AVISS — Automated Verification of Infinite State Systems

(IST-2000-26410)

Deliverable D2.2

Specification of the Intermediate Format IF

1 Introduction

In deliverable D2.1 we specified the High Level Protocol Specification Language (HLPSL) and hence the front-end of a compiler from HLPSL to Intermediate Format (IF). In this deliverable we specify the IF and hence the back-end of the HLPSL2IF compiler.

The output side of the compiler is prone to evolve according to the development of the tools that work on the IF. To be flexible in experiments, we first describe here a grammar that matches a super-set of the output of the HLPSL2IF compiler: the generic IF. This grammar is the specification of the parser of the tools that work on the IF. Second, we give a grammar matching exactly the current output of HLPSL2IF and a grammar that describes an experimental variant which excludes checking for type-flaws but allows faster search for security holes that are not type-flaws.

2 Notation

We will use throughout this document a standard notation for grammar, that is:

- $[xyz]$ is a "character class". This pattern, for example, matches either x or y or z.
- $x - y$ stands for an interval in a character class (all the characters between x and y).
- $\wedge$ is used for exclusion in a character class. For example, $[a - z^\wedge f - m]$ matches all the characters between a and z except those between f and m.
- $r?$, where r is a pattern, stands for zero or one occurrence of pattern r.
- $r+$, where r is a pattern, stands for one or more repetition of pattern r.
- $r*$, where r is a pattern, stands for zero or more repetition of pattern r.
- EOL stands for an End-Of-Line character.

In order to distinguish between characters and non-terminal symbols, we write characters in small caps, e.g. EXAMPLE, and non-terminal symbols in italics, e.g *example*.

In addition, we use the following shorthands:

Variable	::=	[x]*Text*
ConstText	::=	[a-zA- Z^x] *Text*
Time	::=	s(*Time*) | xTime
Digit	::=	[0-9]
Number	::=	*Digit+*
Char	::=	[a-zA-Z]
Text	::=	*(Digit* | *Char* | _ *)+*
GenericText	::=	[*ASCII(32)* - *ASCII(126)*]*

GenericText matches any sequence of printable character that does not contain the EOL character.

3 Intermediate Format File Description

In addition to rewrite rules that describe the protocol, we have to add, in the intermediate format, information that describes to what part of the protocol description a rule is related to. This information can then be used to give a specific status to some rules. We formalize this by giving, for the Intermediate Format file, a grammar that describes how rules are introduced. The current grammar for this description is given in Figure 1.

The rest of this deliverable focusses on the description of the *Rule* item.

[1]The *RuleIdentifier* parts of two different labelled rules are always different.

$$\begin{array}{rcl}
\textit{FileDescription} & ::= & \textit{Type EOL (LabelledRule} \mid \textit{CommentLine} \mid \textit{EOL)*} \\
\textit{Type} & ::= & \#\ \textsc{option=untyped} \mid \#\ \textsc{option=typed} \\
\textit{CommentLine} & ::= & \#\#\ \textit{GenericText EOL} \\
\textit{LabelledRule} & ::= & \textit{Label Rule EOL} \\
\textit{Label} & ::= & \#\ \textsc{lb}{=}\textit{RuleIdentifier}\ ,\ \textsc{type}{=}\textit{RuleSection EOL} \\
\textit{RuleIdentifier} & ::= & \textit{Text}^{1} \\
\textit{RuleSection} & ::= & \textsc{Protocol\_Rules} \mid \textsc{Invariant\_Rules} \mid \textsc{Decomposition\_Rules} \\
 & & \mid \textsc{Intruder\_Rules} \mid \textsc{Init} \mid \textsc{Goal} \mid
\end{array}$$

