AVISS — Automated Verification of Infinite State Systems

(IST-2000-26410)

Deliverable D3.6[1]

Final definition, implementation and experimentation with the SAT model checker

[1]This document supersedes Deliverable D3.3.

1 Introduction

The AVISS prototype verification tool for security protocols has the architecture shown in Figure 1. Protocols are specified in the high-level protocol specification language HLPSL, and the HLPSL2IF translator developed in WP2 translates these specifications into the low-level but tool-independent Intermediate Format, IF. The design and implementation of the three approaches that work on the IF is the task of the third work-package WP3, which consists of three steps:

- *Encoding.* Specifications in the Intermediate Format are translated into tool-specific encodings that fall into the scope of application of our three model-checkers.
- *Experiments.* The model-checkers are applied to the verification problems of the corpus [4] of security protocols.
- *Tuning.* The experiments indicate ways to improve the encodings as well as the inference strategies implemented in the available automated deduction engines.

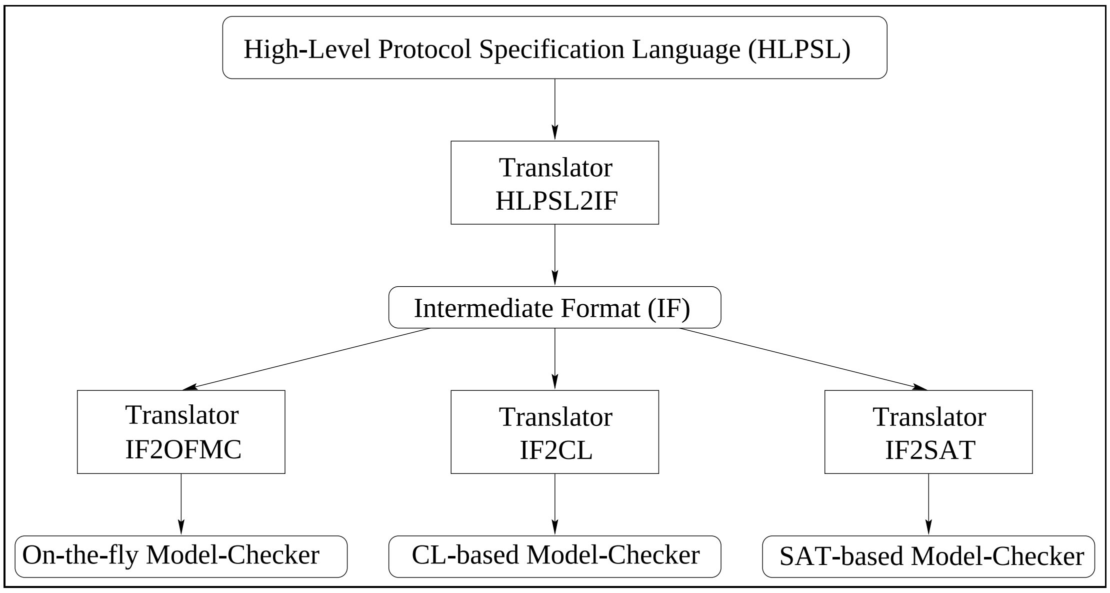

Figure 1: Architecture of the prototype verification tool

This deliverable (D3.6) reports on our implementation of task T3.3 which includes:

1. the definition of the encoding from the IF to the input format for the model-checker based on propositional satisfiability checking,
2. implementation of a prototype translator implementing the encoding,
3. experimentation with problems from the corpus, and
4. tuning of the encoding and of the model-checker based on propositional satisfiability checking.

After a brief overview in Section 2 we start in Section 3 by giving a formal definition of the notion of security problem encoded by the IF. In Section 4 we introduce the notion of planning problem, illustrate how planning problems can be encoded into propositional logic, and then—in Section 5—we describe the tool, SATE, we have developed that performs this compilation step. In Section 6 we presents our translation of security problems (encoded in the IF) into corresponding planning problems (encoded in the SATE specification language). In Section 7 we illustrate the user interface of our SAT-based model checker, and in Section 8 we present the experimental results obtained so far. In Section 9 we draw some final conclusion and discuss some future developments.

Note that a detailed comparison of the relative performance of the three tools (see [3, 7, 9]) of the AVISS project will be subject of future deliverables. A preliminary comparison on a selection of examples is given in the paper [1], which we have written as part of deliverable D4.2. See also our own paper [2], and the project webpage

`www.informatik.uni-freiburg.de/~softech/research/projects/aviss`

2 Overview

More specifically, we describe the SAT Encoder (SATE) we are currently developing and using in the context of the AVISS Project to perform SAT-based model checking of security protocols.

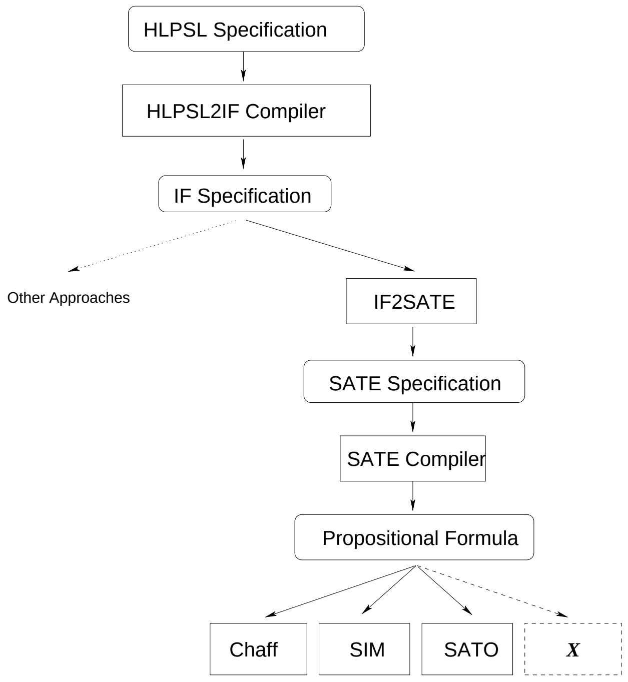

Figure 2: Architecture of the SAT-based Model-Checker

Figure 2 shows the architecture of the SAT-based Model-Checker. As in the other approaches (see Deliverables D3.4 and D3.5) the HLPSL2IF compiler translates the HLPSL specification into the Intermediate Format (IF, see Deliverable D2.2). The module IF2SATE, reads and translates a IF specification representing a security problem into an equivalent planning problem expressed in the SATE language. The SATE compiler takes as input the planning problem and some parameters (see Section 5.2) and generates a propositional formula whose satisfiability guarantees the reachability of a goal state from an initial state. By feeding the propositional formula to a SAT solver (currently Chaff [12], SIM [6], and SATO [13] are interfaced to SATE)[2] it is possible to determine its satisfiability. Since the SAT encoding computed by SATE is constructive, from any model of the propositional formula it is always possible to extract a sequence of rule applications leading from the initial state to a goal state (we call *trace* such a sequence). Therefore whenever a SAT solver finds a model for the formula, SATE extracts and displays the corresponding trace.

[2]The box labeled by $\mathcal{X}$ in Figure 2 indicates that the our architecture is *open* to the incorporation of new SAT-solvers.

3 Security Problems

A *security problem* is a tuple $\Xi = \langle \mathcal{S}, \mathcal{L}, \mathcal{R}, \mathcal{I}, \mathcal{B} \rangle$ where $\mathcal{S}$ is a set of atomic formulae of a sorted first-order language called *facts*, $\mathcal{L}$ is a set of function symbols called *rule labels*, and $\mathcal{R}$ is a set of rewrite rules of the form $L \xrightarrow{\ell} R$, where L and R are finite subsets of $\mathcal{S}$ such that the variables occurring in R occur also in L, and ℓ is an expression of the form $l(\vec{x})$ where $l \in \mathcal{L}$ and $\vec{x}$ is the vector of variables obtained by ordering lexicographically the variables occurring in L. Let S be a state and $(L \xrightarrow{\ell} R) \in \mathcal{R}$, if σ is a substitution such that $L\sigma \subseteq S$, then one possible next state of S is $S' = (S \setminus L\sigma) \cup R\sigma$ and we indicate this with $S \xrightarrow{\ell\sigma} S'$. The components $\mathcal{I}$ and $\mathcal{B}$ of a security problem are sets of states whose elements represent the initial and the bad states of the protocol respectively. We specify a security problem by means of the IF language (see Deliverable 2.2). A *solution to a security problem* Ξ (i.e. an attack to the protocol) is a sequence of states $S_1, \ldots, S_n$ such that $S_i \xrightarrow{\ell_i\sigma_i} S_{i+1}$ for $i = 1, \ldots, n$ and there exist $S_\mathcal{I} \in \mathcal{I}$, $S_\mathcal{B} \in \mathcal{B}$ such that $S_\mathcal{I} \subseteq S_1$ and $S_\mathcal{B} \subseteq S_n$.

4 Planning Problems and their Encoding into SAT

Our proposed reduction of security problems to propositional logic is carried out in two steps. Security problems are first translated into planning problems which are in turn encoded into propositional formulae.

A *planning problem* is a tuple $\Pi = \langle \mathcal{F}, \mathcal{A}, Ops, I, G \rangle$, where $\mathcal{F}$ and $\mathcal{A}$ are disjoint sets of variable-free atomic formulae of a sorted first-order language called *fluents* and *actions* respectively; *Ops* is a set of expressions of the form

$$op(Act, Pre, Add, Del)$$

where $Act \in \mathcal{A}$ and *Pre*, *Add*, and *Del* are finite sets of fluents such that $Add \cap Del = \emptyset$; I and G are boolean combinations of fluents representing the initial and the final states respectively. A state is represented by a set of fluents. An action is applicable in a state S iff the action preconditions occur in S and the application of the action leads to a new state obtained from S by removing the fluents in *Del* and adding those in *Add*. A *solution to a planning problem* Π is a sequence of actions whose execution leads from an initial to a final state and the precondition of each action appears in the state to which it applies.

4.1 Encoding Planning Problems into SAT

Let $\Pi = \langle \mathcal{F}, \mathcal{A}, Ops, I, G \rangle$ be a planning problem with finite $\mathcal{F}$ and $\mathcal{A}$ and let n be a positive integer, then it is possible to build a set of propositional formulae Φ_Π^n such that any model of Φ_Π^n corresponds to a plan for Π of length n and vice versa. The encoding of a planning problem into a set of SAT formulae can be done in a variety of ways (see [10, 5] for a survey). The basic idea is to add an additional time-index to the actions and fluents to indicate the state at which the action begins or the fluent holds. Fluents are thus indexed by 0 through n and actions by 0 through $n-1$. If p is a fluent or an action and i is an index in the appropriate range, then $i{:}p$ is the corresponding time-indexed propositional formula.

