corresponding Result buffer if **Fetch Result** variable is also set to true.

The drawback is that XEmacs will hang if the backend does not terminate. Note also that Ofmc is not sensitive to this value, and will always be launched asynchronously.

When a backend or the compiler is launched asynchronously, one need to use the navigation buttons $\boxed{<<}$ *and* $\boxed{>>}$ *to go to the result buffer. Once in the right buffer, one should use* $\boxed{\textbf{\textit{Update}}}$ *to see the result. This should be done only once the tool has terminated.*

- **Fetch Result**: Set this value to nil if you do not want the mode to automatically display the result of a process, i.e. compilation or verification. There is no automatic fetching when the process spawned is not synchronous with XEmacs. This is due to a race condition that would often result in XEmacs displaying an out-of-date version.
- **Navigation Button Wrap**: The variable permits to define the behavior of the $\boxed{<<}$ button when visiting an HLPSL buffer and of the $\boxed{>>}$ button when visiting a result buffer. If set to nil, these buttons will produce no effect. If set to any other value, the $\boxed{<<}$ button permits to jump directly from an HLPSL buffer to the corresponding result buffer. The name of this buffer is computed with respect to the backend that would be used for next analysis of a IF buffer. Conversely, the $\boxed{>>}$ permits to jump directly from a result file to the corresponding HLPSL specification.

  These two buttons do not launch any application nor update the content of a buffer visiting a file.

B HLPSL Semantics

B.1 Preliminaries

The semantics of HLPSL is based on the Temporal Logic of Actions [12] (TLA, for short) a powerful logic which is well-suited to the specification of concurrent systems like security protocols. TLA itself has an intuitive and easily understandable semantics, making it a formalism that protocol designers and engineers can find accessible.

Although TLA allows for the description of parallel processes, in the context of HLPSL we model protocols by providing an interleaving semantics. The latter is obtained by restricting the capabilities of the Intruder (see Section B.2.3): in the case of the Dolev-Yao [10] intruder model, for instance, we make the intruder send messages one at the time.

B.1.1 The Transition System. TLA specifies a transition system by describing its allowed behaviours by means of a single formula of the form:

$$System(\Delta) \;\triangleq\; Inits(\Delta) \wedge \Box Next(\Delta, \Delta')$$

where Δ is the set of state variables ranging on a domain $\mathcal{D}$[4] (Δ' refers to this set of variables in the next state), *Inits* and *Next* are formulae representing the initial states and the next-state relation, respectively. The above formula corresponds to a transition system $\mathcal{T} = \langle \Sigma, \mathcal{I}, \rightarrow \rangle$, where Σ, the set of states, is a set of total assignments $\sigma : \Delta \mapsto \mathcal{D}$; $\mathcal{I} \subseteq \Sigma$ is the set of initial states, that is, for each $\sigma_0 \in \mathcal{I}$, $\models_{\sigma_0} Inits(\Delta)$; finally, $\rightarrow \subseteq \Sigma \times \Sigma$ is the transition relation such that $\sigma_1 \rightarrow \sigma_2$ iff $\models_{\sigma_1 \cup \sigma_2'} Next(\Delta, \Delta')$ where $\sigma' = \{\langle x', d\rangle \mid \langle x, d\rangle \in \sigma\}$.

For a sequence of states $\pi = \sigma_0, \sigma_1, \ldots$, we define $\pi(i) = \sigma_i$, $\pi^{\geq i} = \sigma_i, \sigma_{i+1}, \ldots$, and $\pi_{\leq i} = \sigma_0, \sigma_1, \ldots, \sigma_i$ for $i = 0, 1, \ldots$, and we say that π is a *behaviour* in $\mathcal{T}$ iff $\sigma_i \rightarrow \sigma_{i+1}$ for each $i = 0, 1, \ldots$.

B.1.2 Events and Actions. A *state predicate* is a TLA formula on a role's state variables and constants. Examples of valid state predicates include $X = 5$ and $S = done$. A *transition predicate* is similar but may include primed variables. If V is a tuple of state variables and V' the correspondent tuple of primed state variables, then we define the set of *actions* as those transition predicates $p(V, V')$ with the property that $\forall V : \exists V' : p(V, V')$.[5] Actions may therefore include stuttering steps. A *basic event* is a conjunction of transition predicates, at least one of which is of the form $p(V') \neq p(V)$, where V is a tuple of variables and $p(V)$ is a state predicate. This definition ensures that events are *non-stuttering*, i.e. at least one state variable changes.[6]

[4]For simplicity we consider a single domain for all the variables. However what follows can be easily extended to multi-domain.

[5]Notice that, the satisfaction of this property causes the transition relation to be *total*.

[6]Notice that, $p(V') \neq p(V) \Rightarrow V' \neq V$ can be simply derived from $V' = V \Rightarrow p(V') = p(V)$ by means of contraposition.

B.1.3 Transitions. The next-state relation of basic roles is defined through a set of transitions rules within the `transition` section. HLPSL distinguishes between two different types of transitions, respectively called *spontaneous actions* (denoted by the `--|>` arrow) and *immediate reactions* (specified using the `=|>` arrow). Both kinds of transitions are preceded by a label, which may be an alphanumeric string starting with a lower case letter and ending with a period. It is worth pointing out that these labels carry no information about the order in which the transitions fire. They are merely names.

