AVISS — Automated Verification of Infinite State Systems

(IST-2000-26410)

Deliverable D3.3

Preliminary definition, implementation and experimentation with the SAT model checker

1 Introduction

The objective of the third work-package WP3 of our project AVISS IST-2000-26410 is to experiment the chosen automated deduction techniques implemented in the prototype verification tool against the verification problems of the corpus [1] of security protocols.

The first step (*encoding*) is the definition of encodings that translates the protocol verification problems, obtained by applying the translator of WP2 to the corpus, into deduction problems falling into the scope of application of the chosen automated deduction techniques. (The architecture of the prototype is shown again in Figure 1.) These encodings, together with the translation mechanism set up in WP2, allows us to run the available automated deduction engines against the verification problems of the corpus (*experiments*). The experiments indicate ways to improve the encodings as well as the inference strategies implemented in the available automated deduction engines (*tuning*).

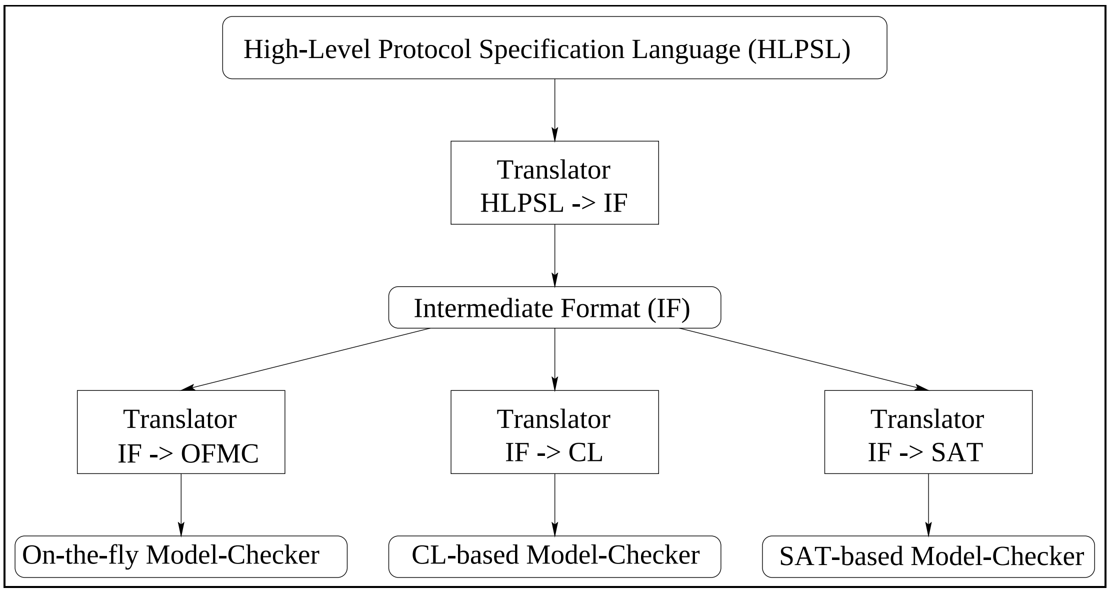

Figure 1: Architecture of the prototype verification tool

This deliverable D3.3 describes our implementation of task T3.3, i.e. define the encoding from the IF to the input format for the model-checker based on propositional satisfiability checking, develop a prototype translator implementing the encoding, and experiment with problems from the corpus, tuning the encoding and/or the model-checker based on propositional satisfiability checking.

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 shows the architecture of the SAT-based Model-Checker. As in the other approaches (see Deliverables D3.1 and D3.2) the HLPSL2IF compiler translates the HLPSL specification into the Intermediate Format (IF — see Deliverable D2.2). The parser of the IF, IF2PIF, reads a IF specification and translates it into an equivalent specification in the Prolog Intermediate Format (PIF). The PIF preserves the rewrite rules representation of the IF but it is amenable to further transformations. The PIF2SATE compiler translates the PIF specification into a STRIPS problem expressed in the SATE specification language which is then fed to SATE. SATE takes as input the STRIPS problem and two parameters specifying the bound in the number of operation applications and the bound on the depth of the terms during expansion, respectively, 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 [7] and SATO [9] are interfaced to SATE) 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.