The set of formulae Φ_Π^n is the smallest set (intended conjunctively) such that:

- **Initial State Axioms:** $0{:}I \in \Phi_\Pi^n$;
- **Goal State Axioms:** $n{:}G \in \Phi_\Pi^n$;
- **Universal Axioms:** for each $op(\alpha, Pre, Add, Del) \in Ops$ and $i = 0, \ldots, n-1$:

$$(i{:}\alpha \supset \bigwedge\{i{:}p \mid p \in Pre\}) \in \Phi_\Pi^n$$
$$(i{:}\alpha \supset \bigwedge\{(i+1){:}p \mid p \in Add\}) \in \Phi_\Pi^n$$
$$(i{:}\alpha \supset \bigwedge\{\neg(i+1){:}p \mid p \in Del\}) \in \Phi_\Pi^n$$

- **Explanatory Frame Axioms:** for all fluents f and $i = 0, \ldots, n-1$:

$$(i{:}f \wedge \neg(i+1){:}f) \supset \bigvee \{i{:}\alpha \mid \mathit{op}(\alpha, \mathit{Pre}, \mathit{Add}, \mathit{Del}) \in \mathit{Ops}, f \in \mathit{Del}\} \in \Phi_{\Pi}^{n}$$
$$(\neg i{:}f \wedge (i+1){:}f) \supset \bigvee \{i{:}\alpha \mid \mathit{op}(\alpha, \mathit{Pre}, \mathit{Add}, \mathit{Del}) \in \mathit{Ops}, f \in \mathit{Add}\} \in \Phi_{\Pi}^{n}$$

- **Conflict Exclusion Axioms:** for $i = 0, \ldots, n-1$:

$$\neg(i{:}\alpha_1 \wedge i{:}\alpha_2) \in \Phi_{\Pi}^{n}$$

  for all $\alpha_1 \neq \alpha_2$ such that $\mathit{op}(\alpha_1, \mathit{Pre}_1, \mathit{Add}_1, \mathit{Del}_1) \in \mathit{Ops}$, $\mathit{op}(\alpha_2, \mathit{Pre}_2, \mathit{Add}_2, \mathit{Del}_2) \in \mathit{Ops}$, and $\mathit{Pre}_1 \cap \mathit{Del}_2 \neq \emptyset$ or $\mathit{Pre}_2 \cap \mathit{Del}_1 \neq \emptyset$.

It is immediate to see that the number of literals in Φ_{Π}^{n} is in $O(n|\mathcal{F}| + n|\mathcal{A}|)$ where $|\mathcal{F}|$ and $|\mathcal{A}|$ are the cardinalities of the sets $\mathcal{F}$ and $\mathcal{A}$, respectively. Moreover the number of Universal Axioms is in $O(nP_0|\mathcal{A}|)$ where P_0 is the maximal number of fluents mentioned in an operator (usually a small number); the number of Explanatory Frame Axioms is in $O(n|\mathcal{F}|)$; finally, the number of Conflict Exclusion Axioms is in $O(n|\mathcal{A}|^2)$.

5 SATE

To express a planning problem we are currently using the SATE specification language. The SATE compiler takes a SATE specification and, on the base of some parameters, generates a propositional formula whose satisfiability guarantees the reachability of a goal state from an initial state.

5.1 The SATE Specification Language

The SATE *specification* language allows us to specify SATE problems and it is defined by the grammar of Figure 3.

The main constructs of the language are:

- **Sort declaration** (*sort_ decl*). For example, the declarations

```
sort(fluent).
sort(action).
sort(time).
```

  declare `fluent`, `action`, and `time` to be sorts.

- **Super-Sort declaration** (*super_ sort_ decl*). For example, the declarations

```
super_sort(atom,mr).
super_sort(atom,pk).
```

  declare `atom` to be super-sort of both `mr` (sort representing the agents) and `pk` (sort representing the public and private keys).

- **Constant declaration** (*constant_ decl*). For example, the constant declarations

```
constant(0,time).
constant(s(time),time).
constant(mr(user),mr).
constant(a,user).
```

  declare `0` and `a` to be individual constants of sort `time` and `user`, respectively, and `s` and `mr` to be function symbols mapping elements of sort `time` into elements of sort `time` and of sort `user` into elements of sort `mr`, respectively.

```
                        sate_spec  ::=  ( assertion . )*
                        assertion  ::=  sort_decl | super_sort_decl | constant_decl |
                                        invariant | is_not_conflict_fluent | monotone | static
                                        initial_state | goal_state | operator
                        sort_decl  ::=  sort(sort_id)
                  super_sort_decl  ::=  super_sort(sort_id,sort_id)
                    constant_decl  ::=  constant(arity,sort_id)
                        invariant  ::=  invariant(invariant_composite_fluent)
           is_not_conflict_fluent  ::=  is_not_conflict_fluent(fluent)
                         monotone  ::=  monotone(fluent)
                           static  ::=  static(fluent)
                    initial_state  ::=  facts(init)
                       goal_state  ::=  goal(condition,goal)
                         operator  ::=  action(action,condition,pre,add,del)
                            arity  ::=  const_id | const_id(sort_id*)
                   fluent, action  ::=  const_id | const_id(term*)
                             term  ::=  var | const_id | const_id(term*)
                    pre, add, del  ::=  [fluent*]
                       init, goal  ::=  [wff*]
       invariant_composite_fluent  ::=  term = term | fluent | ~ fluent | fluent <=> fluent
                              wff  ::=  fluent | ~ wff | wff & wff | wff ∨ wff
                sort_id, const_id  ::=  [a-z,A-Z][_,a-z,A-Z,0-9]*
                              var  ::=  [_,A-Z][_,a-z,A-Z,0-9]*
```

Proviso: In the production rule for *operator*:
$Vars(cond), Vars(prec), Vars(add), Vars(del) \subseteq Vars(actions)$.

Legenda: $X^\star$ abbreviates $(X(,X)^*)$, and $X_1, \ldots, X_n ::= E$ abbreviates $X_1 ::= E \ldots X_n ::= E$.

Figure 3: Grammar for the SATE specification language

- **Invariant** (*invariant*). For example, the invariant

  ```
  invariant(~(m(_,_,A,A,_,_))).
  ```

  states, by means of the logical negation, that (at each time instant) the fields third and fourth of the above fluent are different.

- **Is not a conflict fluent** (*is_not_conflict_fluent*). For example, the declaration

  ```
  is_not_conflict_fluent(P).
  ```

  states that for all fluents `P` we can relax the conflict axioms condition.

- **Static** (*static*). For example, the declaration

  ```
  static(P).
  ```

  states that all fluents `P` are statics i.e. their initial values cannot possibly change. Formally, if p is a ground fluent instance of `P` then $0{:}p \equiv i{:}p$ for $i = 1, \ldots, n$.

- **Monotone** (*monotone*). For example, the declaration

  ```
  monotone(P).
  ```

  states that all fluents `P` can change their value from false to true, but cannot from true to false. Formally, if p is a ground fluent instance of `P` then $i{:}p \supset i+1{:}p$ for $i = 0, \ldots, n-1$.

- **Initial State** (*initial_ state*). For example, the assertion

  ```
  facts([h(s(s(s(0)))),
           wk(0,mr(intruder),mr(a),etc,1,true,1),
           wk(1,mr(a),mr(b),etc,1,true,1),
           inknw(mr(a),mr(a),1,1),
           inknw(mr(a),mr(b),2,1),
           inknw(mr(a),pk(ka),3,1),
           inknw(mr(a),primed(pk(ka)),4,1),
           inknw(mr(a),pk(kb),5,1),
           inknw(mr(b),mr(b),1,1),
           inknw(mr(b),pk(ka),2,1),
           inknw(mr(b),pk(kb),3,1),
           inknw(mr(b),primed(pk(kb)),4,1)]).
  ```

  asserts that the listed fluents hold at time 0.

- **Goal** (*goal_ state*).[3] For example, the assertion

  ```
  goal(true,[~(witness(mr(a),mr(b),X1,X2)),request(mr(b),mr(a),X1,X2)]).
  ```

  states that every state S such that it exists a substitution σ such that the fluents

  request(mr(b),mr(a),X1σ,X2σ)
  witness(mr(a),mr(b),X1σ,X2σ)

  are true and false in S, respectively, is a goal state.

- **Operator** (*operator*). For example, the assertion

  ```
  action(step_0(Xc,XKb,XKa,XB,XA,XTime),
          true,
          [h(s(XTime)),
           inknw(XA,XA,1,Xc),
           inknw(XA,XB,2,Xc),
           inknw(XA,XKa,3,Xc),
           inknw(XA,primed(XKa),4,Xc),
           inknw(XA,XKb,5,Xc),
           wk(0,mr(intruder),XA,etc,1,true,Xc)],
          [h(XTime),
           m(1,XA,XA,XB,crypt(XKb,c(nonce(c(na,XTime)),XA)),Xc),
           inknw(XA,XA,1,Xc),
           inknw(XA,XB,2,Xc),
           inknw(XA,XKa,3,Xc),
           inknw(XA,primed(XKa),4,Xc),
           inknw(XA,XKb,5,Xc),
           wk(2,XB,XA,nonce(c(na,XTime)),1,true,Xc)],
          [h(s(XTime)),
           inknw(XA,XA,1,Xc),
           inknw(XA,XB,2,Xc),
           inknw(XA,XKa,3,Xc),
           inknw(XA,primed(XKa),4,Xc),
           inknw(XA,XKb,5,Xc),
           wk(0,mr(intruder),XA,etc,1,true,Xc)]).
  ```

[3]In deliverable D3.3 we only allowed a list of fluents as argument to *goal*. Currently we have removed this restriction and we allow a list of arbitrary boolean combination of fluents as argument to `goal`.

states that if S is a state and σ a substitution such that {`h(s(XTime))`σ, `wk(0,mr(intruder),XA,etc,1,true,Xc)`σ, `inknw(XA,XA,1,Xc)`σ, `inknw(XA,XB,2,Xc)`σ, `inknw(XA,XKa,3,Xc)`σ, `inknw(XA,primed(XKa),4,Xc)`σ, `inknw(XA,XKb,5,Xc)`σ} $\subseteq S$, then ($S\setminus$ {`h(s(XTime))`σ, `wk(0,mr(intruder),XA,etc,1,true,Xc)`σ, `inknw(XA,XA,1,Xc)`σ, `inknw(XA,XB,2,Xc)`σ, `inknw(XA,XKa,3,Xc)`σ, `inknw(XA,primed(XKa),4,Xc)`σ, `inknw(XA,XKb,5,Xc)`σ})$\cup$ { `h(XTime)`σ, `wk(2,XB,XA,nonce(c(na,XTime)),1,true,Xc)`σ, `m(1,XA,XA,XB,crypt(XKb,c(nonce(c(na,XTime)),XA)),Xc)`σ, `inknw(XA,XA,1,Xc)`σ, `inknw(XA,XB,2,Xc)`σ, `inknw(XA,XKa,3,Xc)`σ, `inknw(XA,primed(XKa),4,Xc)`σ, `inknw(XA,XKb,5,Xc)`σ } is a successor state of S.

Let D_S be a SATE specification then for each sort s such that `sort(`s`)` $\in D_S$ then $||s||$ and $|s|$ are the extension (i.e. the set of ground terms such that their sort is s) and the cardinality (i.e. number of terms contained in $||s||$) of s, respectively. More precisely, $||s|| = \{t \mid$ `constant(`t`,`s`)` $\in D_S$`, and` t `is an individual constant`$\} \cup \{f(t_1,\ldots,t_k) \mid$ `constant(`$f(s_1,\ldots,s_k)$`,`s`)` $\in D_S$, $t_i \in ||s_i||$, and $i = 1,\ldots,k\} \cup \{t \mid$ `super_sort(`s`,`$subs$`)` $\in D_S$, and $t \in ||subs||\}$. It is simple to extend the concept of sort extension to the constant declarations occurring in D_S by means of

- $||$`constant(`t`,`s`)`$|| = \{t\}$ if t is an individual constant;
- $||$`constant(`$f(s_1,\ldots,s_k)$`,`s`)`$|| = \{f(t_1,\ldots,t_k) \mid t_i \in ||s_i||$, and $i = 1,\ldots,k\}$.