Spontaneous actions relate a state predicate on the left-hand-side (LHS) with an action on the right-hand-side (RHS). Intuitively, a spontaneous action transition *sp* `--|>` *act* expresses that, whenever we are in a state that fulfils state predicate *sp*, we *may* make a transition labelled by the action *act* into a new state. Note, however, that we do not require that this transition fire right away. When and if it does fire, we obtain the new state by taking the old state and applying any side effects that *act* might produce; that is, any state variables not affected by *act* remain the same. We call such transitions "spontaneous" because, although they are enabled whenever state predicate *sp* is true, they convey nothing about when they must fire.[7]

Immediate reaction transitions, on the other hand, have the form *event* `=|>` *act* and relate a trigger event, *event*, on the LHS with an associated action, *act*, on the RHS. This expresses that, whenever we take a transition that satisfies the non-stutter event predicate *event*, then we *must* immediately (more precisely, simultaneously) execute action *act*. Hence as soon as *event* holds, we obtain the new state by taking the old state and applying any side effects that *act* might produce; that is, any state variables not affected by *act* remain the same.

B.1.4 Communication, Channels, and Signals. Communication takes place over channels, which are themselves merely variables with values like any other. By convention, we generally assign channels convenient names like `SND` and `RCV` and then write `SND(Msg)` and `RCV(Msg)`. This is, however, merely a shorthand. The former action, assuming that it appears on the RHS of a transition, meaning it is a write action, is a short form for `SND'=Msg`. The latter, assuming that it appears on the LHS of a transition and is therefore a read event, is short for `(RCV_flag' /= RCV_flag) /\ (RCV' = Msg)`, where `RCV_flag` is a binary flag which is toggled each time a new message arrives on channel `RCV`. Recalling our restriction that events contain at least one predicate of the form `X' /= X`, we can see that the former is an action and the latter an event.

HLPSL also supports signals, that is, asynchronous events (assuming they occur on the LHS of a transition). A signal can be seen as a channel through which no information is sent. Signals in HLPSL are of the form `SGNL()`. As for channels, the latter is just a shorthand for

[7]It can be the case that, at a given point, a spontaneous transition is enabled but does not fire directly and, meanwhile, some other transition is applied such that its effects make the spontaneous transition no longer enabled. In that case, the latter will not be able to fire anymore, unless it becomes enabled once again thanks to the effects of some other transition.

`(SGNL_flag' /= SGNL_flag)`, meaning that something has changed in that particular channel (but no messages have been sent through it, since there is no assignment `SGNL_flag'=Msg` as for normal communication channels).

In the rest of this section we will show how to express a security protocol specified in HLPSL as a TLA formula. We will formally give the meaning of checking that a security protocol achieves its security properties under an hostile environment.

B.2 Formal Semantics

B.2.1 Messages. We begin by specifying the structure of messages and the properties of the operations on the set *Msg* of all messages. Particularly, we will focus on pairing $pair(M_1, M_2) = M_1.M_2$ and asymmetric encryption, $acrypt(K, M) = \{M\}_K$, but we can easily extend this model of messages to include other operators like symmetric encryption, exponentiation, XOR, and their associated properties.

In general, HLPSL allows for the declaration of algebraic equations that specify an equational theory $\approx$. Let Ω be the signature from which *Msg* is constructed, the interpretation of HLPSL is required to be a quotient interpretation of a free term algebra $\mathcal{T}_\Omega$ modulo the equational theory $\approx$ (see [16]). For instance, $\approx$ could include an equation stating that pairing is associative $Pair(Pair(m1, m2), m3) \approx Pair(m1, Pair(m2, m3))$, equations expressing properties of cryptographic algebraic operators (e.g. the associativity and commutativity properties of the XOR operator, properties on the inverse of an asymmetric key, like $inv(inv(k)) \approx k$), etc.

B.2.2 HLPSL Roles. In this subsection, we show how HLPSL roles are mapped into TLA.

Figure 3 shows the structure of a basic role (denoted with B) and of a composed role (denoted with P) in HLPSL, where Ψ_B and Ψ_P are the sets of parameters, $pl \in \Psi_B$ is the agent playing B, $R_1, R_2, \ldots, R_m$ are the component roles (for brevity, sequential composition, the loop construct and the acceptance conditions are not discussed), Λ_B and Λ_P are the sets of local variables, and Ω_B and Ω_P are the sets of *owned* variables. Moreover, even if it is not depicted explicitly by Figure 3, let Υ_B, Υ_P be the sets of fresh variables which are updated by the role.

In the rest of this section, given a role named R, we denote with $\mathcal{R}$ its entire HLPSL definition which includes the roles signature (the name, R, and the parameters, that we denote with $\Psi(\mathcal{R})$), the player (for basic roles), the local and owned variables (that we denote with $\Lambda(\mathcal{R})$ and $\Omega(\mathcal{R})$ respectively), $Init_R$ (that we denote with $Init(\mathcal{R})$), and the set of transitions or the compositions defined therein. Same goes for the set of fresh variables, denoted with $\Upsilon(\mathcal{R})$. Moreover, if R is a basic role, in $\mathcal{R}$ the set $\Psi(\mathcal{R})$ of its parameters is augmented with a new variable U that will later be instantiated with a unique number. For instance, in a parallel composition this allows to distinguish between two (or more) identical instances of the same basic role.