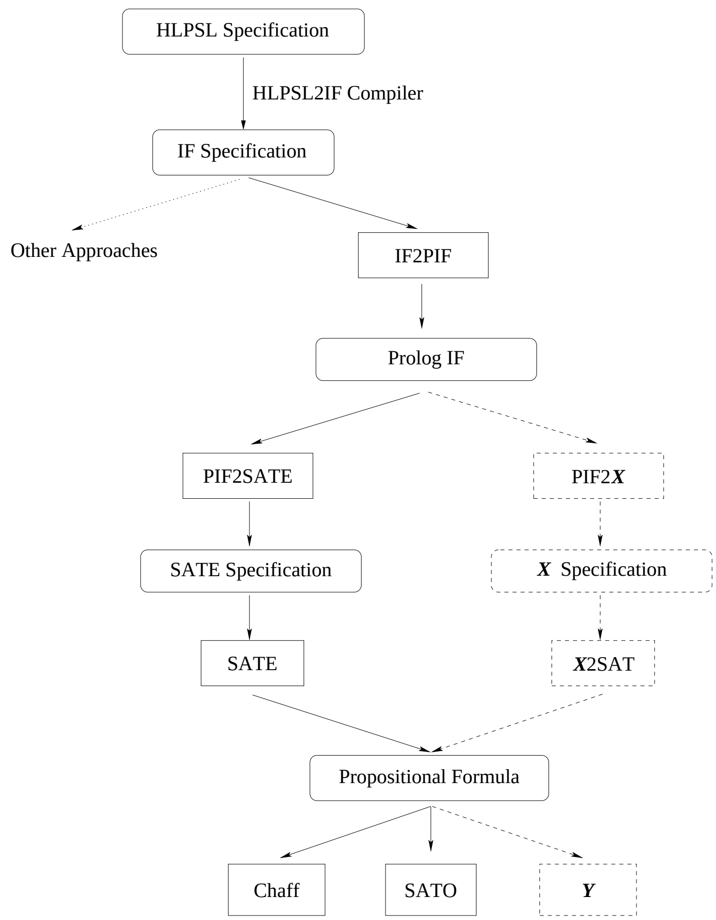

Figure 2: Architecture of the SAT-based Model-Checker

This report is organized in the following way. We start by presenting the IF parser and the difference between the IF and the PIF (section 3). Then we introduce the SATE specification language (section 4) and we discuss the translation from the PIF to the SATE specification language (section 5). We conclude by describing SATE (section 6).

3 The IF Parser

The parser of the IF, IF2PIF, parses specifications in the IF and returns equivalent specifications in the Prolog Intermediate Format (PIF). The parser is written using definite clause grammars (see [2, 8]).[1]

[1] *Definite clause grammars* extend the well-known context-free grammars. They admit both a declarative and a procedural interpretation and are therefore an ideal means to build parsers in a flexible way.

The parser implements two phases. In the first it parses the options lines and in the second it parses the rewrite rules. At the end the PIF file contains the following prolog fact:

`problem(`$OPTs$`,`RRs`).`

where $OPTs$ is the prolog list containing the parsed options[2] and RRs is the prolog list containing the parsed rewrite rules. In order to define the content of the above lists we present the translation from IF to PIF. If t is an IF syntactic construct, then $(\![t]\!)$ is the corresponding syntactic construct of the PIF.

- Let id be an IF identifier such that $id \neq \mathtt{I}$. If 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 $id = \mathtt{I}$ (and hence represents the IF intruder constant) then $(\![id]\!) = \mathtt{intruder}$, otherwise $(\![id]\!) = id$. For example, $(\![\mathtt{Se01}]\!) = \mathtt{se01}$, $(\![\mathtt{I}]\!) = \mathtt{intruder}$, $(\![\mathtt{xA}]\!) = \mathtt{xA}$, and $(\![\mathtt{etc}]\!) = \mathtt{etc}$.

- Let end be the IF symbol representing the end of the IF knowledge list and $t_1, \ldots, t_n$ be IF terms, then $(\![\mathtt{c}(t_1,\mathtt{c}(t_2,\ldots\mathtt{c}(t_n, end)\ldots))]\!) = [(\![t_1]\!), (\![t_2]\!), \ldots, (\![t_n]\!)]$. For example, $(\![\mathtt{c(a,c(b,c(ka,etc)))}]\!) = \mathtt{[a,b,ka]}$.

- Let t be an IF term, then $(\![\mathtt{pk}(t)\mathtt{'}]\!) = \mathtt{primed(pk(}(\![t]\!)\mathtt{))}$.

- Let f be an IF function symbol different from $\mathtt{c}$, then $(\![f(t_1, \ldots, t_n)]\!) = f((\![t_1]\!), \ldots, (\![t_n]\!))$.

- Let $t_1, \ldots, t_n$ be IF terms, then $(\![t_1. \cdots .t_n]\!) = [(\![t_1]\!), \ldots, (\![t_n]\!)]$.

- Let $label$ and $category$ be two identifiers, l, r be two dotted terms, and `eol` be the end-of-line character, then $(\![\mathtt{\#\ lb=}label\mathtt{,type=}category\ \mathtt{eol}\ l\mathtt{=>}r]\!) = \mathtt{lrr}((\![label]\!), (\![category]\!), (\![l]\!), (\![r]\!))$, and $\mathtt{lrr}((\![label]\!), (\![category]\!), (\![l]\!), (\![r]\!))$ is member of the list RRs.

- Let $label$ and $category$ be two identifiers and l a dotted term, then $(\![\mathtt{\#\ lb=}label\mathtt{,\ type=}category\ \mathtt{eol}\ l]\!) = \mathtt{lrr}((\![label]\!), (\![category]\!), (\![l]\!), \mathtt{[]})$, and $\mathtt{lrr}((\![label]\!), (\![category]\!), (\![l]\!), \mathtt{[]})$ is member of the list RRs.

- Let op be an IF option identifier, then $(\![\mathtt{\#\ option=}op\ \mathtt{eol}]\!) = (\![op]\!)$ and $(\![op]\!)$ is member of the list $OPTs$.

4 STRIPS Problems and the SATE Specification Language

The SATE specification language is inspired by STRIPS [5]. A STRIPS *problem* is a tuple $\langle \mathcal{F}, \mathcal{A}, Ops, I, G \rangle$, 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

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

where $Action \in \mathcal{A}$ and Pre, Add, and Del are finite sets of fluents; I is a finite set of fluents and G is a boolean combination of fluents representing the initial and the final states respectively.