5.2 The SATE Compiler

The SATE compiler takes as input the SATE specification and, on the base of the following parameters:

- `steps` i.e. the bound in the number of operation applications;
- `ses_rep` i.e. the bound in the number of iterations per session;
- `multi_fresh_const` i.e. a boolean parameter. In particular, if `multi_fresh_const` is false then it is allowed the generation of a fresh constant for each fresh constant identifier occurring in the SATE specification, otherwise it is allowed the generation of a fresh constant for each possible pair composed by a fresh constant identifier and a session repetition identifier;
- `term_depth_bound` i.e. the bound on the depth of the terms during extension,[4]

generates a propositional formula whose satisfiability guarantees the reachability of a goal state from an initial state in a number of steps bounded by the number of operation applications (`steps`).

As shown in Figure 4, the SATE compiler works in three sequential phases. During the first, it analyzes the SATE specification and builds a table for both explanatory frame axioms and conflict exclusion axioms useful to speed up the other phases. In the second phase, by using the tables previously generated it produces an axioms schema file for each of the axioms encoding presented in Section 4.1. Finally, in the third phase it reads each of the above axioms schema file and generates the equivalent DIMACS (standard language used as input by state-of-the-art SAT solver to specify a propositional formula) file. This phases splitting gives us a twofold advantage. It improves the performance of the SATE compiler of 2 order of magnitude and it allows the construction of the propositional formula in an incremental way e.g. if the number of operation applications changes or if a goal declaration changes it is not necessary to repeat all the above phases but it is sufficient to repeat only a part of the third phase.

[4]Note that, if the optimizations successively presented are applied then this parameter can be, automatically, fixed.

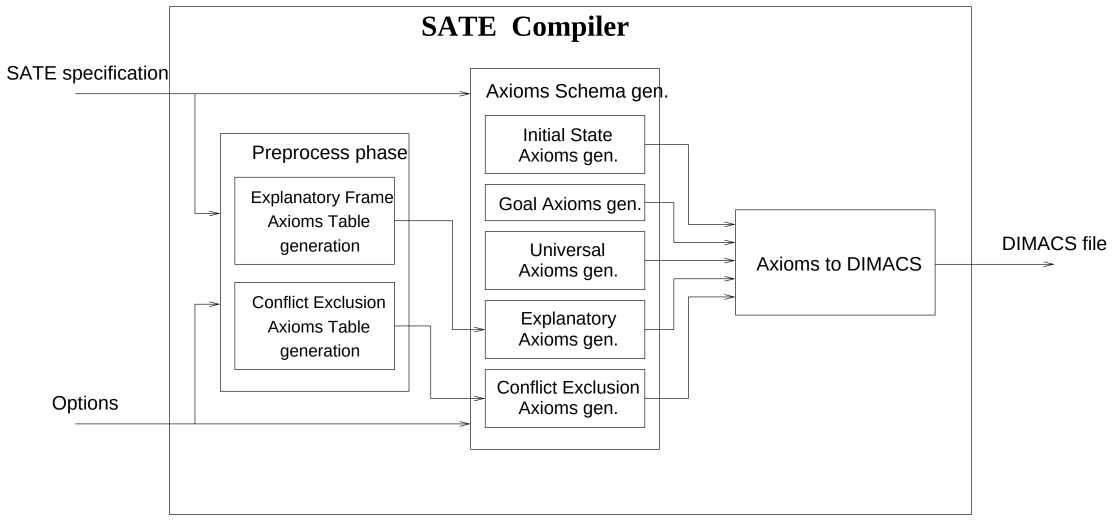

Figure 4: Architecture of the SATE Compiler

6 Security Problems as Planning Problems

Given a security problem $\Xi = \langle \mathcal{S}, \mathcal{L}, \mathcal{R}, \mathcal{I}, \mathcal{B} \rangle$, it is possible to build a planning problem $\Pi_\Xi = \langle \mathcal{F}_\Xi, \mathcal{A}_\Xi, Ops_\Xi, I_\Xi, G_\Xi \rangle$ such that each solution to Π_Ξ can be translated back to a solution to Ξ: $\mathcal{F}_\Xi$ is the set of facts $\mathcal{S}$; $\mathcal{A}_\Xi$ and Ops_Ξ are the smallest sets such that $\ell\sigma \in \mathcal{A}_\Xi$ and $op(\ell\sigma, L\sigma, R\sigma, L\sigma \setminus R\sigma) \in Ops$ for all $(L \xrightarrow{\ell} R) \in \mathcal{R}$ and all ground substitutions σ; finally $I_\Xi = \bigvee_{S_\mathcal{I} \in \mathcal{I}} \bigwedge S_\mathcal{I}$ and $G_\Xi = \bigvee_{S_\mathcal{B} \in \mathcal{B}} \bigwedge S_\mathcal{B}$.

We now define the translation from IF (security problem) to the SATE (planning problem) specification language. First we present a simple translation and then we show some optimizations. Our goal is to convert a IF file representing a security protocol into a SATE file representing the same protocol. In order to do this we have to declare the SATE constructs (sorts, super-sorts, constants, invariants, initial state, goal, operators, etc.) characterizing a protocol. Some of these declarations are independent from the protocol. For example, the following declarations,

```
sort(bool).
constant(true,bool).
constant(false,bool).
```

are independent from the protocol, because the sort `bool` representing the boolean data type is used in each protocol. Figure 5 shows the protocol independent super-sort and constant declarations.[5]

In addition to these we have to define the protocol dependent declarations. Let $\mathcal{S}$ be the IF specification and $D_\mathcal{S}$ be the set of protocol dependent declarations; then:

- if t is a IF term, then $VAR(t)$ is the function returning the set of IF variables occurring in t; for example, $VAR(\texttt{h(xTime)}) = \{\texttt{xTime}\}$;
- if x is a IF variable occurring in $\mathcal{S}$, then $SORT(x, S)$ is the function returning the SATE sort of x on the base of the position of it inside the specification $\mathcal{S}$; for example, if `h(xTime)` occurs in $\mathcal{S}$, then $SORT(\texttt{xTime}, \mathcal{S}) = \texttt{time}$;
- if t is a IF syntactic construct, then $(\![t]\!)$ is the corresponding syntactic construct of SATE such that:

[5]To increase the readability, the sort declarations are not given, but they can be easily inferred by the other declarations. For example, `super_sort(knw_el,msg)` implies that `knw_el` and `msg` are sorts and `constant(crypt(msg,msg),crypt)` implies that `crypt` and `msg` are sorts.

```
super_sort(knw_el,msg).

super_sort(msg,binary).
super_sort(msg,unary).

super_sort(binary,crypt).
super_sort(binary,scrypt).
super_sort(binary,pair).
super_sort(binary,funct).
super_sort(binary,rcrypt).

super_sort(unary,atom).

super_sort(atom,mr).
super_sort(atom,pk).
super_sort(atom,sk).
super_sort(atom,nonce).
super_sort(atom,fu).

super_sort(ped,pk).
super_sort(ped,pkinv).
super_sort(public_key,public_key_id).
super_sort(public_key,fresh_public_key).
super_sort(public_key,fresh_intruder_const).

super_sort(symmetric_key,symmetric_key_id).
super_sort(symmetric_key,fresh_symmetric_key).
super_sort(symmetric_key,fresh_intruder_const).

super_sort(fresh_nonce,fresh_intruder_const).


constant(0,time).
constant(s(time),time).
constant(true,bool).
constant(false,bool).
```

```
constant(s(session),session).
constant(f(session),fsecrecy).
constant(h(time),fluent).
constant(w(wstep,mr,mr,knw,knw,bool,session),fluent).
constant(m(mstep,mr,mr,mr,msg,session),fluent).
constant(i(knw_el),fluent).
constant(witness(mr,mr,authentication_id,knw_el),fluent).
constant(request(mr,mr,authentication_id,knw_el),fluent).
constant(secret(knw_el,fsecrecy),fluent).
constant(give(knw_el,fsecrecy),fluent).

constant(etc,knw).
constant(c(knw_el,knw),knw).
constant(etc,etc).

constant(crypt(msg,msg),crypt).
constant(scrypt(msg,msg),scrypt).
constant(funct(msg,msg),funct).
constant(c(msg,msg),pair).
constant(rcrypt(msg,msg),rcrypt).

constant(pk(public_key),pk).
constant(primed(pkinv),pkinv).
constant(c(fresh_public_id,time),fresh_public_key).

constant(sk(symmetric_key),sk).
constant(c(fresh_symmetric_id,time),fresh_symmetric_key).

constant(nonce(fresh_nonce),nonce).
constant(nonce(nonce_id),nonce).
constant(c(fresh_nonce_id,time),fresh_nonce).

constant(fu(fu_term),fu).
constant(mr(user),mr).

constant(c(intruder_const,intruder_const),fresh_intruder_const).
```

Figure 5: Protocol independent declarations

  - if *id* is an IF identifier different from `I` and the first character of *id* is an upper-case letter then $(\![id]\!)$ is the identifier obtained from *id* by replacing the first character with the corresponding lower-case letter; otherwise, if $id =$ `I` (and hence represents the IF intruder constant) then $(\![id]\!) =$ `intruder`, otherwise $(\![id]\!) = id$; for example, $(\![$`Se01`$]\!) =$ `se01`, $(\![$`I`$]\!) =$ `intruder`, and $(\![$`etc`$]\!) =$ `etc`;
  - if v is a IF variable (hence it begins with the character `x`) then $(\![v]\!)$ is the variable obtained from v by replacing the first character with `X`; for example, $(\![$`xA`$]\!) =$ `XA`;
  - $(\![f(t_1,\ldots,t_n)]\!) = f((\![t_1]\!),\ldots,(\![t_n]\!))$ for each other function symbol f.