As a first preprocessing step, we uniformly rename the roles local variables in order to avoid

```
role B(Ψ_B) played_by pl def=
  local Λ_B
  owns Ω_B
  init Init_B
  transition
    lb_1.  ev_1 =|> act_1
    ...
    lb_n.  ev_n =|> act_n
end role
```

(a) Definition of a basic role

```
role P(Ψ_P) def=
  local Λ_P
  owns Ω_P
  init Init_P
  composition
    R_1 ∧...∧ R_m
end role
```

(b) Definition of a composed role

Figure 3: HLPSL roles: generic structure.

possible name clashes with the environment. To aid readability, we still denote with $\Lambda(\mathcal{B})$, $\Lambda(\mathcal{P})$ the sets of local, renamed variables.

Since immediate reactions can also be used to express spontaneous actions, we focus here only on the formers. We will proceed inductively translating to TLA, starting with basic roles $\mathcal{B}$ and then giving the translation of the composed role $\mathcal{P}$ in terms of the translation of its components, $R_1, R_2, \ldots, R_m$.

In order to describe the TLA translation of a basic role, let us first define tr_j to be the j-th transition rule of the basic role (i.e., $tr_j \triangleq ev_j$ `=|>` act_j), $\Upsilon(tr, \mathcal{B})$ to be the set of variables that are freshly generated by the transition tr in $\mathcal{B}$ (i.e., $\Upsilon(tr, \mathcal{B}) \triangleq \{v \mid v \; in \; \Upsilon(\mathcal{B}), \; v' \text{ occurs in } tr\}$), and $Used_{\langle \mathcal{B}, tr_j \rangle}$ (with $j = 1, \ldots, n$) be TLA variables each one devised to model a set that keeps track of those fresh values that have already been generated in executing the transition tr_j (with $j = 1, \ldots, n$) of $\mathcal{B}$. Moreover, let $Used_{\mathcal{B}}$ be the TLA variable representing the set that keeps track of those fresh values that have already been generated by the role $\mathcal{B}$. The TLA translation of a basic role is then given by Figure 4. Notice that also a channel Ch can be owned by a role. In such a case, $Mod(Ch, \mathcal{B})$ is intended to be applied on transitions in which the channel macro has been replaced with its appropriate conjunction of predicates (see section B.1.4).[8]

Intuitively, formula (1) of Figure 4 states that initially $Init(\mathcal{B})$ holds, and in every step the above conjunction of formulae denoted with $Next(\mathcal{B})$ must be satisfied. Namely, (2) states that if an event is triggered, then the changes specified by the corresponding action take place, the fresh variables updated by the transition are assigned to different values that have never been used as fresh value by this transition; although it is not explicitly stated, the player's knowledge is implicitly extended with all the terms that can be derived by analysing[9] the messages received, and

[8]However, it is recommended to avoid ownership of channels since it can be cause of clashes with the TLA formulae declared for the intruder.

[9]For instance, if an agent knows a symmetric key k and receives a cyphertext $\{M\}_k$, then M is added to the knowledge.

$$TLA(\mathcal{B}) \;\triangleq\; Init(\mathcal{B}) \wedge \Box\, Next(\mathcal{B}) \tag{1}$$

where $Next(\mathcal{B})$ is defined as:

$$\begin{array}{ll}
\wedge & \bigwedge_j ev_j \Rightarrow \wedge\ act_j \\
& \qquad\quad \wedge \bigwedge_{(\upsilon_1,\upsilon_2\ in\ \Upsilon(tr_j,\mathcal{B}))} \upsilon_1{}' \neq \upsilon_2{}' \\
& \qquad\quad \wedge \bigwedge_{(\upsilon\ in\ \Upsilon(tr_j,\mathcal{B}))} \upsilon' \notin Used_{\langle\mathcal{B},tr_j\rangle} \wedge \upsilon' \in Used_{\langle\mathcal{B},tr_j\rangle}{}'
\end{array} \tag{2}$$

$$\wedge \quad \bigwedge_{(i\neq j)} Used_{\langle\mathcal{B},tr_j\rangle} \cap Used_{\langle\mathcal{B},tr_j\rangle} = \emptyset \tag{3}$$

$$\wedge \quad \bigwedge_{(\omega\ in\ \Omega(\mathcal{B}))} \omega' \neq \omega \Rightarrow Mod(\omega,\mathcal{B}) \tag{4}$$

$$\wedge \quad Used_{\mathcal{B}} = \bigcup_j Used_{\langle\mathcal{B},tr_j\rangle} \tag{5}$$

$$\wedge \quad Used_{\mathcal{B}} \subseteq Used_{\mathcal{B}}{}' \tag{6}$$

with

$$Mod(x,\mathcal{B}) \;\triangleq\; \bigvee_j \{ev_j \mid x'\ occurs\ in\ tr_j\}. \tag{7}$$

Figure 4: Translation of a basic role in into TLA

we assume the player to be always able to compose the messages he is going to send. Conjunction (3) imposes that the sets of fresh values issued by different transitions are disjoint. Besides this, (4) states that if one of the variables owned by the role changes, then the variable is actually modified by this role. It is our convention that if a role owns a variable then this variable is never modified by any role "outside" the current one. Finally, (5) defines a TLA variable representing all the fresh values used by the role $\mathcal{B}$ and (6) imposes that such a set grows up monotonically.