The SATE *specification* language allows us to specify SATE problems and 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).
```

[2] At now $OPTs = [\mathtt{typed}]$ or $OPTs = [\mathtt{untyped}]$.

sate_spec	::=	(*assertion* .)*
assertion	::=	*sort_decl* | *super_sort_decl* | *constant_decl* | *invariant* | *initial_state* | *goal* | *operator*
sort_decl	::=	`sort`(*sort_id*)
super_sort_decl	::=	`super_sort`(*sort_id*,*sort_id*)
constant_decl	::=	`constant`(*arity*,*sort_id*)
invariant	::=	`invariant`(*composite_fluent*)
initial_state	::=	`facts`(*init*)
goal	::=	`goal`(*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**)
init, goal, pre, add, del	::=	[*fluent**]
composite_fluent	::=	*term* = *term* | *fluent* | ~ *fluent* | *fluent* <=> *fluent*
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

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.

- **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.

- **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([m(1,mr(a),mr(a),mr(b),crypt(pk(kb),c(nonce(na,s(s(0))),mr(a))),1)]).
```

states that all states containing the fluent

```
m(1,mr(a),mr(a),mr(b),crypt(pk(kb),c(nonce(na,s(s(0))),mr(a))),1)
```

are goal states.

- **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] Currently we only allow a list of fluents as argument to `goal`, whereas in principle it is possible to have an arbitrary boolean combination of fluents. This restriction will be lifted in the next deliverable.

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.

5 Translating PIF into the SATE Specification Language

We now define the translation from PIF to the SATE specification language. First we present a simple translation and then we show some optimizations.

Our goal is to convert a PIF 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 and operators) 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 4 shows the protocol independent super-sort and constant declarations.[4]

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

- if t is a PIF syntactic construct, then $[\![t]\!]$ is the corresponding syntactic construct of SATE;
- if t is a PIF term, then $VAR(t)$ is the function returning the set of PIF variables occurring in t; for example, VAR(`h(xTime)`) = {`xTime`};
- if x is a PIF 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 $SORT$(`xTime`, $\mathcal{S}$) = `time`;
- for each PIF constant symbol s (hence the first character of s is not the `x` character) if
  - `mr(`s`)` occurs in $\mathcal{S}$, then `constant(`s`,user)` $\in D_{\mathcal{S}}$;
  - `pk(`s`)` occurs in $\mathcal{S}$, then `constant(`s`,public_key)` $\in D_{\mathcal{S}}$;
  - `primed(pk(`s`))` occurs in $\mathcal{S}$, then `constant(`s`,private_key)` $\in D_{\mathcal{S}}$;
  - `sk(`s`)` occurs in $\mathcal{S}$, then `constant(`s`,private_key)` $\in D_{\mathcal{S}}$;
  - `fu(`s`)` occurs in $\mathcal{S}$, then `constant(`s`,fu_symbol)` $\in D_{\mathcal{S}}$;
  - `nonce(c(`s`,xTime))` occurs in $\mathcal{S}$, then `constant(`s`,fresh_nonce_id)` $\in D_{\mathcal{S}}$;
  - `pk(c(`s`,xTime))` occurs in $\mathcal{S}$, then `constant(`s`,fresh_public_id)` $\in D_{\mathcal{S}}$;
  - `primed(pk(c(`s`,xTime)))` occurs in $\mathcal{S}$, then `constant(`s`,fresh_private_id)` $\in D_{\mathcal{S}}$;
  - `sk(c(`s`,xTime))` occurs in $\mathcal{S}$, then `constant(`s`,fresh_symmetric_id)` $\in D_{\mathcal{S}}$;
  - `c(`s`,`s`)` occurs in $\mathcal{S}$, then `constant(`s`,intruder_const)` $\in D_{\mathcal{S}}$;

[4] To increase the readability, the sort declarations are not given, but they can be derived 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).


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(secret(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(pk(private_key),pkinv).
constant(primed(pkinv),pk).
constant(sk(symmetric_key),sk).
constant(fu(fu_symbol),fu).
constant(mr(user),mr).
constant(nonce(fresh_nonce),nonce).

constant(pk(fresh_public_key),pk).
constant(pk(fresh_private_key),pkinv).
constant(sk(fresh_symmetric_key),sk).
constant(pk(fresh_intruder_const),pk).
constant(pk(fresh_intruder_const),pkinv).
constant(sk(fresh_intruder_const),sk).
constant(nonce(fresh_intruder_const),nonce).