- for each IF constant symbol s (hence the first character of s is not the `x` character)
  - if `mr(`s`)` occurs in $\mathcal{S}$ and s is not the constant `I`, then `constant(`$(\![s]\!)$`,user_honest)` $\in D_{\mathcal{S}}$;
  - if `mr(I)` occurs in $\mathcal{S}$, then `constant(intruder,user_intruder)` $\in D_{\mathcal{S}}$;
  - if `pk(`s`)` occurs in $\mathcal{S}$, then `constant(`$(\![s]\!)$`,public_key_id)` $\in D_{\mathcal{S}}$;
  - if `sk(`s`)` occurs in $\mathcal{S}$, then `constant(`$(\![s]\!)$`,symmetric_key_id)` $\in D_{\mathcal{S}}$;
  - if `fu(`s`)` occurs in $\mathcal{S}$, then `constant(`$(\![s]\!)$`,fu_term)` $\in D_{\mathcal{S}}$;
  - if `nonce(`s`)` occurs in $\mathcal{S}$, then `constant(`$(\![s]\!)$`,nonce_id)` $\in D_{\mathcal{S}}$;
  - if `nonce(c(`s`,xTime))` occurs in $\mathcal{S}$, then `constant(`$(\![s]\!)$`,fresh_nonce_id)` $\in D_{\mathcal{S}}$;
  - if `pk(c(`s`,xTime))` occurs in $\mathcal{S}$, then `constant(`$(\![s]\!)$`,fresh_public_key_id)` $\in D_{\mathcal{S}}$;
  - if `sk(c(`s`,xTime))` occurs in $\mathcal{S}$, then `constant(`$(\![s]\!)$`,fresh_symmetric_key_id)` $\in D_{\mathcal{S}}$;
  - if `c(`s`,`s`)` occurs in $\mathcal{S}$, then `constant(`$(\![s]\!)$`,intruder_const)` $\in D_{\mathcal{S}}$;
  - if `witness(_,_,`s`,_)` occurs in $\mathcal{S}$, then `constant(`$(\![s]\!)$`,authentication_id)` $\in D_{\mathcal{S}}$;

- for each IF rewrite rule

  `# lb=_, type=Protocol_Rules` $t_1 \ldots t_n$ `=>` $t_{n+1} \ldots t_{n+m}$

  occurring in $\mathcal{S}$ and for each $k, j = 1,\ldots,n+m$ such that $t_k =$ `m(`mi`,_,_,_,_,_)`, $t_j =$ `w(`wi`,_,_,_,_,_,_)`, then $\{$`constant(`mi`,mstep)`, `constant(`wi`,wstep)`$\} \in D_{\mathcal{S}}$;

- for each

  `# lb=_, type=Init` $t_1 \ldots t_n$

  occurring in $\mathcal{S}$, then `facts([`$(\![t_1]\!),\ldots,(\![t_n]\!)$`])` $\in D_{\mathcal{S}}$;

- for each rewrite rule

  `# lb=`*label*`, type=`*cat* $lt_1 \ldots lt_n$ `=>` $rt_1 \ldots rt_m$

  occurring in $\mathcal{S}$ such that *cat* $\in\{$`Protocol_Rules, Intruder_Rules, Decomposition_Rules`$\}$ (see Deliverable D2.2), then

```
action(label((|x_1|),...,(|x_k|)),true,
        [(|lt_1|),...,(|lt_n|)],
        [at_1,...,at_{m-j}],
        [dt_1,...,dt_{n-j}]
) ∈ D_S
```

  and `constant(`*label*`(`$s_1,\ldots,s_k$`),action)` $\in D_{\mathcal{S}}$, where:

  - $\{x_1,\ldots,x_k\} = VAR(lt_1) \cup \cdots \cup VAR(lt_n) \cup VAR(rt_1) \cup \cdots \cup VAR(rt_m)$,
  - $s_i = SORT(x_i, \mathcal{S})$ for each $i = 1,\ldots,k$,
  - $\{at_1,\ldots,at_{m-j}\} \equiv \{(\![rt_1]\!),\ldots,(\![rt_m]\!)\} \setminus \{(\![lt_1]\!),\ldots,(\![lt_n]\!)\}$, and

  – $\{dt_1, \ldots, dt_{n-j}\} \equiv \{(\![lt_1]\!), \ldots, (\![lt_n]\!)\} \setminus \{(\![rt_1]\!), \ldots, (\![rt_m]\!)\}$;

- for each

  `# lb=_, type=Goal` $t_1 \ldots t_n$

  occurring in $\mathcal{S}$ then `goal(true,[`$(\![t_1]\!), \ldots, (\![t_n]\!)$`])` $\in D_{\mathcal{S}}$;

The number of ground instances of fluents and of ground instances of actions directly affect the number of clauses and of propositional variables generated by the SATE module (see section 4.1 and [5]). The ground instances of fluents (actions) are given by extension of the constant declarations relative to the sort `fluent` (`action`). For example the extension of the declaration

```
constant(m(mstep,mr,mr,mr,msg,session),fluent)
```

generates a number of fluents equal to

$$|\texttt{mstep}| * |\texttt{mr}|^3 * |\texttt{msg}| * |\texttt{session}|$$

where $|s|$ is the number of ground terms of sort s.

A direct application of the above approach (namely the reduction of a security problem Ξ to a planning problem Π_Ξ followed by a SAT-compilation of Π_Ξ) is not immediately usable, because it specifies a language too big for the applicability of the SAT encoding. In order to decrease the number of ground fluents and actions we apply the optimizations presented in the following subsections.

6.1 Invariants

Invariants are an effective way to decrease the number of ground instances of fluents. The idea is to use some invariants for describing what ground instances of fluents are allowed and what are not. For example, to state that the fields official sender and receiver in a message term must be different we can use:

```
invariant(~(m(_,_,A,A,_,_)))
```

where the symbol `~` is the logic negation. In the same way, to avoid the ground instances of fluents in which the fields sender and receiver of a principal, witness or request term are equal, we can use:

```
invariant(~(w(_,A,A,_,_,_,_)))
invariant(~(witness(A,A,_,_)))
invariant(~(request(A,A,_,_)))
```

Applying the above invariants the number of ground instances of fluents generated by the declarations

```
constant(m(mstep,mr,mr,mr,msg,session),fluent)
constant(w(wstep,mr,mr,knw,knw,bool,session),fluent)
constant(witness(mr,mr,authentication_id,knw_el),fluent)
constant(request(mr,mr,authentication_id,knw_el),fluent)
```

become respectively:

$$|\texttt{mstep}| * |\texttt{mr}|^2 * (|\texttt{mr}| - 1) * |\texttt{msg}| * |\texttt{session}|$$
$$|\texttt{wstep}| * |\texttt{mr}| * (|\texttt{mr}| - 1) * |\texttt{knw}|^2 * |\texttt{bool}| * |\texttt{session}|$$
$$|\texttt{mr}| * (|\texttt{mr}| - 1) * |\texttt{authentication\_id}| * |\texttt{knw\_el}|$$
$$|\texttt{mr}| * (|\texttt{mr}| - 1) * |\texttt{authentication\_id}| * |\texttt{knw\_el}|$$

6.2 Protocol Dependent Messages

Note that in Figure 5 the declarations about the messages (`msg`) are very general and can be specialized. In fact the extension of these declarations contains a lot of useless terms i.e. those messages which are not allowed by the protocol. The idea is to restrict the generated terms on the base of the messages allowed by the protocol. In order to do this we analyze the IF specification $\mathcal{S}$ to extract the patterns of the possible exchanged messages allowed by the protocol. The translation from IF to SATE changes and, in particular, for each rewrite rule

$$\texttt{\# lb=\_, type=Protocol\_Rules } t_1 \ldots t_n \texttt{ => } t_{n+1} \ldots t_{n+m}$$

occurring in $\mathcal{S}$ and for each $k = 1, \ldots, n+m$ such that $t_k = \texttt{m(}i, \_, \_, \_, msg, \_\texttt{)}$ then $Cs \cup$ {`constant(m(mstep_`i`,mr,mr,mr,msg_`i`,session),fluent)`, `sort(msg_`i`)`, `sort(mstep_`i`)`, `super_sort(msg_`i`,`s`)`} $\in D_{\mathcal{S}}$ where Cs and s are computed by `preprocess_msg(`$msg, i, s, Cs, 1, 1$`)`. This predicate (partially given in Figure 7) analyzes the IF message msg computing s, i.e. the sort of msg (previous a concatenation with i and its position in a tree representing the pattern of msg), and Cs, i.e. the list of declarations useful to define the SATE sub-language relative to msg. By representing the constant declarations of Cs in a tree each node $n_{i,j}$ (i and j are the vertical and the horizontal position, resp.), storing a symbol $f_{i,j}$ and a SATE sort $s_{i,j}$, characterizes the SATE terms contained in the sort extension $\|s_{i,j}\|$ by means of:

- if $n_{i,j}$ is a leaf and $f_{i,j} \neq \_$ then $\|s_{i,j}\| = \{f_{i,j}\}$;
- if $n_{i,j}$ is a leaf and $f_{i,j} = \_$ then $\|s_{i,j}\|$ is built as presented in Section 5.1;
- if $n_{i,j}$ has only a child then $\|s_{i,j}\| = \{f_{i,j}(l) \mid l \in \|s_{i+1,j*2-1}\| \}$;
- if $n_{i,j}$ has two children then $\|s_{i,j}\| = \{f_{i,j}(l,r) \mid l \in \|s_{i+1,j*2-1}\|$, and $r \in \|s_{i+1,j*2}\| \}$;

where the symbol _ in $n_{i,j}$ represents every term of sort $s_{i,j}$.

Let us consider the example of the Needham-Schroeder Public-Key (NSPK) protocol [4]. The IF rewrite rules relative to its protocol steps are given in Figure 8[6]. In particular, let us consider the message term (it is contained in the right-hand-side of the first rewrite rule)

```
m(1,mr(a),mr(a),mr(b),crypt(pk(kb),c(nonce(c(na,xTime)),mr(a))),xc)
```

The application of the previously described procedure leads to the invocation of

```
preprocess_msg(crypt(pk(kb),c(nonce(c(na,xTime)),mr(a))),1,SORT,Cs,1,1).
```

that is satisfied by means of the following assignments:[7]

```
SORT = crypt_1_1_1,
Cs = [
  constant(crypt(pk_1_2_1,pair_1_2_2),crypt_1_1_1),
  constant(c(nonce_1_3_3,mr_1_3_4),pair_1_2_2),
  constant(nonce(fresh_term_1_4_5),nonce_1_3_3),
  constant(c(fresh_nonce_id_1_5_9,time),fresh_term_1_4_5),
  constant(na,fresh_nonce_id_1_5_9),
  constant(mr(user_1_4_7),mr_1_3_4),
  constant(a,user_1_4_7),
  constant(pk(public_key_id_1_3_1),pk_1_2_1),
  constant(kb,public_key_id_1_3_1),
  ...
```

The tree relative to these constant declarations is shown in Figure 6.[8] Note that a path from a

[6]In the rest of the document we use the prolog list representation to increase the readability. Note that `[]` is equivalent to `etc`,`[x]` is equivalent to `c(x,etc)` and so on.

[7]To increase the readability we present only the constant declarations.

[8]Note that the sort characterizing the node is between brackets.