An agent may simultaneously participate both in different roles and in different sessions of the protocol. In this case, the two role instances could share some internal variables of the agent. This variable sharing is not done through channels, but it is a straightforward consequence of using the same TLA variable in both the formulae representing the roles that share the variable. In this version of the AVISPA Tool, only the sharing of variables of type `set` is allowed.

Note that a transition that has to refer to the already known value of a variable will use the name of this variable, without prime sign, in any side of the transition. Viceversa, when a transition has to assign a value to a variable will use the name of this variable with prime.

Lastly, let $\mathcal{B}$ be a basic role. With $\mathcal{B}^p$ we denote the role obtained from $\mathcal{B}$ by replacing the variable U with the value p and by replacing each variable $\lambda \in \Lambda$ with λ_p. On composed roles, $\mathcal{P}^p$ denotes $\mathcal{P}$ itself, acting as the identity function. This is done in order to keep same basic role instances distinct when they are involved in parallel compositions.

The TLA translation of the parallel composition of $\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_m$ is as follows:

$$\begin{aligned}
TLA(\mathcal{P}) \quad \triangleq \quad & \wedge \bigwedge_{i=1}^{m} TLA(\mathcal{R}_i^{p_i}) && (8)\\
& \wedge Init(\mathcal{P}) && (9)\\
& \wedge \Box \ \wedge \bigwedge_{(\omega \ in \ \Omega(\mathcal{P}))} \omega' \neq \omega \Rightarrow Mod(\omega, \mathcal{P}) && (10)\\
& \qquad \wedge \bigwedge_{(\lambda \ in \ \Lambda(\mathcal{P}))} \lambda' \neq \lambda \Rightarrow Mod(\lambda, \mathcal{P}) && (11)\\
& \qquad \wedge \bigcap_{i=1}^{m} Used_{\mathcal{R}_i^{p_i}} = \emptyset && (12)\\
& \qquad \wedge Used_{\mathcal{P}} = \bigcup_{i=1}^{m} Used_{\mathcal{R}_i^{p_i}} && (13)
\end{aligned}$$

where $p_1, p_2, \ldots, p_m$ are the positions of $TLA(\mathcal{R}_1^{p_1})$, $TLA(\mathcal{R}_2^{p_2}), \ldots, TLA(\mathcal{R}_m^{p_m})$ respectively in the tree of the main role at the topmost-level of the HLPSL hierarchy, and [10]

$$Mod(x, \mathcal{P}) \quad \triangleq \quad \bigvee_{i=1}^{m} Mod(x, \mathcal{R}_i^{p_i}).$$

It is thus defined as the conjunction of each component role $TLA(\mathcal{R}_i^{p_i})$ (see (8)) and some terms accounting for extra initial constraints (see (9)), taking ownership of variables (see (10)) and freshness (see (12)). Namely (10) states that if a variable is owned by $\mathcal{P}$ then the value of such variable can be modified only by applying transitions of any of the component roles $\mathcal{R}_i^{p_i}$. Same goes for local variables (11): a local variable can be modified only by any of the component roles. For what concerns freshness, (12) imposes that the sets of fresh values issued by all roles $\mathcal{R}_i^{p_i}$ are disjoint, and (13) defines a TLA variable that keeps track of those fresh values that have already been used by $\mathcal{P}$, i.e. the union of the fresh values used by each $\mathcal{R}_i^{p_i}$.[11] Moreover, it is immediate to see that $\bigcup_{i=1}^{m} \Omega(\mathcal{R}_i^{p_i}) \subseteq \Omega(\mathcal{P})$, $\bigcup_{i=1}^{m} \Lambda(\mathcal{R}_i^{p_i}) \subseteq \Lambda(\mathcal{P})$, and $\bigcup_{i=1}^{m} \Upsilon(\mathcal{R}_i^{p_i}) \subseteq \Upsilon(\mathcal{P})$.

The TLA translation of sequentially composed roles, which we omit here, is analogous. One must augment the translation with an auxiliary variable recording which of the roles is executing and take into account the acceptance conditions.

B.2.3 Intruder Model. We formalise the capabilities of the intruder as a set of rules the intruder may execute. We focus here on the well-known Dolev-Yao (DY) intruder model of [10] but note that the definition of alternate intruder models is a simple matter of axiomatically describing their capabilities. In this way, we can easily model a system in which the intruder has full DY capabilities over certain communication channels, can only listen on others, and has no access to a third set of channels.

The DY intruder controls any channel tagged with the `(dy)` attribute. Let S and R be the number of sending and receiving DY channels used in a given protocol. In the sequel, SND_i and

[10]Notice that, the sets of basic and composed roles are disjoint. Therefore it is immediate to select the appropriate *Mod* predicate to be applied on a given role.