constant(c(fresh_public_id,time),fresh_public_key).
constant(c(fresh_private_id,time),fresh_private_key).
constant(c(fresh_symmetric_id,time),fresh_symmetric_key).
constant(c(fresh_nonce_id,time),fresh_nonce).
constant(c(intruder_const,intruder_const),fresh_intruder_const).
```

Figure 4: Protocol independent declarations

- for each `lrr`($label$,cat,[$t_1,\ldots,t_n$],[$t_{n+1},\ldots,t_{n+m}$]) occurring in $\mathcal{S}$ such that $cat \in \{$`protocol_Rules`$\}$ and for each $k, j = 1,\ldots,n{+}m$ such that $t_k = \mathtt{m}(mi, \_, \_, \_, \_, \_)$, $t_j = \mathtt{w}(wi, \_, \_, \_, \_, \_, \_)$, then $\{$`constant`(mi,`mstep`),`constant`(wi,`wstep`)$\} \in D_{\mathcal{S}}$;
- for each `lrr(_,init,`[$t_1,\ldots,t_n$]`,[])` occurring in $\mathcal{S}$, then `facts`([$[\![t_1]\!],\ldots,[\![t_n]\!]$]) $\in D_{\mathcal{S}}$;
- for each `lrr`($label$,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

  $$\begin{array}{l}\mathtt{action}(label([\![x_1]\!],\ldots,[\![x_k]\!]),\mathtt{true},\\ \qquad [[\![lt_1]\!],\ldots,[\![lt_n]\!]],\\ \qquad [[\![rt_1]\!],\ldots,[\![rt_m]\!]],\\ \qquad [[\![lt_1]\!],\ldots,[\![lt_n]\!]]\\ ) \in D_{\mathcal{S}}\end{array}$$

  and `constant`($label(s_1,\ldots,s_k)$,`action`) $\in D_{\mathcal{S}}$, where $x_1,\ldots,x_k = VAR(lt_1)\cup\cdots\cup VAR(lt_n)\cup VAR(rt_1)\cup\cdots\cup VAR(rt_m)$ and $s_i = SORT(x_i,\mathcal{S})$ for each $i = 1,\ldots,k$;
- for each `lrr(_,goal,`[$t_1,\ldots,t_n$]`,[])` occurring in $\mathcal{S}$ then `goal`([$[\![t_1]\!],\ldots,[\![t_n]\!]$]) $\in D_{\mathcal{S}}$;
- $[\![\mathtt{w}(ws, s, r, [t_1,\ldots,t_n], [t_{n+1},\ldots,t_{n+m}], b, ses)]\!] =$ $\mathtt{w}([\![ws]\!],[\![s]\!],[\![r]\!],\mathtt{c}([\![t_1]\!],\ldots\mathtt{c}([\![t_n]\!],\mathtt{etc})\ldots),\mathtt{c}([\![t_{n+1}]\!],\ldots\mathtt{c}([\![t_{n+m}]\!],\mathtt{etc})\ldots),[\![b]\!],[\![ses]\!])$;
- $[\![f(t_1,\ldots,t_n)]\!] = f([\![t_1]\!],\ldots,[\![t_n]\!])$;
- if v be a PIF 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, $[\![\mathtt{xA}]\!] = \mathtt{XA}$.

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 6 and [4]). The ground instances of fluents (actions) are given by expansion of the constant declarations relative to the sort `fluent` (`action`). For example the expansion of the declaration

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

generates a number of fluents equal to

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

where $|s|$ returns the number of expanded terms relative to the sort s. In order to decrease the number of ground fluents and actions we apply the optimizations presented in the following subsections.

5.1 Invariants

This is an useful 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 term are equal, we can use:

```
invariant(~(w(_,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)
```

become respectively:

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

5.2 Protocol dependent messages

Note that in Figure 4 the declarations about the messages (`msg`) are too general. In fact the expansion of these declarations generates 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 avoid to express the declarations about the messages as independent from the protocol and we analyze the PIF specification $\mathcal{S}$ to extract the patterns of the possible exchanged messages allowed by the protocol.

In particular, for each `lrr(`*label*`,`*cat*`,_,[`$rt_1, \ldots, rt_m$`])` occurring in $\mathcal{S}$ such that $cat \in \{$`protocol_Rules`$\}$ and for each $k = 1, \ldots, m$ such that $rt_k =$ `m(`i`,_,_,_,`msg`,_)` then $\{$`sort(msg_`i`), sort(mstep_`i`), super_sort(msg_`i`,`$sort$`),` `constant(m(mstep_`i`,mr,mr,mr,msg_`i`,session),fluent)`$\} \cup Cs \in D_{\mathcal{S}}$ where Cs and $sort$ are computed by the procedure PREPROCESS_MSG($msg, i, sort, Cs$). This procedure (partially given in Figure 5) executes a syntactical analysis of the PIF message msg computing $sort$, i.e. the sort of msg (previous a concatenation with i), and Cs, i.e. the list of constant declarations useful to define the SATE sub-language relative to msg.

Let us consider the example of the Needham-Schroeder Public-Key (NSPK) protocol (see [1]). The PIF rewrite rules relative to the NSPK protocol steps are given in Figure 6. For each of these rewrite rule we analyze the right-hand-side (fourth parameter of the term `lrr`) and if it contains a message term of the form `m(`i`,_,_,_,`msg`,_)` then we extract the field msg. Finally we call the procedure PREPROCESS_MSG($msg, i, sort, Cs$) returning the SATE $sort$ of msg and the set of constant declarations relative to msg. In this case we call the procedure three times and the results are the following:

```
PREPROCESS_MSG(crypt(xKb,c(xNa,xA)),1,
               crypt_1,
               [  constant(crypt(pk,pair_1),crypt_1),constant(c(atom,atom),pair_1),
                  super_sort(knw_el,pair_1),super_sort(knw_el,crypt_1)
               ]
)

PREPROCESS_MSG(crypt(xKa,c(xNa,nonce(c(nb,xTime)))),2,
               crypt_2,
               [  constant(crypt(pk,pair_2),crypt_2),constant(c(atom,nonce),pair_2),
                  super_sort(knw_el,pair_2),super_sort(knw_el,crypt_2)
               ]
)

PREPROCESS_MSG(crypt(xKb,xNb),3,
               crypt_3,
               [constant(crypt(pk,atom),crypt_3),super_sort(knw_el,crypt_3)]
)
```

Hence at the end of the above process $D_{\mathcal{S}}$ will contain the following declarations:

```
sort(msg_1).
sort(mstep_1).
```

```
super_sort(msg_1,crypt_1).
constant(m(mstep_1,mr,mr,mr,msg_1,session),fluent).

sort(msg_2).
sort(mstep_2).
super_sort(msg_2,crypt_2).
constant(m(mstep_2,mr,mr,mr,msg_2,session),fluent).

sort(msg_3).
sort(mstep_3).
super_sort(msg_3,crypt_3).
constant(m(mstep_3,mr,mr,mr,msg_3,session),fluent).