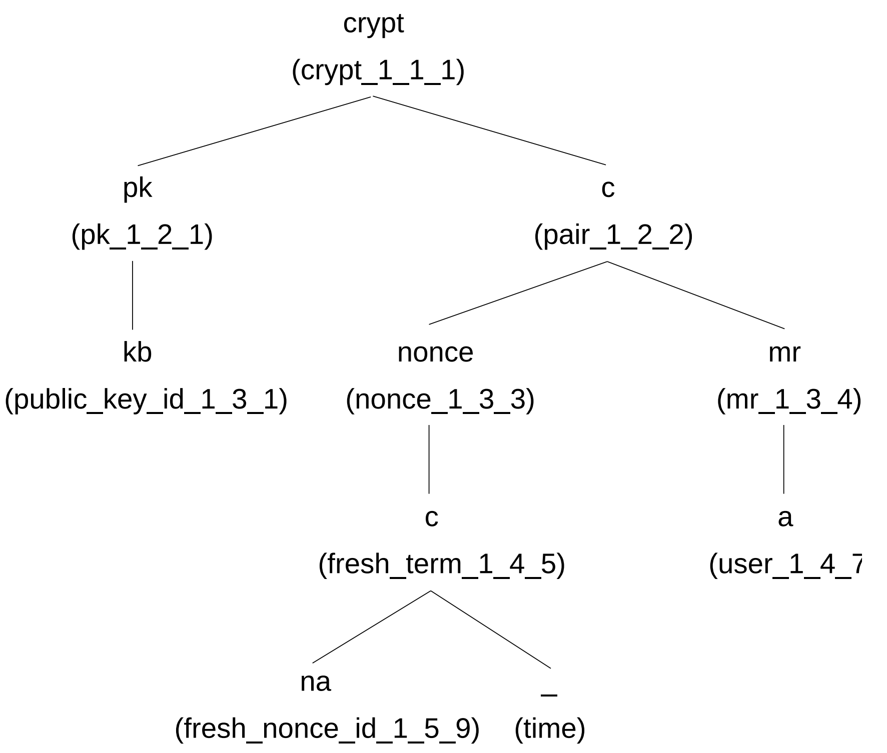

Figure 6: Tree representation of a message

node to a leaf gives a term of the sort relative to that node. For example, `pk(kb)` ∈ ||`pk_1_2_1`||, and `crypt(pk(kb),c(nonce(c(na,s(0))),mr(a)))` ∈ ||`crypt_1_1_1`||.

By repeating the above procedure for each message terms in the protocol rules of the NSPK we specify the SATE language for each possible message allowed by the NSPK protocol. Finally, the following declarations are added to D_S:

```
sort(msg_1).
sort(mstep_1).
super_sort(msg_1,crypt_1_1_1).
constant(m(mstep_1,mr,mr,mr,msg_1,session),fluent).
...

sort(msg_2).
sort(mstep_2).
super_sort(msg_2,crypt_2_1_1).
constant(m(mstep_2,mr,mr,mr,msg_2,session),fluent).
...
sort(msg_3).
sort(mstep_3).
super_sort(msg_3,crypt_3_1_1).
constant(m(mstep_3,mr,mr,mr,msg_3,session),fluent)
...
```

Note that using the general declaration of Figure 5, the number of ground instances of fluents relative to the message terms was:

$$|\texttt{mstep}| * |\texttt{mr}|^3 * |\texttt{msg}| * |\texttt{session}|$$

```
procedure preprocess_msg($msg$,$lb$,$sort$,$Ds$,$d$,$w$)
   inputs: $msg$, a IF message
           $lb$, a label
           $sort$, the sort relative to the message $msg$
           $Ds$, a set of SATE declarations
           $d$, the vertical position (depth) of $msg$
           $w$, the horizontal position (width) of $msg$
case
      $msg$ == crypt($submsg_1$,$submsg_2$) :
          $nd \leftarrow d + 1$
          $lw \leftarrow w * 2 - 1$
          $rw \leftarrow w * 2$
          preprocess_msg($submsg_1$,$lb$,$submsgsort_1$,$Ds$,$nd$,$lw$)
          preprocess_msg($submsg_2$,$lb$,$submsgsort_2$,$Ds$,$nd$,$rw$)
          constant(crypt($submsgsort_1$,$submsgsort_2$),crypt_$lb$_$d$_$w$) $\in Ds$
          super_sort(knw_el,crypt_$lb$_$d$_$w$) $\in Ds$
          $sort \leftarrow$ crypt_$lb$_$d$_$w$
          break
      $msg$ == c($submsg_1$,$submsg_2$) :
          $nd \leftarrow d + 1$
          $lw \leftarrow w * 2 - 1$
          $rw \leftarrow w * 2$
          preprocess_msg($submsg_1$,$lb$,$submsgsort_1$,$Ds$,$nd$,$lw$)
          preprocess_msg($submsg_2$,$lb$,$submsgsort_2$,$Ds$,$nd$,$rw$)
          constant(c($submsgsort_1$,$submsgsort_2$),pair_$lb$_$d$_$w$) $\in Ds$
          super_sort(knw_el,pair_$lb$_$d$_$w$) $\in Ds$
          $sort \leftarrow$ pair_$lb$_$d$_$w$
          break
      $msg$ == pk($submsg$) and $submsg$ is an IF variable:
          $sort \leftarrow$ pk
          break
      $msg$ == pk($submsg$) and $submsg$ is an IF constant:
          $nd \leftarrow d + 1$
          $lw \leftarrow w * 2 - 1$
          constant(pk(public_key_id_$lb$_$nd$_$lw$),pk_$lb$_$d$_$w$) $\in Ds$
          constant($submsg$,public_key_id_$lb$_$nd$_$lw$) $\in Ds$
          super_sort(knw_el,pk_$lb$_$d$_$w$) $\in Ds$
          $sort \leftarrow$ pk_$lb$_$d$_$w$
          break
      $msg$ == nonce(c($submsg$,xTime)) and $submsg$ is an IF constant :
          $nd \leftarrow d + 1$
          $lw \leftarrow w * 2 - 1$
          $nnd \leftarrow nd + 1$
          $llw \leftarrow lw * 2 - 1$
          constant(nonce(fresh_term_$lb$_$nd$_$lw$),nonce_$lb$_$d$_$w$) $\in Ds$
          constant(c(fresh_nonce_id_$lb$_$nnd$_$llw$,time),fresh_term_$lb$_$nd$_$lw$) $\in Ds$
          constant($submsg$,fresh_nonce_id_$lb$_$nnd$_$llw$) $\in Ds$
          super_sort(knw_el,nonce_$lb$_$d$_$w$) $\in Ds$
          $sort \leftarrow$ nonce_$lb$_$d$_$w$
          break
          $sort \leftarrow$ nonce
          break
      $msg$ == nonce($submsg$) and $submsg$ is an IF variable :
          $sort \leftarrow$ nonce
          break
esac
```

Figure 7: Pseudo-code of the procedure `preprocess_msg`

```
# lb=Step_0_1, type=Protocol_Rules
h(s(xTime)).
w(0,mr(I),mr(a),[],[mr(a),mr(b),pk(ka),pk(ka)',pk(kb)],true,xc)
=>
h(xTime).
m(1,mr(a),mr(a),mr(b),crypt(pk(kb),c(nonce(c(na,xTime)),mr(a))),xc).
w(2,mr(b),mr(a),[nonce(c(na,xTime))],[mr(a),mr(b),pk(ka),pk(ka)',pk(kb)],true,xc)

# lb=Step_0_2, type=Protocol_Rules
h(s(xTime)).
w(0,mr(I),mr(a),[],[mr(a),mr(I),pk(ka),pk(ka)',pk(ki)],true,xc)
=>
h(xTime).
m(1,mr(a),mr(a),mr(I),crypt(pk(ki),c(nonce(c(na,xTime)),mr(a))),xc).
w(2,mr(I),mr(a),[nonce(c(na,xTime))],[mr(a),mr(I),pk(ka),pk(ka)',pk(ki)],true,xc)

# lb=Step_1_1, type=Protocol_Rules
h(s(xTime)).
m(1,mr(xr),mr(a),mr(b),crypt(pk(kb),c(nonce(xna),mr(a))),xc2).
w(1,mr(a),mr(b),[],[mr(a),mr(b),pk(ka),pk(kb),pk(kb)'],xbool,xc)
=>
h(xTime).
m(2,mr(b),mr(b),mr(a),crypt(pk(ka),c(nonce(xna),nonce(c(nb,xTime)))),xc2).
w(3,mr(a),mr(b),[crypt(pk(kb),c(nonce(xna),mr(a))),nonce(c(nb,xTime))],
                [mr(a),mr(b),pk(ka),pk(kb),pk(kb)'],true,xc)

# lb=Step_2_1, type=Protocol_Rules
h(s(xTime)).
m(2,mr(xr),mr(b),mr(a),crypt(pk(ka),c(nonce(xna),nonce(xnb))),xc2).
w(2,mr(b),mr(a),[nonce(xna)],[mr(a),mr(b),pk(ka),pk(ka)',pk(kb)],xbool,xc)
=>
h(xTime).
m(3,mr(a),mr(a),mr(b),crypt(pk(kb),nonce(xnb)),xc2).
w(0,mr(I),mr(a),[],[mr(a),mr(b),pk(ka),pk(ka)',pk(kb)],true,s(xc))

# lb=Step_2_2, type=Protocol_Rules
h(s(xTime)).
m(2,mr(xr),mr(I),mr(a),crypt(pk(ka),c(nonce(xna),nonce(xnb))),xc2).
w(2,mr(I),mr(a),[nonce(xna)],[mr(a),mr(I),pk(ka),pk(ka)',pk(ki)],xbool,xc)
=>
h(xTime).
m(3,mr(a),mr(a),mr(I),crypt(pk(ki),nonce(xnb)),xc2).
w(0,mr(I),mr(a),[],[mr(a),mr(I),pk(ka),pk(ka)',pk(ki)],true,s(xc))

# lb=Step_3_1, type=Protocol_Rules
h(s(xTime)).
m(3,mr(xr),mr(a),mr(b),crypt(pk(kb),nonce(xnb)),xc2).
w(3,mr(a),mr(b),[crypt(pk(kb),c(nonce(xna),mr(a))),nonce(xnb)],
                [mr(a),mr(b),pk(ka),pk(kb),pk(kb)'],xbool,xc)
=>
h(xTime).
w(1,mr(a),mr(b),[],[mr(a),mr(b),pk(ka),pk(kb),pk(kb)'],true,s(xc))
```

Figure 8: Protocol rewrite rules of the NSPK protocol

while using this optimization that number decreases to the value:

$$(|\texttt{mstep\_1}| * |\texttt{msg\_1}| + |\texttt{mstep\_2}| * |\texttt{msg\_2}| + |\texttt{mstep\_3}| * |\texttt{msg\_3}|) * |\texttt{mr}|^3 * |\texttt{session}| = (|\texttt{msg\_1}| + |\texttt{msg\_2}| + |\texttt{msg\_3}|) * |\texttt{mr}|^3 * |\texttt{session}|$$

Since

$$|\texttt{mstep}| * |\texttt{msg}| \gg |\texttt{msg\_1}| + |\texttt{msg\_2}| + |\texttt{msg\_3}|$$

the number of ground instances of fluents decreases considerably. If we assume that the NSPK protocol is executed by three agents with the respective public and private keys and two nonces, then we have $|\texttt{mstep}| * |\texttt{msg}| \simeq 5,690,000$, while $|\texttt{msg\_1}| + |\texttt{msg\_2}| + |\texttt{msg\_3}| \simeq 1,000$.

6.3 Elimination of the Boolean Field from Principal Terms

The boolean field in the principal term is useful to avoid useless repetitions of IF impersonate rule. For example the rule

```
# lb=Impersonate_1_1, type=Intruder_Rules
h(s(xTime)).
w(1,mr(xA),mr(xB),etc,c(mr(xA),c(mr(xB),c(pk(xKa),c(pk(xKb),c(pk(xKb)',etc))))),true,xc).
i(mr(xA)).
i(mr(xB)).
i(crypt(pk(xKb),c(nonce(xNa),mr(xA))))
=>
h(xTime).
m(1,mr(I),mr(xA),mr(xB),crypt(pk(xKb),c(nonce(xNa),mr(xA))),0).
w(1,mr(xA),mr(xB),etc,c(mr(xA),c(mr(xB),c(pk(xKa),c(pk(xKb),c(pk(xKb)',etc))))),false,xc).
i(mr(xA)).
i(mr(xB)).
i(crypt(pk(xKb),c(nonce(xNa),mr(xA))))
```

can be applied iff the boolean field in the principal term is true. This kind of tricks are crucial for decreasing the branching factor of the search space, but in our case the branching factor seems to be not so critical as the language size.[9] Hence we eliminate the boolean field from the principal term i.e. we use the following:

- for each principal term then

$$(\!|\texttt{w}(s, r, ak, ik, \_, ses)|\!) = \texttt{w}((\!|s|\!),(\!|r|\!),(\!|ak|\!),(\!|ik|\!),(\!|ses|\!))$$

It is simple to see that the above assumption halves the size of the language relative to the principal terms.