[11]Notice that, the monotonicity of $Used_{\mathcal{P}}$ simply follow by the monotonicity of each $Used_{\mathcal{R}_i^{p_i}}$.

RCV_k (with $1 \leq i \leq S$ and $1 \leq k \leq R$) refer to sending and receiving DY channels, respectively. The DY intruder reads every sending channel SND_i (namely, it reads every message that the agents write on these channels), analyses the messages, (i.e. generates terms and messages based on them), and inserts the composed messages into any receiving channel RCV_k. Unlike knowledge of roles, the "knowledge of the intruder" (IK) is made explicit in the formulae of Figure 5 and Figure 6. Namely, the set IK contains all the terms that the intruder may compose (with respect to his knowledge). The initial value of this TLA (set) variable is set explicitly in HLPSL: it is the union of the sets defined in the `intruder_knowledge = {...}` declarations, and monotonically increases according to the formulae of Figure 5: as the intruder reads a new message from a channel SND_i, (14), as he analyses his knowledge by decomposing a pair into its components, (15), or decrypting encrypted terms if he possesses the appropriate key, (16), or as he composes new terms – generating pairs, (17), encrypting a message using a known key, (18), or generating fresh terms (19).[12] Part of the intruder behaviour is thus formalised by the formula of Figure 6.

$$
\begin{aligned}
Read(SND_i) &\triangleq \exists_{m \in Msg} \ \wedge \ SND_i(m) \wedge \ IK' = IK \cup \{m\} & (14)\\
ASplit &\triangleq \exists_{m1,m2 \in Msg} \ \wedge \ pair(m1,m2) \in IK \wedge \ IK' = IK \cup \{m1,m2\} & (15)\\
AAdec &\triangleq \exists_{k,m \in Msg} \ \wedge \ acrypt(k,m) \in IK \ \wedge \ inv(k) \in IK \wedge \ IK' = IK \cup \{m\} & (16)\\
GPair &\triangleq \exists_{m1,m2 \in Msg} \ \wedge \ m1 \in IK \wedge \ m2 \in IK \wedge \ IK' = IK \cup \{pair(m1,m2)\} & (17)\\
GAcrypt &\triangleq \exists_{k,m \in Msg} \ \wedge \ k \in IK \wedge \ m \in IK \wedge \ IK' = IK \cup \{acrypt(k,m)\} & (18)\\
GFresh &\triangleq \exists_{x \in Msg} \ \wedge \ x \notin IUsed \ \wedge \ x \in IUsed' \ \wedge \ IK' = IK \cup \{x\} & (19)
\end{aligned}
$$

Figure 5: Dolev-Yao intruder knowledge formulae

It must be noted that, according to the definition given in Figure 6 , the intruder is able to send

$$
\begin{aligned}
Intruder_{DY} \triangleq \Box \ \wedge \ IK' \neq IK \Rightarrow & \vee \bigvee_{k=1}^{S} Read(SND_k)\\
& \vee \ ASplit \vee AAdec\\
& \vee \ GPair \vee GAcrypt \vee GFresh\\
\wedge \bigwedge_{i=1}^{R} & (RCV_i(msg) \Rightarrow msg \in IK')
\end{aligned}
$$

Figure 6: Dolev-Yao intruder behaviour

messages on more than one channel at the same time. Since we provide an interleaving semantics for HLPSL, that behaviour needs to be restricted by adding the constraint depicted in Figure 7.

[12] Notice that, the TLA variable *IUsed* keeps track of those fresh values that have already been generated by the intruder.

$$\mathit{Interleaving}_{DY} \quad \triangleq \quad \bigvee_{i=1}^{R} \left(\text{RCV\_flag'}_i \neq \text{RCV\_flag}_i \wedge \bigwedge_{j=1,j\neq i}^{R} \text{RCV\_flag'}_j = \text{RCV\_flag}_j\right)$$

Figure 7: Dolev-Yao intruder behaviour: necessary condition for interleaving semantics.

However, the formula in Figure 7, alone, does not guarantee the interleaving semantics: in fact, the HLPSL syntax allows to specify two left-hand sides of two transitions (within the same basic role) such that they both can become enabled, allowing both transitions to fire. In that case, the interleaving semantics is preserved by adding a "trigger" to the left-hand sides in order to enable one transition at the time. The trigger may be, for instance, a signal.[13]

B.2.4 Freshness. In the previous paragraphs we showed that each role *Role* keeps track in the set $\mathit{Used}_{\mathit{Role}}$ of those fresh values that have already been generated by itself. In doing this *Role* also guarantees the freshness of these values for what concerns its execution. The intruder makes the same for itself too and therefore we can guarantee the freshness of the whole system simply by enforcing the following:

$$\mathit{Nonce\_Prop} \quad \triangleq \quad \Box \quad \mathit{Used}(\mathcal{TR}) \cap \mathit{IUsed} = \emptyset$$

where *TR* is the name of the role at the topmost-level of the HLPSL hierarchy.[14]

B.2.5 Goals. In HLPSL, goals are specified as temporal formulae built on top of goal facts that are explicitly asserted by basic roles in executing their transitions. To assert a goal fact corresponds to assigning the truth value to a HLPSL boolean variable representing the goal fact.

