1 Introduction

This manual describes the ProVerif software package version 1.87beta. Additional information can be found in the research paper, in the file docs/paper.pdf in this distribution. ProVerif is a tool for automatically analyzing the security of cryptographic protocols. In particular, ProVerif is capable of proving observational equivalence between processes. The aim of this beta is to extend the class of equivalences that ProVerif handles. As such, this beta can only take input file with an equivalence query. Any other query has beendisabled. Moreover, only typed processes are allowed in this beta. All these restrictions will be lifted in the next stable release.

2 Definition of destructors

In ProVerif version 1.86pl4, destructors are modelled using rewrite rules of the form:

\[
\text{reduc}\ \text{forall } x_{1,1} : t_{1,1}, \ldots, x_{1,n_1} : t_{1,n_1} ;
\]
\[
g(M_{1,1}, \ldots, M_{1,k}) = M_{1,0} ;
\]

\[
\ldots
\]

\[
\text{forall } x_{m,1} : t_{m,1}, \ldots, x_{m,n_m} : t_{m,n_m} ;
\]
\[
g(M_{m,1}, \ldots, M_{m,k}) = M_{m,0}.
\]

where \(g\) is a destructor of arity \(k\). The terms \(M_{1,1}, \ldots, M_{1,k}, M_{1,0}\) are built from the application of constructors to variables \(x_{1,1}, \ldots, x_{1,n_1}\) of types \(t_{1,1}, \ldots, t_{1,n_1}\) respectively (and similarly for the other rewrite rules). The return type of \(g\) is the type \(M_{1,0}\) and \(M_{1,0}, \ldots, M_{m,0}\) must have the same type. We similarly require that the arguments of the destructor have the same type; that is, \(M_{i,1}, \ldots, M_{i,k}\) have the same types as \(M_{i,1}, \ldots, M_{i,k}\) for \(i \in [2, m]\), and these types are the types of the arguments of \(g\). The variables that occur in \(M_{1,0}\) must also occur in \(M_{i,1}, \ldots, M_{i,k}\).

When the term \(g(M_{1,1}, \ldots, M_{1,k})\) (or an instance of that term) is encountered during execution, it is replaced by \(M_{1,0}\), and similarly for the other rewrite rules. When several rules can be applied, one possibility is chosen (but ProVerif considers all possibilities when reasoning). When no rule can be applied, the destructor fails.

2.1 Changes in ProVerif version 1.87beta

Although this beta continues to accept destructors defined as in ProVerif version 1.86pl4, we introduce a new way to model the behaviour of destructors.

\[
\text{fun } g(t_1, \ldots, t_k) : t
\]

\[
\text{reduc}\ \text{forall } x_{1,1} : t_{1,1}, \ldots, x_{1,n_1} : t_{1,n_1} ;
\]
\[
g(M_{1,1}, \ldots, M_{1,k}) = M_{1,0}
\]
otherwise ...
otherwise forall \( x_{m,1} : t_{m,1}, \ldots, x_{m,n_m} : t_{m,n_m} ; g(M_{m,1}, \ldots, M_{m,k}) = M_{m,0} \).

One can see this declaration as a sequence of rewrite rules instead of a set of rewrite rules. Thus, when the term \( g(N_1, \ldots, N_n) \) is encountered, ProVerif will try to apply the first rewrite rule of the sequence, \( \text{forall } x_{1,1} : t_{1,1}, \ldots, x_{1,n_1} ; g(M_{1,1}, \ldots, M_{1,k}) = M_{1,0} \). If this rewrite rule is applicable, then the term \( g(N_1, \ldots, N_n) \) is replaced by \( M_{1,0} \) (or an instance of that term), else ProVerif try the second rewrite rule of the sequence and so on. If no rule can be applied, the destructor fails. This definition of destructors allows one to define new destructors that could not be defined in ProVerif version 1.86pl4.

1 fun eq (bitstring, bitstring): bool
2    reduc forall x: bitstring; eq(x, x) = true
3    otherwise forall x: bitstring, y: bitstring; eq(x, y) = false.

With this definition, \( eq(M, N) \) can be reduced to false only if \( M \) and \( N \) are different modulo the equational theory. Stress that if the function \( eq \) was defined in the previous model as follows:

5    reduc forall x: bitstring; eq(x, x) = true;
6    forall x: bitstring, y: bitstring; eq(x, y) = false.

then \( eq(M, M) \) could be reduced to false and true.

As previously mentioned, when no rule can be applied, the destructor fails. However, this formalism does not allow a destructor to succeed when one of its arguments fails. To lift this restriction, we allow to represent the case of failure by the special value \text{fail}.

8 fun test (bool, bitstring, bitstring): bitstring
9    reduc
10       forall x: bitstring, y: bitstring; test(true, x, y) = x
11       otherwise forall c: bool, x: bitstring; test(c, x, y) = y
12       otherwise forall x: bitstring, y: bitstring; test(fail, x, y) = y.