6.4 Bounding the Number of Session Runs

Let s and j be the bound in the number of operation applications and the number of protocol steps characterizing a protocol session, then the maximum number of times a session can be repeated is $max\_ses\_rep = \lceil s/j \rceil$. Our experience shows that the protocol attacks require a number of session repetitions (ses_rep) that is less than max_ses_rep. The role of the `session_repetitions` parameter is to bound the number of sessions runs to be considered during the analysis of the protocol. For example, the NSPK protocol is characterized by 3 protocol steps ($j = 3$), the attack is found with $s = 10$ and $ses\_rep = 1$, while $max\_ses\_rep = \lceil 10/3 \rceil = 4$. By using this

[9]In theory the lack of those tricks and therefore a bigger branching factor should increase the time of the SAT solving phase, but in practice it does not.

optimization we reduce the cardinality of the sort `session` (in the case of the NSPK protocol, we reduce it of a factor 4) and therefore the number of fluents and actions which are dependent from it.

6.5 Knowledge Splitting

The idea of the knowledge splitting comes from some features of the acquired and initial knowledge. Specifically we can note that:

1. the domain of the initial knowledge is fixed by the initial state and this domain is a subset of the domain relative to the acquired knowledge;
2. the initial knowledge is a static[10] property of an agent involved in a protocol session;
3. the ordering of the terms in both kinds of knowledge is relevant.

Feature 1 suggests to use two different sorts for the initial and the acquired knowledge, feature 2 suggests to define a new fluent relative to the initial knowledge, and feature 3 suggests to use a sequence of fluents containing one single knowledge term and an index stating the position of this term in the knowledge rather than to use lists.

The protocol dependent declarations change on the base of the following:

- if t is a IF session term, then $\llbracket t \rrbracket$ is the SATE representation of the session base of t such that:
  - $\llbracket \mathtt{s}(ses) \rrbracket = \llbracket ses \rrbracket$;
  - if ses is a IF constant then $\llbracket ses \rrbracket = \llparenthesis ses \rrparenthesis$;
  - if ses is a IF variable then $\llbracket ses \rrbracket = ses'$ where ses' is the concatenation of the sequence of chars `NEW` and of $\llparenthesis ses \rrparenthesis$; for example $\llbracket xc \rrbracket =$`NEWXc`;
- for each principal term such that the initial and acquired knowledge are empty then
$$\llparenthesis \mathtt{w}(ws, s, r, \mathtt{[]}, \mathtt{[]}, \_, ses) \rrparenthesis = \begin{array}{r} \mathtt{wk}(\llparenthesis ws \rrparenthesis, \llparenthesis s \rrparenthesis, \llparenthesis r \rrparenthesis, \mathtt{etc}, 1, \llparenthesis ses \rrparenthesis), \\ \mathtt{inknw}(\llparenthesis r \rrparenthesis, \mathtt{etc}, 1, \llbracket ses \rrbracket) \end{array}$$
- for each principal term such that the initial knowledge is empty and $k \geq 1$ then
$$\llparenthesis \mathtt{w}(ws, s, r, [ak_1, \ldots, ak_k], \mathtt{[]}, \_, ses) \rrparenthesis = \begin{array}{r} \mathtt{wk}(\llparenthesis ws \rrparenthesis, \llparenthesis s \rrparenthesis, \llparenthesis r \rrparenthesis, \llparenthesis ak_1 \rrparenthesis, 1, \llparenthesis ses \rrparenthesis), \\ \vdots \qquad\qquad \\ \mathtt{wk}(\llparenthesis ws \rrparenthesis, \llparenthesis s \rrparenthesis, \llparenthesis r \rrparenthesis, \llparenthesis ak_k \rrparenthesis, n, \llparenthesis ses \rrparenthesis), \\ \mathtt{inknw}(\llparenthesis r \rrparenthesis, \mathtt{etc}, 1, \llbracket ses \rrbracket) \end{array}$$
- for each principal term such that the acquired knowledge is empty and $l \geq 1$ then
$$\llparenthesis \mathtt{w}(ws, s, r, \mathtt{[]}, [ik_1, \ldots, ik_l], \_, ses) \rrparenthesis = \begin{array}{r} \mathtt{wk}(\llparenthesis ws \rrparenthesis, \llparenthesis s \rrparenthesis, \llparenthesis r \rrparenthesis, \mathtt{etc}, 1, \llparenthesis ses \rrparenthesis), \\ \mathtt{inknw}(\llparenthesis r \rrparenthesis, \llparenthesis ik_1 \rrparenthesis, 1, \llbracket ses \rrbracket), \\ \vdots \qquad\qquad \\ \mathtt{inknw}(\llparenthesis r \rrparenthesis, \llparenthesis ik_l \rrparenthesis, m, \llbracket ses \rrbracket) \end{array}$$
- for each principal term such that $k \geq 1$ and $l \geq 1$ then
$$\llparenthesis \mathtt{w}(ws, s, r, [ak_1, \ldots, ak_k], [ik_1, \ldots, ik_l], \_, ses) \rrparenthesis = \begin{array}{r} \mathtt{wk}(\llparenthesis ws \rrparenthesis, \llparenthesis s \rrparenthesis, \llparenthesis r \rrparenthesis, \llparenthesis ak_1 \rrparenthesis, 1, \llparenthesis ses \rrparenthesis), \\ \vdots \qquad\qquad \\ \mathtt{wk}(\llparenthesis ws \rrparenthesis, \llparenthesis s \rrparenthesis, \llparenthesis r \rrparenthesis, \llparenthesis ak_k \rrparenthesis, n, \llparenthesis ses \rrparenthesis), \\ \mathtt{inknw}(\llparenthesis r \rrparenthesis, \llparenthesis ik_1 \rrparenthesis, 1, \llbracket ses \rrbracket), \\ \vdots \qquad\qquad \\ \mathtt{inknw}(\llparenthesis r \rrparenthesis, \llparenthesis ik_l \rrparenthesis, m, \llbracket ses \rrbracket) \end{array}$$

[10] It does not change during the simulation of the protocol session.

- for each `w(`i, _, _, $[ak_1, \ldots, ak_k]$, $[ik_1, \ldots, ik_l]$, _, _`)` occurring in $\mathcal{S}$ then

  ```
  ds_1 ∪ ··· ∪ ds_l ∪
  {sort(wstep_i), constant(i,wstep_i)} ∪
  {sort(knw_index_1),...,sort(knw_index_k)} ∪
  {constant(1,knw_index_1),...,constant(k,knw_index_k)} ∪
  {sort(knw_el_i_1),...,sort(knw_el_i_k)} ∪
  {super_sort(knw_el_i_1,s_1),...,super_sort(knw_el_i_k,s_k)} ∪
  {
     constant(wk(wstep_i,mr,mr,knw_el_i_1,knw_index_1,session),fluent),
                                         ⋮
     constant(wk(wstep_i,mr,mr,knw_el_i_k,knw_index_k,session),fluent)
  } ∈ D_S
  ```

  where the knowledge element sort s_j and the appropriate SATE language declarations ds_j ($j = 1, \ldots, k$) are computed by the predicate `preprocess_msg(`ak_j`,knw_el_`i`_`j`,`s_j`,`ds_j`)`;

- for each

  ```
  # lb=_, type=Init t_1 ... t_n
  ```

  occurring in $\mathcal{S}$ and for each $k = 1, \ldots, n$ such that $t_k =$ `w(_,_,mr(`rec`),_,[],_,`ses`)` then

  ```
  {sort(session_base_ses),constant(ses,session_base_ses) } ∪
  {sort(session_ses_rep_0),...,sort(session_ses_rep_sr) } ∪
  {super_sort(session_ses_rep_0,session_base_ses),sort(session_ses) } ∪
  {
     constant(s(session_ses_rep_0),session_ses_rep_1)
     super_sort(session_ses,session_ses_rep_1)
                                                   ⋮
     constant(s(session_ses_rep_(sr − 1)),session_ses_rep_sr)
     super_sort(session_ses,session_ses_rep_sr)
  }
  {
     super_sort(session,session_ses),
     super_sort(session_ses,session_base_ses),
     super_sort(session_base,session_base_ses)
  } ∪
  {sort(user_rec),constant(rec,user_rec)} ∪
  {sort(mr_rec),constant(mr(user_rec,mr_rec)} ∪
  {sort(inknw_index_1),constant(1,inknw_index_1)} ∪
  {sort(inknw_rec_ses_el_1),super_sort(inknw_rec_ses_el_1,knw_el_empty)} ∪
  {
     constant(inknw(user_rec,inknw_rec_ses_el_1,inknw_index_1,session_base_ses),fluent)
  } ∈ D_S
  ```

  where sr is the value of the session repetitions parameter (see 6.4) which declares how many times a session can be repeated during the simulation (e.g. `set(session_repetitions,1)` states that each protocol session can be iterated only once);

- for each

  ```
  # lb=_, type=Init t_1 ... t_n
  ```

  occurring in $\mathcal{S}$ and for each $k = 1, \ldots, n$ such that $t_k =$ `w(_,_,mr(`rec`),_,`$[ik_1, \ldots, ik_l]$`,_,`ses`)` with $l \geq 1$ then

```
{sort(session_base_ses),constant(ses,session_base_ses) } ∪
{sort(session_ses_rep_0),...,sort(session_ses_rep_sr) } ∪
{super_sort(session_ses_rep_0,session_base_ses),sort(session_ses) } ∪
{
   constant(s(session_ses_rep_0),session_ses_rep_1)
   super_sort(session_ses,session_ses_rep_1)
                                       ⋮
   constant(s(session_ses_rep_(sr − 1)),session_ses_rep_sr)
   super_sort(session_ses,session_ses_rep_sr)
}
{
   super_sort(session,session_ses),
   super_sort(session_ses,session_base_ses),
   super_sort(session_base,session_base_ses)
} ∪
{sort(user_⟦rec⟧),constant(⟦rec⟧,user_⟦rec⟧)} ∪
{sort(mr_rec),constant(mr(user_rec,mr_rec)} ∪
{sort(inknw_index_1),...,sort(inknw_index_l)} ∪
{constant(1,inknw_index_1),...,constant(l,inknw_index_l) ∪
ds_1 ∪ ··· ∪ ds_l ∪
{
   constant(inknw(user_⟦rec⟧,sort_1,inknw_index_1,session_base_ses),fluent),
                                       ⋮
   constant(inknw(user_⟦rec⟧,sort_l,inknw_index_l,session_base_ses),fluent),
} ∈ D_S
```

where sr is the value of session repetitions parameter (see above), ds_i and $sort_i$ $(i = 1, \ldots, l)$ are computed by the predicate call `preprocess_msg(`ik_i`,inknw_`rec`_`ses`_el_`i`,`$sort_i$`,`ds_i`)`