Let Γ be the collection of such boolean variables, *TR* be name of the role at the topmost-level of the HLPSL hierarchy, and $\mathcal{T} = \langle \Sigma, \mathcal{I}, \rightarrow \rangle$ the transition system represented by the following TLA formula

$$\begin{array}{ll} \wedge & \mathit{TLA}(\mathcal{TR}) \\ \wedge & \bigwedge_{(\gamma \ \mathit{in} \ \Gamma)} \wedge\ \gamma = \text{FALSE} \\ & \qquad\qquad \wedge\ \Box\ (\gamma' = \text{TRUE} \Leftrightarrow \mathit{Mod}(\gamma, \mathcal{TR})) \end{array} \tag{20}$$

where (20) states that a goal fact *(i)* is initially false and *(ii)* holds only in those states reached by those transitions that assert it.

[13] Let, for instance, `RCV(X')=|>`act_1 and `RCV(Z')=|>`act_2 be two transitions defined within the same role. In that case, if the intruder sends a message on channel RCV, both `RCV(X')` and `RCV(Z')` become enabled, and both transitions fire at the same time. To avoid such parallelism one needs to rewrite the transitions as `TRIG1()`$\wedge$`RCV(X')=|>`act_1 and `TRIG2()`$\wedge$`RCV(Z')=|>`act_2.

[14] Alternatively, we could enforce that all the set of used fresh values are disjoint.

Besides this, let π be a behaviour in $\mathcal{T}$ and ϕ a generic safety temporal formula, then ϕ holds in π at time i, denoted with $(\pi, i) \models \phi$, is inductively defined as in Figure 8.[15] A safety temporal

$$\begin{array}{lll}
(\pi, i) \models p & \text{iff} & p \text{ holds in } \pi(i) \\
(\pi, i) \models \neg\phi_1 & \text{iff} & (\pi, i) \not\models \phi_1 \\
(\pi, i) \models \phi_1 \vee \phi_2 & \text{iff} & (\pi, i) \models \phi_1 \text{ or } (\pi, i) \models \phi_2 \\
(\pi, i) \models \phi_1 \wedge \phi_2 & \text{iff} & (\pi, i) \models \phi_1 \text{ and } (\pi, i) \models \phi_2 \\
& & \\
(\pi, i) \models \boxdot\phi_1 & \text{iff} & i > 0 \text{ and } (\pi, i-1) \models \phi_1 \\
(\pi, i) \models \Diamond\phi_1 & \text{iff} & \text{exists } 0 \leq j \leq i \text{ s.t. } (\pi, j) \models \phi_1
\end{array}$$

Figure 8: The semantics of safety temporal formulae

formula ϕ is valid on a behaviour π in $\mathcal{T}$, denoted with $\pi \models \phi$, iff $(\pi, 0) \models \phi$. A safety temporal formula ϕ is universally valid in $\mathcal{T}$, written $\mathcal{T} \models \Box\phi$, iff $\pi \models \phi$ for every behaviour π in $\mathcal{T}$.

Let $\mathcal{G}$ be the HLPSL safety formula the security protocol is required to satisfy, then we say that the security protocol (specified in HLPSL) achieves its security properties (expressed in HLPSL too) iff $\mathcal{T} \models \bigwedge_{(g \ in \ \mathcal{G})} \forall(g)$.[16]

[15]In Figure 8, ϕ_1 and ϕ_2 are safety temporal formulae and p is a goal fact.

[16]Let ϕ be a safety temporal formula, then $\forall(\phi)$ is its universal closure.

C IF Semantics

In this section, we formally describe the semantics of the IF. Recall that, as we remarked above, the translation performed by the HLPSL2IF translator defines a semantics for the HLPSL in terms of the IF, which provides an alternative to the semantics of HLPSL based on TLA (see Appendix B).

The basis of the semantics are terms, which are built from the constants and function symbols of the prelude and the IF files. As it is the case for HLPSL, we assume that all terms are interpreted in the quotient algebra of the free algebra and the equational theory defined in the prelude file.

To smoothly integrate the existential quantifier, we assume a set of fresh constants that is disjoint from all constants in the prelude and if file. For these constants, we assume a function `fresh` that maps a state and a set of variables to a substitution that replaces the variables with constants that do not appear in the given state.

Types Let *type* be a partial function that yields for every constant and variable the respective type that has been declared.

Note that our syntax allows also *compound* types (see Section 2.1.1), e.g.

$$\texttt{M} : \texttt{scrypt}(\texttt{symmetric\_key}, \texttt{pair}(\texttt{nonce}, \texttt{agent}))$$

Such a variable declaration is used when the receiver is not supposed to analyse a certain message-part according to the protocol. For instance, in the case of the Otway-Rees protocol, A should send to B a message M that is encrypted with a key K_{AS} that is shared between A and a trusted server S. B has to forward this message M to S and cannot read it himself. Hence an intruder, impersonating A, can send any message in the place of M since B will not try to analyse it. For a *typed* model, however, we want B to accept M only if it is of the proper format (according to the protocol), i.e. if it is an encryption with a symmetric key and the contents after decryption are also of the proper format. In other words, even though B cannot decrypt the message, we assume that he can check whether the received message is of the correct type and pattern, and reject it if not.