Figure 1: Description of the file containing the Intermediate Format

4 Generic Intermediate Format

As mentioned above, it is foreseeable that, in the project months to come, there will be some changes in the IF. We thus give now a grammar that is supposed to be stable and that provides a framework for experiments:

$$\begin{array}{rcl}
\textit{Rule} & ::= & \textit{State} => \textit{State} \mid \textit{State} \\
\textit{State} & ::= & \textit{Term} \mid \textit{Term} \cdot \textit{Term} \\
\textit{Term} & ::= & \textsc{h}\textit{(Time)} \mid \textit{Principal} \mid \textit{MessageTerm} \mid \textit{IntrudersKnowledge} \mid \textit{Secret} \mid \textit{Message} \\
\textit{Principal} & ::= & \textsc{w}\textit{(Message,Message,Message,Message,Message,Message,Message)} \\
\textit{MessageTerm} & ::= & \textsc{m}\textit{(Message,Message,Message,Message,Message,Message)} \\
\textit{IntrudersKnowledge} & ::= & \textsc{i}\textit{(Message)} \\
\textit{Secret} & ::= & \textsc{secret}\textit{(Message,}\textsc{f}\textit{(Message))}
\end{array}$$

According to this grammar, a rule is a transition from one state to a new state (except for rules that describe initial states and goal states). A state is a list of terms and a term is either *Principal* (expressing the state of one principal: his short- and long-term knowledge and the message he is expecting), *MessageTerm* (a message that is sent and not yet received), *IntrudersKnowledge* (what the intruder has already found out), or *Secret* (stating that some information must be kept secret).

And we define the messages with:

$$\begin{array}{rcl}
\textit{Message} & ::= & \textsc{crypt}\textit{(Message,Message)} \mid \textsc{scrypt}\textit{(Message,Message)} \\
 & & \mid \textsc{c}\textit{(Message,Message)} \mid \textsc{funct}\textit{(Message,Message)} \mid \textsc{rcrypt}\textit{(Message,Message)} \\
 & & \mid \textsc{mr}\textit{(Message)} \mid \textsc{nonce}\textit{(Message)} \mid \textsc{pk}\textit{(Message)} \mid \textsc{sk}\textit{(Message)} \\
 & & \mid \textsc{fu}\textit{(Message)} \mid \textit{Message'} \mid \textit{ConstText} \mid \textit{Number} \mid \textit{Variable} \mid \textit{Time}
\end{array}$$

An atomic message like a key, a nonce, or a principal name is declared by a name of alphanumeric characters. It can be thought of as representing a sequence of bits in real world protocols. Type information is added to these atomic items like $\textsc{mr(a)}$, $\textsc{sk(kab)}$, or $\textsc{pk(ka)}$. This type information determines which properties the Atom has, e.g. we assume

$$\textsc{scrypt(sk(kab),scrypt(sk(kab),xMessage))} = \textsc{xMessage}\ .$$

A message can now be an atomic message or a composed message (e.g. coupling $\textsc{c}$, encryption $\textsc{crypt}$, etc.).

5 Current Intermediate Format

In the previous section we gave a generic framework for the IF. However there are for instance many message terms that don't make sense like $\textsc{pk}(\textsc{sk}(ka))$. In this section we give a tighter description of the messages and rules that can occur in the output of HLPSL2IF. We first consider the (default) untyped version and describe the differences for the typed version in the next section.

The grammar for HLPSL2IF's output can be split in two parts. The first of these is about terms exchanged during the protocol, and the second one is about the "administration of these terms", that is representation of messages, of principals, roles, intruder's knowledge and so on. First give the grammar for the messages:

$$\begin{array}{rcl}
\mathit{Message} & ::= & \mathit{Binary} \mid \mathit{Unary} \\
\mathit{Binary} & ::= & \mathit{Couple} \mid \mathit{SymmetricCypher} \mid \mathit{PublicCypher} \mid \mathit{Function} \mid \mathit{XOR} \\
\mathit{Couple} & ::= & \textsc{c}(\mathit{Message},\mathit{Message}) \\
\mathit{SymmetricCypher} & ::= & \textsc{scrypt}(\textsc{sk}(\mathit{Atomic}),\mathit{Message}) \mid \textsc{scrypt}(\mathit{Variable},\mathit{Message}) \\
\mathit{PublicCypher} & ::= & \textsc{crypt}(\textsc{pk}(\mathit{Atomic}),\mathit{Message}) \mid \textsc{crypt}(\mathit{Variable}',\mathit{Message}) \\
 & & \textsc{crypt}(\textsc{pk}(\mathit{Atomic})',\mathit{Message}) \mid \textsc{crypt}(\mathit{Variable},\mathit{Message}) \\
\mathit{Function} & ::= & \textsc{funct}(\textsc{fu}(\mathit{Atomic}),\mathit{Message}) \mid \textsc{funct}(\mathit{Variable},\mathit{Message}) \\
\mathit{XOR} & ::= & \textsc{rcrypt}(\mathit{Message},\mathit{Message}) \\
\mathit{Unary} & ::= & \mathit{Variable} \mid \textsc{mr}(\mathit{Atomic}) \mid \textsc{pk}(\mathit{Atomic}) \mid \textsc{pk}(\mathit{Atomic})' \mid \textsc{sk}(\mathit{Atomic}) \mid \\
 & & \textsc{nonce}(\mathit{Atomic}) \mid \textsc{fu}(\mathit{Atomic}) \\
\mathit{Atomic} & ::= & \mathit{Const} \\
\mathit{Const} & ::= & \mathit{FreshConst} \mid \mathit{ConstText} \\
\mathit{FreshConst} & ::= & \textsc{c}(\mathit{ConstText},\mathit{Time}) \mid \textsc{c}(\textsc{ni},\textsc{ni})
\end{array}$$

Let us now consider the roles description. Formally, we represent *roles* when using the `--rules` option, and *principals* when using the `--init` option. And during a protocol execution, roles get naturally instantiated as principals. In order to do this, the terms representing principals and roles have exactly the same structure. The grammar is:

$$\begin{array}{rcl}
\mathit{Principal} & ::= & \textsc{w}(\mathit{Step},\mathit{Sender},\mathit{Receiver},\mathit{AcqKnowledge},\mathit{InitialKnowledge},\mathit{Boolean},\mathit{Session}) \\
\mathit{Step} & ::= & \mathit{Number} \mid \mathit{Variable} \\
\mathit{Sender} & ::= & \mathit{Id} \\
\mathit{Receiver} & ::= & \mathit{Id} \\
\mathit{Id} & ::= & \textsc{mr}(\mathit{Atomic}) \mid \mathit{Variable} \\
\mathit{AcqKnowledge} & ::= & \mathit{ListOfMessages} \\
\mathit{InitialKnowledge} & ::= & \mathit{ListOfMessages} \\
\mathit{ListOfMessages} & ::= & \textsc{etc} \mid \textsc{etc2} \mid \mathit{Variable} \mid \textsc{c}(\mathit{Message},\mathit{ListOfMessages}) \\
\mathit{Boolean} & ::= & \textsc{true} \mid \textsc{false} \mid \mathit{Variable} \\
\mathit{Session} & ::= & \textsc{s}(\mathit{Session}) \mid \mathit{Number} \mid \mathit{Variable}
\end{array}$$

The message description is based on the same idea:

$$\begin{array}{rcl}
\mathit{MessageTerm} & ::= & \textsc{m}(\mathit{Step},\mathit{RealSender},\mathit{OfficialSender},\mathit{Receiver},\mathit{Contents},\mathit{Session}) \\
\mathit{RealSender} & ::= & \mathit{Id} \\
\mathit{OfficialSender} & ::= & \mathit{Id} \\
\mathit{Receiver} & ::= & \mathit{Id} \\
\mathit{Contents} & ::= & \mathit{Message}
\end{array}$$

The Intruder's knowledge description is:

$$\mathit{IntrudersKnowledge} \quad ::= \quad \textsc{i}(\mathit{Message})$$

A state of the protocol is given as a non-empty multiset of terms, in the following way:

$$\begin{array}{rcl}
\mathit{State} & ::= & \mathit{Term} \mid \mathit{State} \cdot \mathit{Term} \\
\mathit{Term} & ::= & \mathit{Principal} \mid \mathit{MessageTerm} \mid \mathit{IntrudersKnowledge} \mid \mathit{Secret} \\
\mathit{Secret} & ::= & \textsc{secret}(\mathit{Variable},\textsc{f}(\mathit{Session}))
\end{array}$$