Note that the extension of `constant(w(wstep,mr,mr,knw,knw,session),fluent)` generates a number of ground instances of fluents equal to $|\texttt{wstep}| * |\texttt{mr}|^2 * |\texttt{knw}|^2 * |\texttt{session}|$ where $|\texttt{knw}| = |\texttt{knw\_el}|^{|\texttt{knw\_el}|}$. In fact let us suppose that `{x,y,z}` is the set of knowledge elements then $||\texttt{knw}|| = \{$`[]`, `[x]`, `[y]`, `[z]`, `[x,x]`, `[x,y]`, `[y,x]`,..., `[z,z,z]`$\}$ and $|\texttt{knw}| = 3^3 = 27$. By using the knowledge splitting optimization then we extend both the constant declarations[11]

```
constant(wk(wstep,mr,mr,knw_el,knw_index,session),fluent)
constant(inknw(mr,inknw_el,inknw_index,session),fluent)
```

and since $|\texttt{knw\_index}| \leq |\texttt{knw\_el}|$ and $|\texttt{inknw\_index}| \leq |\texttt{inknw\_el}|$, the number of generated ground instances of fluents is equal to or less than $|\texttt{wstep}| * |\texttt{mr}|^2 * |\texttt{knw\_el}|^2 * |\texttt{session}| + |\texttt{mr}| * |\texttt{inknw\_el}|^2 * |\texttt{session}|$. Exploiting the fact that $|\texttt{inknw\_el}| \leq |\texttt{knw\_el}|$ and making some simplifications, the advantage of using the knowledge splitting is given by the fact that $|\texttt{wstep}| * |\texttt{mr}| * |\texttt{knw\_el}|^2 + |\texttt{inknw\_el}|^2 \ll |\texttt{wstep}| * |\texttt{mr}| * (|\texttt{knw\_el}|^{|\texttt{knw\_el}|})^2$.

6.6 Exploiting Static Fluents

The previous optimization enables a new one. Since the initial knowledge of the honest principal does not change as the protocol execution makes progress, fluents of the form `inknw(`r,ik,ind,ses`)` occurring in the initial state are preserved in all the reachable states and those not occurring in the initial state will not be introduced. In the corresponding planning problem, this means that all

[11]Note that we are considering the worst case. In practice we should extend the declarations `constant(wk(wstep_`i`,mr,mr,knw_el_`i`_`j`,knw_index_`j`,session),fluent)` and `constant(inknw(user_`rec`,`$sort_k$`,inknw_index_`k`,session_base_`ses`),fluent)`. Obviously the language defined by these declarations is contained or equal to the language defined by the declaration we are considering in the above analyses.

the atoms i :`inknw`(r,ik,ind,ses) can be replaced by `inknw`(r,ik,ind,ses) for $i = 0, \dots, n$ thereby reducing the number of propositional letters in the encoding. Moreover, when the initial state is unique, this transformation enables a static simplification of the propositional formula. To specify that the initial knowledge fluents are statics it is sufficient that `static(inknw(_,_,_,_))` $\in D_{\mathcal{S}}$.

6.7 Merging Intruder and Protocol Rules

A further important optimization is based on the idea of merging intruder rules into protocol rules. By merging with a divert intruder rule

```
# lb=Divert_s, type=Intruder_Rules
h(s(xTime)).m(_,mr(s),mr(s),mr(r),msg,_) => h(xTime).i(mr(s)).i(mr(r)).i(msg)
```

a generic protocol rule

`# lb=`*label*`, type=Protocol_Rules` L `=>` R,

such that `m(_,_,`os`,`r`,`msg`,_)` occurs in R, is transformed into

`# lb=`*label*`, type=Protocol_Rules` L `=>` R`.i(`os`).i(`r`).i(`msg`)`

By systematically repeating this step for all protocol rules and then removing divert and memorize intruder rules (see Deliverable D2.2) we get an optimized set of rules. Intuitively speaking, this transformation amounts to incorporating the intruder's activity of memorizing the content of the messages into the protocol rules. The protocol dependent declarations change on the base of the following:

- for each rewrite rule

  `# lb=`*label*`, type=Protocol_Rules` $lt_1 \dots lt_n$ `=>` $rt_1 \dots rt_m$

  occurring in $\mathcal{S}$ such that such that `m(_,_,`os`,`r`,`msg`,_)` $\in \{rt_1 \dots rt_m\}$, then

  ```
  action(label([[x_1]],...,[[x_k]]),true,
         [[[lt_1]],...,[[lt_n]]],
         [at_1,...,at_{m-j},i(os),i(r),i(msg)],
         [dt_1,...,dt_{n-j}]
  ) ∈ D_S
  ```

  and `constant(`*label*`(`$s_1, \dots, s_k$`),action)` $\in D_{\mathcal{S}}$, where:

  - $\{x_1, \dots, x_k\} = VAR(lt_1) \cup \dots \cup VAR(lt_n) \cup VAR(rt_1) \cup \dots \cup VAR(rt_m)$,
  - $s_i = SORT(x_i, \mathcal{S})$ for each $i = 1, \dots, k$,
  - $\{at_1, \dots, at_{m-j}\} \equiv \{[\![rt_1]\!], \dots, [\![rt_m]\!]\} \setminus \{[\![lt_1]\!], \dots, [\![lt_n]\!]\}$, and
  - $\{dt_1, \dots, dt_{n-j}\} \equiv \{[\![lt_1]\!], \dots, [\![lt_n]\!]\} \setminus \{[\![rt_1]\!], \dots, [\![rt_m]\!]\}$;

- for each rewrite rule

  `# lb=`*label*`, type=`*cat* $lt_1 \dots lt_n$ `=>` $rt_1 \dots rt_m$

  occurring in $\mathcal{S}$ such that *cat* $\in${`Intruder_Rules`, `Decomposition_Rules`} and *label* does not begin with the sequence of chars `Divert` or `Memorize`, then

  ```
  action(label([[x_1]],...,[[x_k]]),true,
         [[[lt_1]],...,[[lt_n]]],
         [at_1,...,at_{m-j}],
         [dt_1,...,dt_{n-j}]
  ) ∈ D_S
  ```

  and `constant(`*label*`(`$s_1, \dots, s_k$`),action)` $\in D_{\mathcal{S}}$, where:

- $\{x_1, \ldots, x_k\} = VAR(lt_1) \cup \cdots \cup VAR(lt_n) \cup VAR(rt_1) \cup \cdots \cup VAR(rt_m)$,
- $s_i = SORT(x_i, \mathcal{S})$ for each $i = 1, \ldots, k$,
- $\{at_1, \ldots, at_{m-j}\} \equiv \{\llbracket rt_1 \rrbracket, \ldots, \llbracket rt_m \rrbracket\} \setminus \{\llbracket lt_1 \rrbracket, \ldots, \llbracket lt_n \rrbracket\}$, and
- $\{dt_1, \ldots, dt_{n-j}\} \equiv \{\llbracket lt_1 \rrbracket, \ldots, \llbracket lt_n \rrbracket\} \setminus \{\llbracket rt_1 \rrbracket, \ldots, \llbracket rt_m \rrbracket\}$;

6.8 Time Elimination

The IF term `h(`*time*`)` is a counter which is increased when a message is sent. This counter is currently used only to generate fresh constants during the protocol simulation. In fact each fresh constant term is a pair `c(`*id*`,`*time*`)` where *id* is the identifier of the fresh constant and *time* is the value of the above counter indicating in which "instant" the fresh constant has been generated. Note that the dependency of the fresh constants from *time* affects in a relevant way the cardinality of the fresh constant sort (`fresh_const`) and therefore the number of fluents and, indirectly, the number of actions. Moreover `h(`*time*`)` influences also directly the number of actions, because it occurs in most of the rewrite rules. Finally, the number of fresh constants sufficient to find an attack is usually less than the number of messages present in the attack trace.[12] As consequence, a lot of fresh constants allowed by the SATE specification of the protocol are not used.

The idea of this optimization is to avoid considering the `h(`*time*`)` terms for reducing, considerably, the size of the language of the SATE specification. To do this we have, of course, to introduce an other way for generating and handling fresh constants. The solution is the following. Suppose the action *act* generates the fresh constant fc, then we introduce a new fluent `fresh(`fc`)` stating if fc has not been used (generated) yet[13] and we add that fluent to both the action preconditions list and the action delete list.

For example, in the NSPK protocol, the rule `Step_0_1` (see Figure 8) generates the fresh constant `c(na,`*counter*`)`, then the fluent `fresh(c(na,`*counter*`))` is true in the initial state and

```
action(step_0_1(Xc,NEWXc),
        true,
        [fresh(c(na,counter)),
         inknw(mr(a),mr(a),1,NEWXc),
         inknw(mr(a),mr(b),2,NEWXc),
         inknw(mr(a),pk(ka),3,NEWXc),
         inknw(mr(a),primed(pk(ka)),4,NEWXc),
         inknw(mr(a),pk(kb),5,NEWXc),
         wk(0,mr(intruder),mr(a),etc,1,Xc)],
        [m(1,mr(a),mr(a),mr(b),crypt(pk(kb),c(nonce(c(na,counter)),mr(a))),Xc),
         wk(2,mr(b),mr(a),nonce(c(na,counter)),1,Xc),
         i(mr(a)),
         i(mr(b)),
         i(crypt(pk(kb),c(nonce(c(na,counter)),mr(a))))],
        [fresh(c(na,counter)),
         wk(0,mr(intruder),mr(a),etc,1,Xc)]).
```

is the SATE representation of the IF rule `Step_0_1`.[14] As you can see, if the action is applied then `c(na,`*counter*`)` is generated and `fresh(c(na,`*counter*`))` becomes false. Therefore the action will not be able to be applied more, because in the preconditions it checks if `fresh(c(na,`*counter*`))` is true.

By using the above solution the freshness is guaranteed, but we have, of course, to specify what is the second argument in the pair representing a fresh constant. In fact, we do not use the `h(`*time*`)` terms anymore and therefore the fresh constants cannot be dependent from *time*

[12] Note that this number corresponds to the maximum value of the counter

[13] This fluent will be true in the initial state.

[14] The above action is obtained by applying also the optimizations: boolean elimination from principal terms, knowledge splitting, and merging intruder and protocol rules.

anymore. The idea is to use the session terms as second argument in the pair representing a fresh constant. This means to have one fresh constant for each possible pair $\mathtt{c}(id,ses)$ where id is a fresh constant identifier and $ses \in \|\mathtt{session}\|$. Note that, although we decrease ($|\mathtt{session}| < |\mathtt{time}|$) the number of fresh constants available, we preserve the completeness of the approach. In practice we can do better, because in our experience (i) one single fresh constant for each fresh constant identifier or (ii) a fresh constant for each pair composed by a fresh constant identifier and by a session repetition identifier are sufficient to find the attack. The role of the boolean parameter `multi_fresh_const` (see Section 5.2) is to switch between (i) and (ii).