Semantically, let *op* be an n-ary IF operator, `M` a variable and $t_1, \ldots, t_n$ types (atomic or themselves composed). Then the declaration

$$\texttt{M} : op(t_1, ..., t_n)$$

is equivalent to the declarations

$$\texttt{M}_1 : t_1, \ldots, \texttt{M}_\texttt{n} : t_n$$

if $\texttt{M}_i$ (with $i = 1, \ldots, n$) are fresh variables (that do not appear in the IF file) and every occurrence of `M` in the IF file is replaced with the term $op(\texttt{M}_1, \ldots, \texttt{M}_\texttt{n})$.

One may hence see composed types as syntactic sugar, but they allow us to write the rules for an IF file independent of the question of typing, so that the same IF specification can be analysed with respect to both the typed and the untyped model simply considering or not the signature and types sections.

Unification We define E-unification on IF terms in the standard way, i.e. unification modulo the algebraic theory E defined by the equations of the prelude file, only that types have to be respected.Formally, a unifier of two terms is a substitution, such that the type of every substituted variable agrees with the type of the term it is replaced with. (In an untyped model, the types are not considered and hence do not constrain the unification.) As we adopt the standard notion of sorted unification, we will not go into further details here but refer the reader to [7].

We use the "." as an associative, commutative, and idempotent operator, i.e. we have:

$$\begin{aligned}&t_1.(t_2.t_3) = (t_1.t_2).t_3\\&t_1.t_2 = t_2.t_1\\&t.t = t\end{aligned}$$

Note, however, that these three properties work only on facts and not on messages. With these properties, the operator "." works as a set constructor for facts, and in the following we will consequently talk about sets, union, and set difference for facts as a shorthand.

Rule Application Let us denote an IF rewrite rule by means of the triple $\langle l, exVar, r\rangle$, where l is the LHS of the rule, r is the right-hand side of the rule, and $exVar$ is the list of existentially quantified variables.

A LHS of a rule contains a set of positive and negative facts as well as a set of conditions, i.e. a set of equalities and inequalities as follows: the IF condition `equal`(t_1, t_2) represents equality of the terms t_1 and t_2, `not` represents negation of a condition, and $\leq(t_1, t_2)$ represents $t_1 \leq t_2$. For a substitution σ, we define $\sigma \models Cond$ on conditions as expected:

$$\begin{array}{lll}\sigma \models t_1 = t_2 & \text{iff} & t_1\sigma = t_2\sigma \text{ (where } t_1 \text{ and } t_2 \text{ are arbitrary terms)}\\ \sigma \models t_1 \leq t_2 & \text{iff} & t_1\sigma \leq t_2\sigma \text{ (where } t_1 \text{ and } t_2 \text{ are natural numbers)}\\ \sigma \models \phi \wedge \psi & \text{iff} & \sigma \models \phi \text{ and } \sigma \models \psi \text{ (where } \phi \text{ and } \psi \text{ are conditions)}\\ \sigma \models \neg\phi & \text{iff} & \text{not } \sigma \models \phi \text{ (where } \phi \text{ is a condition)}\end{array}$$

For the LHS l of a rule, we define the functions $PF(l)$ for the positive facts, $NF(l)$ for the negative facts, $PC(l)$ for the positive conditions (i.e. without not), and $NC(l)$ for the negative conditions.

Figure 9 defines when a rule is applicable to a (ground) state by the function *matches* that takes as argument the LHS of the rule l and yields a function that maps a state s to the set of substitutions σ such that $l\sigma$ can be applied to s (this set is empty if the rule is not applicable).[17] In there, the predicate *ground* checks that a given substitution is ground, the function *dom* returns the domain of a given substitution, and the function v returns the variables occurring in a given term. The intuition behind this definition is as follows: we consider every (ground)

$$
\begin{aligned}
matches &: Rule\_LHS \to (State \to 2^{Substitution}) \\
matches\ l\ s &= \{\sigma \mid ground(\sigma), \qquad (21)\\
&\qquad dom(\sigma) = v(PF(l)) \cup v(PC(l)), \\
&\qquad PF(l)\sigma \subseteq s,\ \ \sigma \models PC(l), \qquad (22)\\
&\qquad \forall \rho.\ dom(\rho) = (v(NF(l)) \cup v(NC(l))) \setminus dom(\sigma), \qquad (23)\\
&\qquad\quad NF(l)\sigma\rho \cap s = \emptyset,\ \sigma\rho \models NC(l)\} \qquad (24)
\end{aligned}
$$

Figure 9: Applicability of a IF rewrite rule

substitution σ (see (21)) such that under σ the positive facts can be unified with a subset of the current state (hence $PF(l)\sigma$ is necessarily ground) and the positive conditions are satisfied (see (22)). Furthermore, for all ground substitutions ρ for the remaining variables, i.e. those variables that appear only in negative facts and in negative conditions (see (23)), we postulate that none of the negative facts under $\sigma\rho$ is contained in the state and none of the conditions is satisfied for $\sigma\rho$ (see (24)). We recall that the right-hand side of the rule can only contain variables from the positive facts of the LHS and the existentially quantified variables (which will be replaced by fresh constants below), therefore all substitutions that result from *matches* are ground and so are all successor states. Note also that *matches* is applied in the same way for attack states (which are syntactically the same as a rule's LHS).