We can now give rewrite rules on these states:

Rule	::=	*Initial* | *MessageRule* | *IntruderRule* | *FlawRule* | *SimplificationRule*
Initial	::=	h(xTime)· *State*
MessageRule	::=	*InitialMessageRule* | *RegularMessageRule* | *LastMessageRule*
InitialMessageRule	::=	h(s(xTime)) · *Principal* =>
		h(xTime)· *MessageTerm* · *Principal* · *Secret*
RegularMessageRule	::=	h(s(xTime))· *MessageTerm* · *Principal* =>
		h(xTime)· *MessageTerm* · *Principal* · *Secret*
LastMessageRule	::=	h(s(xTime))· *MessageTerm* · *Principal* =>
		h(xTime) · *Principal* . *Secret*
IntruderRule	::=	*Memorize* | *Divert* | *EavesDropping* | *Impersonate*
Memorize	::=	h(s(xTime)) . m(x1,x2,x3,mr(i),x5,x6) =>
		h(xTime) . i(x2) . i(x5)
Divert	::=	h(s(xTime)) . m(x1,mr(Const),x3,x4,x5,x6) =>
		h(xTime) . i(mr(Const)) . i(x3) . i(x4) . i(x5)
EavesDropping	::=	h(s(xTime)) . m(x1,mr(Const),x3,x4,x5,x6) =>
		h(xTime) . m(x1,mr(Const),x3,x4,x5,x6) . i(mr(Const)).i(x3).i(x4). i(x5)
Impersonate	::=	h(s(xTime)) . *Principal* (. *IntrudersKnowledge*)+ =>
		h(xTime) . *MessageTerm* . *Principal* (. *IntrudersKnowledge*)+
FlawRule	::=	*State*
SimplificationRule	::=	*State* => *State* | *IdempotenceRule* | *SpecialRule*
IdempotenceRule	::=	x1 . x1 => x1
SpecialRule	::=	f(s(x1)) => f(x1)

6 The Typed Version

Note that in the above grammar variables are not typed, e.g. a term like mr(xA) will never occur, only constants are typed like mr(alice). While this partially permits to model type-flaws (mistaking a nonce for a symmetric key, for example), this leads to a branching explosion in the rules where the intruder composes a message (impersonate rules). As a consequence, we have defined a "typed" version of the Intermediate Format that restricts the domain of variables by specifying a type.

For example, suppose that the expected message is an unknown nonce Na. In the "untyped" version of the Intermediate Format, the rule that permits the intruder to compose a message is of the form:

PrincipalTerm . i(xNa) => m(...,xNa,...) . *PrincipalTerm* . i(xNa)

This rule can be used with various values for the variable xNa, including a symmetric key, or an un-decomposed cypher and so on. The number of possibilities is the number of pieces of knowledge the intruder has. On the other hand, when one keeps the type information, as in the "typed" version, this rule becomes:

PrincipalTerm . i(nonce(xNa)) => m(...,nonce(xNa),...) . *PrincipalTerm* . i(nonce(xNa))

In this case, this rule can only be applied with a piece of knowledge of the intruder that is also "typed" with nonce, thus reducing the possibilities of application. The drawback is that it is impossible to find type flaws in this case. The fact that the output of HLPSL2IF is typed or not is stamped by the first line of the file in the Intermediate Format (see Figure 1). The rest of the file follows the same grammar as in the "untyped" version, except for the five following rules, which deal with the construction of message terms:

SymmetricCypher	::=	scrypt(sk(*Atomic*),*Message*)
PublicCypher	::=	crypt(pk(*Atomic*),*Message*) | crypt(pk(*Atomic*)',*Message*)
Function	::=	funct(fu(*Atomic*),*Message*)
Atomic	::=	*Const* | *Variable*
Id	::=	mr(*Atomic*)