constant(crypt(pk,pair_1),crypt_1).
constant(c(atom,atom),pair_1).
constant(crypt(pk,pair_2),crypt_2).
constant(c(atom,nonce),pair_2).
constant(crypt(pk,atom),crypt_3).

super_sort(knw_el,pair_1).
super_sort(knw_el,crypt_1).
super_sort(knw_el,pair_2).
super_sort(knw_el,crypt_2).
super_sort(knw_el,crypt_3).
```

Using the general declaration of Figure 4, the number of ground instances of fluents relative to the message terms was:

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

while using this optimization that number decreases to the value:

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

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 in 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 $|\texttt{msg\_1}| + |\texttt{msg\_2}| + |\texttt{msg\_3}| \simeq 1,000$.

5.3 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[5] property of an agent involved in a protocol session;
3. the ordering of the terms in both kinds of knowledge is relevant.

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

```
procedure PREPROCESS_MSG(msg,lb,sort,Cs)
  inputs: msg, a PIF message
          lb, a label
          sort, the sort relative to the message msg
          Cs, a set of SATE declarations
case
    msg == crypt(_,submsg) :
          PREPROCESS_MSG(submsg,lb,submsgsort,Cs)
          constant(crypt(pk,submsgsort),crypt_lb) ∈ Cs
          super_sort(knw_el,crypt_lb) ∈ Cs
          sort ← crypt_lb
          break
    msg == c(submsg1,submsg2) :
          PREPROCESS_MSG(submsg1,lb,submsgsort1,Cs)
          PREPROCESS_MSG(submsg2,lb,submsgsort2,Cs)
          constant(c(submsgsort1,submsgsort2),pair_lb) ∈ Cs
          super_sort(knw_el,pair_lb) ∈ Cs
          sort ← pair_lb
          break
    msg == pk(submsg) and submsg is an atom :
          sort ← pk
          break
    msg == primed(pk(submsg)) and submsg is an atom :
          sort ← pk
          break
    msg == nonce(c(submsg,_)) and submsg is an atom :
          sort ← nonce
          break
    msg is an atom :
          sort ← atom
          break
esac
```

Figure 5: Pseudo-code of the procedure PREPROCESS_MSG

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:

- for each $\mathtt{w}(\_,\_,\_,[ak_1,\ldots,ak_k],[ik_1,\ldots,ik_l],\_,\_)$ occurring in $\mathcal{S}$ then $\{\mathtt{constant(1,acq\_index)},\ldots,\mathtt{constant(}k\mathtt{,acq\_index)}\} \cup$ $\{\mathtt{constant(1,in\_index)},\ldots,\mathtt{constant(}l\mathtt{,in\_index)}\} \in D_{\mathcal{S}}$;

- for each $\mathtt{lrr(\_,init,[}t_1,\ldots,t_n\mathtt{],[])}$ occurring in $\mathcal{S}$ and for each $k = 1,\ldots,n$ such that $t_k = \mathtt{w}(\_,\_,\_,\_,[ik_1,\ldots,ik_l],\_,\_)$ then $\{\ \mathtt{super\_sort(inknw\_el,}sort_1\mathtt{)},\ldots,\mathtt{super\_sort(inknw\_el,}sort_l\mathtt{)}\ \} \cup Cs_1 \cup \cdots \cup Cs_l \in D_{\mathcal{S}}$ where Cs_i and $sort_i$ $(i = 1,\ldots,l)$ are computed by the procedure calling PREPROCESS_MSG$(ik_i, \mathtt{inknw}, sort_i, Cs_i)$;

- for each principal term such that the initial and acquired knowledge are empty then
$$[\![\mathtt{w}(ws,s,r,\mathtt{[]},\mathtt{[]},ses)]\!] = \begin{array}{l}\mathtt{wk}([\![ws]\!],[\![s]\!],[\![r]\!],\mathtt{etc},1,[\![ses]\!]),\\ \quad\mathtt{inknw}([\![r]\!],\mathtt{etc},1,[\![ses]\!])\end{array}$$

- for each principal term such that the initial knowledge is empty and $k \geq 1$ then
$$[\![\mathtt{w}(ws,s,r,\mathtt{[}ak_1,\ldots,ak_k\mathtt{]},\mathtt{[]},ses)]\!] = \begin{array}{l}\mathtt{wk}([\![ws]\!],[\![s]\!],[\![r]\!],[\![ak_1]\!],1,[\![ses]\!]),\\ \qquad\vdots\\ \mathtt{wk}([\![ws]\!],[\![s]\!],[\![r]\!],[\![ak_k]\!],n,[\![ses]\!]),\\ \quad\mathtt{inknw}([\![r]\!],\mathtt{etc},1,[\![ses]\!])\end{array}$$

- for each principal term such that the acquired knowledge is empty and $l \geq 1$ then

$$\begin{aligned}[\![\mathtt{w}(ws, s, r, \mathtt{[]}, \mathtt{[}ik_1, \ldots, ik_l\mathtt{]}, ses)]\!] = \quad & \mathtt{wk}([\![ws]\!], [\![s]\!], [\![r]\!], \mathtt{etc}, 1, [\![ses]\!]), \\ & \mathtt{inknw}([\![r]\!], [\![ik_1]\!], 1, [\![ses]\!]), \\ & \vdots \\ & \mathtt{inknw}([\![r]\!], [\![ik_l]\!], m, [\![ses]\!])\end{aligned}$$

- for each principal term such that $k \geq 1$ and $l \geq 1$ then

$$\begin{aligned}[\![\mathtt{w}(ws, s, r, \mathtt{[}ak_1, \ldots, ak_k\mathtt{]}, \mathtt{[}ik_1, \ldots, ik_l\mathtt{]}, ses)]\!] = \quad & \mathtt{wk}([\![ws]\!], [\![s]\!], [\![r]\!], [\![ak_1]\!], 1, [\![ses]\!]), \\ & \vdots \\ & \mathtt{wk}([\![ws]\!], [\![s]\!], [\![r]\!], [\![ak_k]\!], n, [\![ses]\!]), \\ & \mathtt{inknw}([\![r]\!], [\![ik_1]\!], 1, [\![ses]\!]), \\ & \vdots \\ & \mathtt{inknw}([\![r]\!], [\![ik_l]\!], m, [\![ses]\!])\end{aligned}$$