Figure 10 describes the semantics of a rule as a state-transition function. In there we use the applicability check *matches*. Besides for this check, the conditions and the negative facts of the rule do not play any role: the transition itself is concerned only with the positive *facts* of the LHS of the rule, the existentially quantified variables, and the right-hand side. Intuitively, if the rewrite rule is applicable to state s (see (25)), then its application leads to a state s' obtained from s by removing the facts in the LHS of the rule and by adding those in the right-hand side to the result (see (26)), where a different fresh constant is generated for any existentially quantified variable of the rule (see (27)). Note that here the semantics of a rule is defined as a state-transition function operating only on ground terms, i.e. s cannot contain variables (otherwise the definition of the transition relation may not behave as one would expect); the resulting s' is then also ground, as the rules cannot introduce any new variables.

[17]Note that 2^S denotes the power-set of a set S.

$$\begin{aligned}
[\![\cdot]\!] \quad &: \quad \mathit{Rule} \rightarrow (\mathit{State} \rightarrow 2^{\mathit{State}}) \\
[\![\langle l, \mathit{exVar}, r\rangle]\!](s) \quad &= \quad \{s' \mid \exists \sigma, \rho.\ \sigma \in \mathit{matches}\ \ l\ \ s, && (25)\\
& \qquad \rho = \mathtt{fresh}(s, \mathit{exVar}), && (26)\\
& \qquad s' = (s \setminus \mathit{PF}(l)\sigma) \cup r\rho\sigma\} && (27)
\end{aligned}$$

Figure 10: Semantics of a IF rewrite rule

Attack States and Properties The ground initial state(s) and a transition relation together define an infinite-state transition system. There are two ways to formulate the goals of the protocol in IF. The first is to specify attack states (which are syntactically built like the LHS of a rule). We define that a protocol is secure for an attack state g iff there is no reachable state s such that $\mathit{matches}\ \ g\ \ s$ holds. The second way (which is newly added to the IF format and not yet supported by the back-ends) is to specify temporal formulae. Due to the syntax, all permissible formulae are safety properties and can thus be checked on finite traces of events. To define the semantics, we therefore label all transitions of the transition system with the set of goal-relevant events (e.g. secret) that occurred in that transition. The protocol is secure for the formula ϕ iff every (finite) word w that the transition system accepts satisfies the formula ϕ (as defined in HLPSL).

References

[1] A. Armando and L. Compagna. Automatic SAT-Compilation of Protocol Insecurity Problems via Reduction to Planning. In *Proceedings of FORTE 2002*, LNCS 2529, pages 210–225. Springer-Verlag, 2002.

[2] A. Armando and L. Compagna. Abstraction-driven SAT-based Analysis of Security Protocols. In *Proceedings of SAT 2003*, LNCS 2919. Springer-Verlag, 2003. Available at `www.avispa-project.org`.

[3] A. Armando, L. Compagna, and P. Ganty. SAT-based Model-Checking of Security Protocols using Planning Graph Analysis. In *Proceedings of FME'2003*, LNCS 2805. Springer-Verlag, 2003.

[4] Alessandro Armando and Luca Compagna. An optimized intruder model for sat-based model-checking of security protocols. In *Proceedings of the IJCAR04 Workshop ARSPA*, 2004. To appear in ENTCS, available at `http://www.avispa-project.org`.

[5] Alessandro Armando and Luca Compagna. Satmc: a sat-based model checker for security protocols. In *Proceedings of the 9th European Conference on Logics in Artificial Intelligence (JELIA 2004)*, LNAI 3229, Lisbon, Portugal, September 2004. Springer-Verlag.

[6] AVISPA. Deliverable 3.2: Assumptions on Environment. Available at `http://www.avispa-project.org`, 2004.

[7] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[8] D. Basin, S. Mödersheim, and L. Viganò. Constraint Differentiation: A New Reduction Technique for Constraint-Based Analysis of Security Protocols. In Vijay Atluri and Peng Liu, editors, *Proceedings of CCS'03*, pages 335–344. ACM Press, 2003. Available at `http://www.avispa-project.org`.

[9] D. Basin, S. Mödersheim, and L. Viganò. OFMC: A Symbolic Model-Checker for Security Protocols. *International Journal of Information Security*, 2004.

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

[11] E. Giunchiglia, M. Maratea, A. Tacchella, and D. Zambonin. Evaluating Search Heuristics and Optimization Techniques in Propositional Satisfiability. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proceedings of IJCAR'2001*, LNAI 2083, pages 347–363. Springer-Verlag, 2001.

[12] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[13] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.

[14] M. Rusinowitch and M. Turuani. Protocol Insecurity with Finite Number of Sessions is NP-complete. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 2001.

[15] SAT. The SAT Live Web Site. `http://www.satlive.org`.

[16] Sperschneider V. and Antoniou G. *Logic, A Foundation for Computer Science*. Addison-Wesley, 1991.

[17] H. Zhang. SATO: An Efficient Propositional Prover. In W. McCune, editor, *Proceedings of CADE 14*, LNAI 1249, pages 272–275. Springer-Verlag, 1997.