6.9 Enforcing Monotonicity of Fluents

A critical issue in the propositional encoding technique described in Section 4.1 is the quadratic growth of the number of Conflict Exclusion Axioms in the number of actions. This fact often confines the applicability of the methods to planning problems with a small number of actions. A way to lessen this problem is to reduce the number of conflicting axioms by applying the following two ideas. First, we consider the intruder knowledge as *monotonic*. Let s be a fact, then we say that s is monotonic iff for all S if $s \in S$ and $S \rightarrow S'$, then $s \in S'$. The idea here is to transform the rules so to make the facts of the form $i(\ldots)$ monotonic. The transformation on the rules is very simple as it amount to adding the monotonic facts occurring in the left hand side of the rule to its right hand side. Second, we introduce the SATE construct `is_not_conflict_fluent(P)` for relaxing the Conflict Exclusion Axioms relative to the fluents `P`. In particular the declarations `is_not_conflict_fluent(wk(_,_,_,_,_,_))` and `is_not_conflict_fluent(m(_,_,_,_))` are added to the SATE specification. The nice effect of these transformations is that the number of Conflict Exclusion Axioms generated by the associated planning problems drops dramatically. For instance, application of this optimization to the analysis of the NSPK protocol and of the Encrypted Key Exchange (EKE) protocols reduces the number of Conflict Exclusion Axioms from 740 to 20 and from 6,678 to 324, respectively.

6.10 Session and Real-Sender Elimination from Message Terms

By analyzing the IF message terms we can see that the real sender and the session fields are redundant. The first is useful to be able to state who, really, has sent the message, but it is neglected by the receiver of the message. Since is possible to understand who, really, has sent the message by analyzing the rule applied to send the message, then we can eliminate the real sender field. The session field is useful only to be able to state in which session the message has been sent. Again it is possible to understand in which session the message has been sent by analyzing the rule applied to send the message, then we can eliminate also the session field. Hence the IF message term is translated on the base of the following:

- for each message term then

$$[\![\mathtt{m}(ms, \_, os, r, msg, \_)]\!] = \mathtt{m}([\![ms]\!], [\![os]\!], [\![r]\!], [\![msg]\!])$$

In conclusion, by applying the above optimization the number of ground fluents is reduced by many orders of magnitude. In the case of the NSPK protocol, the original IF to SATE translation generates a planning problem with millions of fluents and actions, while by using the presented optimizations the number of fluents and actions become 83 and 56 respectively.

7 Implementation

In this section we present a run of our tool against the usual example of the NSPK protocol. Given the IF specification (typed version) of the NSPK protocol and fixed the input parameters by means of the following:

```
SATE file generated in 0.27 sec

STATISTICS (AXIOMS GENERATION)  RUNTIME(sec)
Initial Facts:                  0.56
Goals:                          0.04
Ape Axioms:                     0.06
Explanatory Frame Axioms:       0.07
Conflict Exclusion Axioms:      0.0
                                ------
Total:                          0.73

STATISTICS (AXIOMS TRANSLATION) CLAUSES RUNTIME(sec)
Initial Facts:                  83      0.03
Goals:                          4       0.01
Ape Axioms:                     3722    0.6
Explanatory Frame Axioms:       1660    0.43
Conflict Exclusion Axioms:      20      0.01
                                        ------
Total:                                  1.08

Number of Atoms:        1529
Number of Clauses:      5489

Plan found in 0 sec:

Step 0:  [2.1. a -> I : {na,a}ki]
Step 1:  [decrypt_public_key_1(pk(ki),c(nonce(c(na,fresh)),mr(a)))]
Step 2:  [decompose_1(nonce(c(na,fresh)),mr(a))]
Step 3:  [1.1. I(a) -> b :{na,a}kb]
Step 4:  [1.2. b -> a : {na,nb}ka]
Step 5:  [2.2. I -> a : {na,nb}ka]
Step 6:  [2.3. a -> I : {nb}ki]
Step 7:  [decrypt_public_key_3(pk(ki),nonce(c(nb,fresh)))]
Step 8:  [1.3. I(a) -> b :{nb}kb]
Step 9:  [step_3_1_1_1(c(nb,fresh),c(na,fresh))]
```

Figure 9: Output generated by SATE

```
set(steps,10).
set(session_repetition,1).
set(multi_fresh_term,off).
set(term_depth_bound,8).
```

the SAT-based model checker returns the output displayed in Figure 9. The first row reports the time required by the IF2SATE module to translate the IF specification into SATE. The output then contains the statistics relative to the Axioms Schema Generation and to the translation of them into a propositional formula expressed in the DIMACS format. Then, the numbers of atoms and of clauses charactering the propositional formula are presented. Finally, if an attack is found, then it is reported the time spent by the selected state-of-the-art SAT solver to solve the corresponding SAT instance and the trace of the attack by means of a sequence of steps and for each step a list of the executed actions. In particular, if the executed action states that an agent A has sent a message MSG to the agent B in the protocol step i and in the session s then `s.i.` A `->` B `:` MSG is reported in place of the action name.

8 Experimental Results

We have implemented the above ideas in SATE (SAT Encoder), a SAT-based model-checker for security protocol analysis. Given a security problem Ξ and a bound on the length of the attack n, SATE applies the optimizing transformations previously described to Ξ and obtains a new security problem Ξ'. The security problem Ξ' is translated into a corresponding planning problem $\Pi_{\Xi'}$ which is turn compiled into SAT using the previously methodology outlined. The propositional formula is then fed to a state-of-the-art SAT solver (currently Chaff [12], SIM [6], and SATO [13] are supported) and any model found by the solver is translated back into an attack which is reported to the user.

Table 1: Experimental Results

Protocol Name	AttK	AttL	Fluents	Actions	Wff	OptT	EncT	Solve
Encrypted Key Exchange (EKE)	Parallel session	9	188	145	2,062	0.6	3.1	0.1
ISO Symmetric Key One-Pass Unilateral Authentication	Replay	5	101	175	1,453	0.1	0.9	0.1
Kao-Chow Repeated Authentication	Replay	*	43,472	54,388	$\sim 1,3 \times 10^6$	40.3	3,311.3	*
Needham-Schroeder with Conventional Keys	Replay	11	4,112	8,498	85,509	6.7	174.4	1.1
Needham-Schroeder Public Key (NSPK)	Man-in-the-middle	10	83	56	680	0.3	0.7	0.1

We have run our tool against a selection of problems drawn from [4]. The results of our experiments are reported in Table 1. For each protocol we give the kind (AttK) and length (AttL) of the attack found, the number of fluents (Fluents) and actions (Actions) of the corresponding planning problem, the number of formulae (Wff) in the propositional encoding, the time spent by the optimizing transformations (OptT) and that spent to generate the propositional encoding (EncT) as well as the time spent by Chaff to solve the corresponding SAT instance (Solve).[15]

The experiments show that both the application of the optimizing transformations and the SAT solving activity are carried out very quickly and that the overall time is dominated by the SAT encoding. It is worth pointing out that to perform the propositional encoding we use (an optimized version of) the encoding technique presented in Section 5. Other, more sophisticated, techniques and tools are available (e.g. BlackBox [11]) and can be used to this end. As a final remark, it must be noted that small changes in the protocol require only the re-computation of a small part of the SAT encoding. Therefore the encoding time may have a smaller impact in practical situations than the figures in Table 1 seem to indicate.

As we remarked above, a detailed comparison of the relative performance of the three tools of the AVISS project will be subject of future deliverables (see [3, 7, 8]). A preliminary comparison on a selection of examples is given in [1].

9 Conclusion and Future Work

We have implemented a tool that reduces security problems to propositional logic. Application of the tool to a set of well-known authentication protocols shows that the propositional formulae generated by the tools are solved in a fraction of a second by state-of-the-art SAT-solvers and hence attacks to the protocols are found very quickly by our tool.

The only problem with the approach is the size of and—consequently—the time spent generating the propositional encoding. There are potentially many ways to tackle this problem:

[15]Times have been obtained on a PC with a 1.4 GHz Processor and 512 MB of RAM. A "*" indicates a memory overflow.

- *Message Splitting.* All the messages exchanged during the runs of the protocol are instance of a small set of patterns. Let $t[x_1, \ldots, x_n]$ be such a pattern. Then all its instances are of the form $t[s_1, \ldots, s_n]$ where $s_i \in S_i$ and S_i is a set of (ground) terms for $i = 1, \ldots, n$. Thus the number of ground instances of $t[x_1, \ldots, x_n]$ is $\Pi_{i=1}^{n}|S_i|$. By finding a suitable key (in the data base sense) for such messages it is possible to split the $t[x_1, \ldots, x_n]$ to $t_1[x_1]$,...,$t_n[x_n]$ whose cardinality is $\Sigma_{i=1}^{n}|S_i|$. This optimization may lead to a further dramatic improvement to the size of the encoding.

- *Widening the Use of Iterative Deepening.* Currently our tool can be used to perform iterative deepening on the number of primitive operations, maximal runs of protocols sessions, and maximal number of different nonces. There are other parameters we can consider to this end. For instance we are currently working at identifying some suitable notion of size of the complexity of the messages exchanged by the principals to be used to this end.

- *Properties of Cryptographic Operations as Invariants.* Currently the properties of the cryptographic operations are modeled as primitive operations in the IF. This increases dramatically the number of actions in the corresponding planning problem and hence the size of the final encoding. A more natural way to look at the properties of cryptographic operations is as invariants of the input security problem. However the incorporation of invariants in our encoding mechanism is not straightforward and more work (both theoretical and implementational) is needed to exploit this very promising transformation.

We plan to focus on these questions as part of future work in the remaining months of the AVISS project, as well as in the follow-up RTD project with industry involvement that we are currently planning.

Bibliography

[1] A. Armando, D. Basin, M. Bouallagui, Y. Chevalier, L. Compagna, S. Mödersheim, M. Rusinowitch, M. Turuani, L. Viganò, and L. Vigneron. The AVISS Security Protocol Analysis Tool. January 2002.

[2] A. Armando and L. Compagna. Automatic SAT-Compilation of Security Problems. January 2002.

[3] Y. Chevalier and L. Vigneron. Automated Unbounded Verification of Security Protocols. Technical Report 4369, LORIA, Vandoeuvre les Nancy, February 2002.

[4] J. Clark and J. Jacob. A Survey of Authentication Protocol Literature: Version 1.0, 17. Nov. 1997. URL: `http://www.cs.york.ac.uk/~jac/papers/drareview.ps.gz`.

[5] M. D. Ernst, T. D. Millstein, and D. S. Weld. Automatic SAT-compilation of Planning Problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 1169–1177. Morgan Kaufmann Publishers, 1997.

[6] 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.

[7] Freiburg Group. Deliverable 3.4 : Final definition, implementation and experimentation with the on-the-fly model-checker. 2002.

[8] Genoa Group. Deliverable 3.6 : Final definition, implementation and experimentation with the SAT model-checker. 2002.

[9] Protheo Loria Group. Deliverable 3.5 : Final definition, implementation and experimentation with the constraint-logic model-checker. 2002.

[10] H. Kautz, H. McAllester, and B. Selman. Encoding Plans in Propositional Logic. In L. C. Aiello, j. Doyle, and S. Shapiro, editors, *KR'96: Principles of Knowledge Representation and Reasoning*, pages 374–384. Kaufmann, M., 1996.

[11] Henry A. Kautz and Bart Selman. Unifying SAT-based and graph-based planning. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI'99)*, pages 318–325. Morgan Kaufmann, 1999.

[12] 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.

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