Note that the expansion of `constant(w(wstep,mr,mr,knw,knw,bool,session),fluent)` generates a number of ground instances of fluents equal to $|\mathtt{wstep}| * |\mathtt{mr}|^2 * |\mathtt{knw}|^2 * |\mathtt{bool}| * |\mathtt{session}|$ where $|\mathtt{knw}| = |\mathtt{knw\_el}|^{|\mathtt{knw\_el}|}$. In fact let us suppose that `{x,y,z}` is the set of knowledge elements then the sort `knw` is characterized by 3^3 terms showed in the following set:[6] { `[]`, `[x]`, `[y]`, `[z]`, `[x,x]`, `[x,y]`, `[y,x]`,..., `[z,z,z]` }. If we use the knowledge splitting then we expand the declarations `constant(wk(wstep,mr,mr,knw_el,acq_index,bool,session),fluent)` and `constant(inknw(mr,inknw_el,in_index,session),fluent)` and since $|\mathtt{acq\_index}| \leq |\mathtt{knw\_el}|$ and $|\mathtt{in\_index}| \leq |\mathtt{inknw\_el}|$, the number of generated ground instances of fluents is equal to or less than $|\mathtt{wstep}| * |\mathtt{mr}|^2 * |\mathtt{knw\_el}|^2 * |\mathtt{bool}| * |\mathtt{session}| + |\mathtt{mr}| * |\mathtt{inknw\_el}|^2 * |\mathtt{session}|$. Exploiting the fact that $|\mathtt{inknw\_el}| \leq |\mathtt{knw\_el}|$ and making some simplification, the advantage by using the knowledge splitting is given by the fact that $|\mathtt{wstep}| * |\mathtt{mr}| * |\mathtt{knw\_el}|^2 * |\mathtt{bool}| + |\mathtt{inknw\_el}|^2 \ll |\mathtt{wstep}| * |\mathtt{mr}| * (|\mathtt{knw\_el}|^{|\mathtt{knw\_el}|})^2 * |\mathtt{bool}|$.

In conclusion, by applying the above optimization the number of ground fluents is reduced by many orders of magnitude. In Figure 7 we present the protocol independent declarations we have to use together with the above optimizations.

6 SATE

We recall that SATE takes as input a STRIPS problem and two parameters n and δ specifying the bound in the number of operator applications and the bound on the depth of the terms depth during expansion, respectively, and generates a propositional formula whose satisfiability guarantees the reachability of a goal state from an initial state.[7] The encoding of a STRIP problem into a SAT formula can be done in a variety of ways (see [6, 4] for a survey.) In the first of phase of the project we focused on the *linear encodings*, i.e. encodings in which only one action can occur at each time step. More sophisticated encodings which relax this condition (as, e.g., the *parallelized encodings*) will be considered in the future.

The encodings we consider are based on the idea of adding an additional time-index to the actions and fluents, to indicate the state at which the action begins or the fluent holds. Fluents are 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.

We have considered and experimented with two encodings. The first one is based on the *classical frame axioms*, the second on the *explanatory frame axioms*.

[6] 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] The parameters corresponding to n and δ in the implementation are `steps` and `term_depth_bound` respectively.

6.1 Encoding based on the Classical Frame Axioms

Let S be a SATE specification. The encoding of S based on the Classical Frame Axioms is the smallest set of propositional formulae, S^*, such that:

UNIVERSAL AXIOMS Let `facts`(*list_of_fluents*) $\in S$, then $0\!:\!p \in S^*$ for all $p \in$ *list_of_fluents* and $\neg(0\!:\!p) \in S^*$ for all $p \notin$ *list_of_fluents*. Let `goal`(*list_of_fluents*) $\in S$, then $n\!:\!p \in S^*$ for all $p \in$ *list_of_fluents*. Let `action`$(\alpha, c, pre, add, del) \in S$, then

$$(i\!:\!\alpha\sigma \supset \bigwedge\{i\!:\!p\sigma \mid p \in pre\}) \in S^*$$
$$(i\!:\!\alpha\sigma \supset \bigwedge\{(i+1)\!:\!p\sigma \mid p \in add\}) \in S^*$$
$$(i\!:\!\alpha\sigma \supset \bigwedge\{\neg(i+1)\!:\!p\sigma \mid p \in del\}) \in S^*$$

for all substitutions σ such that $\alpha\sigma$ is ground, $c\sigma$ holds, and $0 \leq i \leq n-1$.

CLASSICAL FRAME AXIOMS Let `action`$(\alpha, c, pre, add, del) \in S$, then

$$i\!:\!\alpha\sigma \supset ((i+1)\!:\!f\sigma \leftrightarrow i\!:\!f\sigma) \in S^*$$

for all substitutions σ such that $\alpha\sigma$ is ground, $c\sigma$ holds, $f \notin add \cup del$, and $0 \leq i \leq n-1$.

AT-LEAST-ONE AXIOMS

$$\bigvee\{i\!:\!\alpha\sigma \mid \texttt{action}(\alpha, c, pre, add, del) \in S, \alpha\sigma \text{ is ground and } c\sigma \text{ holds}\} \in S^*$$

for $0 \leq i \leq n-1$.

To instruct SATE to use the encoding based on the explanatory frame axioms the parameter `encoding` must be set to the list `[ape_axiom,classical_frame_axiom,at_least_one_axiom]`.