In the previous example, the function test returns the third argument even when the first argument fails. A variable \( x \) of type \( t \) can be declared as a possible failure by the syntax: \( x: t \) or \text{fail}. It indicates that \( x \) can be any message or even the special value \text{fail}. Relying on this new declaration of variables, the destructor test could have been defined as follows:

14 fun test (bool, bitstring, bitstring): bitstring
15    reduc
16       forall x: bitstring, y: bitstring; test(true, x, y) = x
17       otherwise forall c: bool or fail, x: bitstring, y: bitstring;
18       test(c, x, y) = y.

The grammar for the declaration of a destructor is given in Figure 1. We adopt the following conventions. \( X^* \) means any number of repetitions of \( X \); and \( [X] \) means \( X \) or nothing. \( \text{seq}(X) \) is a sequence of \( X \), that is, \( \text{seq}(X) = [(X), \ldots, X, \ldots, ] \). (The sequence can be empty, it can be one element, or it can be several elements separated by commas.) \( \text{seq}^+(X) \) is a non-empty sequence of \( X \): \( \text{seq}^+(X) = (\langle X \rangle, \ldots, X, \ldots, \langle X \rangle) \). (It can be one or several elements of \( \langle X \rangle \) separated by commas.) Text in typewriter style should appear as it is in the input file. Text between \( \langle \) and \( \rangle \) represents non-terminals of the grammar. In particular, we will use:

- \langle ident \rangle to denote identifiers which range over an unlimited sequence of letters (a-z, A-Z), digits (0-9), underscores (_), single-quotes (’), and accented letters from the ISO Latin 1 character set where the first character of the identifier is a letter and the identifier is distinct from the reserved words of the language.
- \langle typeid \rangle to denote types, which can be identifiers \langle ident \rangle or the reserved word \text{channel}. 

2
Figure 1 New grammar for the declaration of a destructor

\[
\begin{align*}
\langle \text{term} \rangle & ::= \langle \text{ident} \rangle \\
& \quad | \langle \text{seq}(\text{term}) \rangle \\
& \quad | \langle \text{ident} \rangle \langle \text{seq}(\text{term}) \rangle \\
\end{align*}
\]

\[
\langle \text{mayfailterm} \rangle ::= \langle \text{term} \rangle \\
& \quad | \text{fail} \\
\langle \text{vardecl} \rangle ::= \langle \text{ident} \rangle : \langle \text{typeid} \rangle \\
& \quad | \langle \text{vardecl} \rangle,
\]

\[
\begin{align*}
\langle \text{reduc} \rangle & ::= \forall \langle \text{vardecl} \rangle; \langle \text{ident} \rangle \langle \text{seq}(\text{mayfailterm}) \rangle = \langle \text{mayfailterm} \rangle \text{ otherwise } \langle \text{reduc} \rangle \\
\langle \text{declaration} \rangle & ::= \text{fun} \langle \text{ident} \rangle \langle \text{seq}(\text{typeid}) \rangle : \langle \text{typeid} \rangle \langle \text{reduc} \rangle \text{ [private]}.
\end{align*}
\]

3 Observational equivalence

The most general class of equivalences that ProVerif version 1.86pl4 can prove are equivalences $P \approx Q$ where the processes $P$ and $Q$ have the same structure and differ only in the choice of terms. These equivalences are written in ProVerif by a single “biprocess” that encodes both $P$ and $Q$. Such a biprocess uses the construct \texttt{choice}[$M$, $M'$] to represent the terms that differ between $P$ and $Q$: $P$ uses the first component of the choice, $M$, while $Q$ uses the second one, $M'$. For example:

1. process
2. new sk.a:bitstring; new sk.b:bitstring; new sk.c:bitstring;
3. system(\texttt{choice}[sk.a,sk.c],sk.b)

with system a previously defined process.

3.1 Changes in ProVerif version 1.87beta

In ProVerif version 1.87beta, we provide an automatic procedure that translates a biprocess into possibly several observationally equivalent biprocesses. Hence, if ProVerif fails to prove the equivalence on the input biprocess, it will compute the equivalent simplified biprocesses and try to prove the equivalence on at least one of them. The parameters below is now supported:

\[
\begin{align*}
\text{set simplifyProcess} &= \text{yes. (default value)} \\
\text{set simplifyProcess} &= \text{no}. \\
\text{set simplifyProcess} &= \text{interactif}.
\end{align*}
\]

With the setting \texttt{set simplifyProcess} = \text{no}, ProVerif does not compute the simplified biprocesses. With the setting \texttt{set simplifyProcess} = \text{interactif}, an interactif menu appears when ProVerif fails to prove the equivalence on the input biprocess. This menu allows one to either view the different simplified biprocesses or select one of the simplified biprocess for ProVerif to prove. At last, in ProVerif version 1.87beta, it is no longer require to input a biprocess. ProVerif is now capable of proving equivalence between two processes $P$ and $Q$ that do not have the same structure which is represented by the following command

\[
\text{equivalence } P \ Q
\]

where $P$ and $Q$ are processes that do not contain \texttt{choice}. ProVerif will in fact try to merge the processes $P$ and $Q$ into a biprocess and then prove equivalence of this biprocess. Note that ProVerif is not always capable of merging two processes into a biprocess.