6.2 Encoding based on the Explanatory Frame Axioms

Let S be a SATE specification. The encoding of S based on the Explanatory Frame Axioms is the smallest set of propositional formulae, S^*, such that:

UNIVERSAL AXIOMS As in the encoding based on the classical frame axioms (in the previous section).

EXPLANATORY FRAME AXIOMS

$$(i\!:\!f \wedge \neg(i+1)\!:\!f) \supset$$
$$\bigvee\{i\!:\!\alpha\sigma \mid \texttt{action}(\alpha, c, pre, add, del) \in S, \alpha\sigma \text{ is ground, } c\sigma \text{ holds, and } f \in del\sigma\} \in S^*$$

and

$$(\neg i\!:\!f \wedge (i+1)\!:\!f) \supset$$
$$\bigvee\{i\!:\!\alpha\sigma \mid \texttt{action}(\alpha, c, pre, add, del) \in S, \alpha\sigma \text{ is ground, } c\sigma \text{ holds, and } f \in add\sigma\} \in S^*$$

for all ground instances of fluents f and $0 \leq i \leq n-1$.

Exclusion Axioms

$$\neg(i\!:\!\alpha_1\sigma_1 \wedge i\!:\!\alpha_2\sigma_2) \in S^*$$

for all α_1, α_2 such that $\texttt{action}(\alpha_1, c_1, pre_1, add_1, del_1) \in S$, $\texttt{action}(\alpha_2, c_2, pre_2, add_2, del_2) \in S$, $\alpha_1\sigma_1$ and $\alpha_2\sigma_2$ are ground and such that $\alpha_1\sigma_1 \neq \alpha_2\sigma_2$, $c_1\sigma_1$ and $c_2\sigma_2$ hold, and $0 \leq i \leq n-1$. To avoid the generation of redundant formulae it is convenient to include the additional requirement $\alpha_1\sigma_1 \prec \alpha_2\sigma_2$, where $\prec$ is a lexicographic ordering over actions.

To instruct SATE to use the encoding based on the explanatory frame axioms the parameter `encoding` must be set to the list `[ape_axiom,explanatory_frame_axiom,conflict_exclusion_axiom])`.

6.3 Complexity Analysis

The number of literals is in $O(n|\mathcal{F}| + n|\mathcal{A}|)$ for both encodings. The number of formulae is in $O(n|\mathcal{A}||\mathcal{F}|)$ in the encoding based on the classical frame axioms, whereas it is in $O(n|\mathcal{F}| + n|\mathcal{A}|^2)$ in the encoding based on the explanatory frame axioms. Finally, the total size of the encoding is in $O(n|\mathcal{A}||\mathcal{F}|)$ in the encoding based on the classical frame axioms, and it is in $O(n|\mathcal{A}||\mathcal{F}| + n|\mathcal{A}|^2)$ in the encoding based on the explanatory frame axioms.

If we consider the number of formulae, we can conclude that the encoding based on the explanatory frame axioms is better than that based on the classical frame axioms when $|\mathcal{A}| \ll |\mathcal{F}|$. This observation seems to be contradicted when we consider the total size of the encoding. However it must be noted that the worst case analysis in the encoding based on the explanatory frame axioms assumes that all the actions occur in each explanatory frame axioms. However this situation rarely occurs in practice, as usually only a few actions affect the truth value of any given fluent. If the number of actions affecting the truth value of fluents is bounded by a number k (which is usually small), then the total size of the encoding based on the explanatory frame axioms becomes $O(nk|\mathcal{F}| + n|\mathcal{A}|^2)$, thereby confirming the conclusion that the encoding based on the explanatory frame axioms is better than that based on the classical frame axioms if $|\mathcal{A}| \ll |\mathcal{F}|$.

6.4 Invariant-based Simplification

Let S be a SATE specification we define R_S to be the following set of rewrite rules:

$$\begin{array}{rl} R_S = & \{p \rightarrow \texttt{true} \mid \texttt{invariant}(p) \in S\} \cup \\ & \{p \rightarrow \texttt{false} \mid \texttt{invariant(\~{}}p\texttt{)} \in S\} \cup \\ & \{p \rightarrow q \mid \texttt{invariant(}p\texttt{<=>}q\texttt{)} \in S\} \cup \\ & \{s \rightarrow t \mid \texttt{invariant(}s\texttt{=}t\texttt{)} \in S\} \end{array}$$

The formulae generated by the encoding algorithms are simplified by normalizing their atomic formulae w.r.t. R_S and then performing standard boolean simplifications.[8] This may drastically reduce the size of the formulae.

6.5 Clausification

The simplified formulae are then clausified using a standard clausification technique.

6.6 SAT Solvers

The current version of SATE supports the interface to two state-of-the-art SAT solvers: SATO [9] and Chaff [7]. Interfacing SATE to other solvers is straightforward but it is left for the future work. The selection of the SAT solver is done by setting the parameter `solver` to either `sato` or `chaff`.

[8] To ensure termination of the normalization process we use ordered rewriting (see, e.g., [3]) using the Prolog `@<` ordering relation to compare the rewritten atomic formula with the original one.

```
lrr(step_0,protocol_Rules,
[
    h(s(xTime)),
    w(0,xSe0,xA,[],[xA,xB,xKa,primed(xKa),xKb],xbool,xc)
],
[
    h(xTime),
    m(1,xA,xA,xB,crypt(xKb,c(nonce(c(na,xTime)),xA)),xc),
    w(2,xB,xA,[nonce(c(na,xTime))],[xA,xB,xKa,primed(xKa),xKb],true,xc)
]
)

lrr(step_1,protocol_Rules,
[
    h(s(xTime)),
    m(1,xr,xA,xB,crypt(xKb,c(xNa,xA)),xc2),
    w(1,xA,xB,[],[xB,xKa,xKb,primed(xKb)],xbool,xc)
],
[
    h(xTime),
    m(2,xB,xB,xA,crypt(xKa,c(xNa,nonce(c(nb,xTime)))),xc2),
    w(3,xA,xB,[xA,crypt(xKb,c(xNa,xA)),nonce(c(nb,xTime))],[xB,xKa,xKb,primed(xKb)],true,xc)
]
)

lrr(step_2,protocol_Rules,
[
    h(s(xTime)),
    m(2,xr,xB,xA,crypt(xKa,c(xNa,xNb)),xc2),
    w(2,xB,xA,[xNa],[xA,xB,xKa,primed(xKa),xKb],xbool,xc)
],
[
    h(xTime),
    m(3,xA,xA,xB,crypt(xKb,xNb),xc2),
    w(0,xSe0,xA,[],[xA,xB,xKa,primed(xKa),xKb],true,s(xc))
]
)

lrr(step_3,protocol_Rules,
[
    h(s(xTime)),
    m(3,xr,xA,xB,crypt(xKb,xNb),xc2),
    w(3,xA,xB,[xA,crypt(xKb,c(xNa,xA)),xNb],[xB,xKa,xKb,primed(xKb)],xbool,xc)
],
[
    h(xTime),
    w(1,xA,xB,[],[xB,xKa,xKb,primed(xKb)],true,s(xc))
]
)
```

Figure 6: Protocol rewrite rules of the NSPK protocol

```
super_sort(knw_el,inknw_el).

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


constant(0,time).
constant(s(time),time).
constant(true,bool).
constant(false,bool).
constant(s(session),session).
constant(f(session),fsecrecy).



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

```
constant(h(time),fluent).
constant(wk(wstep,mr,mr,knw_el,acq_index,bool,session),fluent).
constant(inknw(mr,inknw_el,in_index,session),fluent).
constant(i(knw_el),fluent).
constant(secret(knw_el,fsecrecy),fluent).

constant(etc,knw_el).
constant(etc,inknw_el).

constant(pk(public_key),pk).
constant(pk(private_key),pkinv).
constant(primed(pkinv),pk).
constant(sk(symmetric_key),sk).
constant(fu(fu_symbol),fu).
constant(mr(user),mr).
constant(nonce(fresh_nonce),nonce).

constant(pk(fresh_public_key),pk).
constant(pk(fresh_private_key),pkinv).
constant(sk(fresh_symmetric_key),sk).
constant(pk(fresh_intruder_const),pk).
constant(pk(fresh_intruder_const),pkinv).
constant(sk(fresh_intruder_const),sk).
constant(nonce(fresh_intruder_const),nonce).

constant(c(fresh_public_id,time),fresh_public_key).
constant(c(fresh_private_id,time),fresh_private_key).
constant(c(fresh_symmetric_id,time),fresh_symmetric_key).
constant(c(fresh_nonce_id,time),fresh_nonce).
constant(c(intruder_const,intruder_const),fresh_intruder_const).
```

Figure 7: Protocol independent declarations using the optimizations

Bibliography

[1] J. Clark and J. Jacob. A survey of authentication protocol literature: Version, 1997.

[2] Alain Colmerauer. Les grammaires de metamorphose. Technical report, Groupe d'Intelligence Artificielle, Université de Marseille-Luminy, November 1975.

[3] N. Dershowitz and J.P. Jouannaud. Rewriting systems. In *Handbook of Theoretical Computer Science*, pages 243–320. Elsevier Publishers, Amsterdam, 1990.

[4] Michael D. Ernst, Todd D. Millstein, and Daniel S. Weld. Automatic SAT-compilation of planning problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 1169–1177, San Francisco, August 23–29 1997. Morgan Kaufmann Publishers.

[5] Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3–4):189–208, 1971.

[6] Henry Kautz, David McAllester, and Bart Selman. Encoding plans in propositional logic. In Luigia Carlucci Aiello, Jon Doyle, and Stuart Shapiro, editors, *KR'96: Principles of Knowledge Representation and Reasoning*, pages 374–384. Morgan Kaufmann, San Francisco, California, 1996.

[7] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, June 2001.

[8] Fernando C. N. Pereira and David H. D. Warren. Definite clause grammars for language analysis. In Karen Sparck-Jones Barbara J. Grosz and Bonnie Lynn Webber, editors, *Readings in Natural Language Processing*, pages 101–124. Morgan Kaufmann, Los Altos, 1980.

[9] H. Zhang. SATO: An efficient propositional prover. In William McCune, editor, *Proceedings of the 14th International Conference on Automated deduction*, volume 1249 of *LNAI*, pages 272–275, Berlin, July 13–17 1997. Springer.