ProVerif 2.02: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial

Bruno Blanchet, Ben Smyth, Vincent Cheval, and Marc Sylvestre

Bruno.Blanchet@inria.fr, research@bensmyth.com, vincent.cheval@inria.fr, marc.sylvestre@inria.fr

July 31, 2020

Acknowledgements

This manual was written with support from the Direction Générale pour l'Armement (DGA) and the EPSRC project *UbiVal* (EP/D076625/2). ProVerif was developed while Bruno Blanchet was affiliated with INRIA Paris-Rocquencourt, with CNRS, Ecole Normale Supérieure, Paris, and with Max-Planck-Institut für Informatik, Saarbrücken. This manual was written while Bruno Blanchet was affiliated with INRIA Paris-Rocquencourt and with CNRS, Ecole Normale Supérieure, Paris, Ben Smyth was affiliated with Ecole Normale Supérieure, Paris and with University of Birmingham, Vincent Cheval was affiliated with CNRS and Inria Nancy, and Marc Sylvestre was affiliated with INRIA Paris. The development of ProVerif would not have been possible without the helpful remarks from the research community; their contributions are greatly appreciated and further feedback is encouraged.

Contents

1 Introduction **1**
1.1 Applications of ProVerif . . . 1
1.2 Scope of this manual . . . 2
1.3 Support . . . 2
1.4 Installation . . . 2
1.4.1 Installation via OPAM . . . 3
1.4.2 Installation from sources (Linux/Mac/cygwin) . . . 3
1.4.3 Installation from binaries (Windows) . . . 4
1.4.4 Emacs . . . 5
1.4.5 Atom . . . 5
1.5 Copyright . . . 5

2 Getting started **7**

3 Using ProVerif **11**
3.1 Modeling protocols . . . 11
3.1.1 Declarations . . . 11
3.1.2 Example: Declaring cryptographic primitives for the handshake protocol . . . 13
3.1.3 Process macros . . . 14
3.1.4 Processes . . . 15
3.1.5 Example: handshake protocol . . . 17
3.2 Security properties . . . 18
3.2.1 Reachability and secrecy . . . 19
3.2.2 Correspondence assertions, events, and authentication . . . 19
3.2.3 Example: Secrecy and authentication in the handshake protocol . . . 20
3.3 Understanding ProVerif output . . . 22
3.3.1 Results . . . 22
3.3.2 Example: ProVerif output for the handshake protocol . . . 23
3.4 Interactive mode . . . 30
3.4.1 Interface description . . . 30
3.4.2 Manual and auto-reduction . . . 31
3.4.3 Execution of 0, $P \mid Q$, $!P$, new, let, if, and event . . . 31
3.4.4 Execution of inputs and outputs . . . 32
3.4.5 Button “Add a term to public” . . . 33
3.4.6 Execution of insert and get . . . 33
3.4.7 Handshake run in interactive mode . . . 33
3.4.8 Advanced features . . . 34

4 Language features **35**
4.1 Primitives and modeling features . . . 35
4.1.1 Constants . . . 35
4.1.2 Data constructors and type conversion . . . 35
4.1.3 Natural numbers . . . 36
4.1.4 Enriched terms . . . 37

4.1.5 Tables and key distribution . . . 39
4.1.6 Phases . . . 39
4.1.7 Synchronization . . . 40
4.2 Further cryptographic operators . . . 41
4.2.1 Extended destructors . . . 41
4.2.2 Equations . . . 42
4.2.3 Function macros . . . 45
4.2.4 Process macros with fail . . . 46
4.2.5 Suitable formalizations of cryptographic primitives . . . 46
4.3 Further security properties . . . 49
4.3.1 Complex correspondence assertions, secrecy, and events . . . 50
4.3.2 Observational equivalence . . . 55

5 Needham-Schroeder: Case study 63
5.1 Simplified Needham-Schroeder protocol . . . 64
5.1.1 Basic encoding . . . 64
5.1.2 Security properties . . . 65
5.2 Full Needham-Schroeder protocol . . . 68
5.3 Generalized Needham-Schroeder protocol . . . 70
5.4 Variants of these security properties . . . 74
5.4.1 A variant of mutual authentication . . . 74
5.4.2 Authenticated key exchange . . . 77
5.4.3 Full ordering of the messages . . . 82

6 Advanced reference 85
6.1 Proving correspondence queries by induction . . . 85
6.1.1 Single query . . . 85
6.1.2 Group of queries . . . 87
6.2 Axioms, restrictions, and lemmas . . . 89
6.3 Predicates . . . 95
6.4 Referring to bound names in queries . . . 98
6.5 Exploring correspondence assertions . . . 99
6.6 ProVerif options . . . 100
6.6.1 Command-line arguments . . . 100
6.6.2 Settings . . . 102
6.7 Theory and tricks . . . 111
6.7.1 The resolution strategy of ProVerif . . . 111
6.7.2 Performance and termination . . . 112
6.7.3 Alternative encodings of protocols . . . 118
6.7.4 Applied pi calculus encodings . . . 119
6.7.5 Sources of incompleteness . . . 120
6.7.6 Misleading syntactic constructs . . . 122
6.8 Compatibility with CryptoVerif . . . 123
6.9 Additional programs . . . 125
6.9.1 `test` . . . 125
6.9.2 `analyze` . . . 126
6.9.3 `addexpectedtags` . . . 127

7 Outlook 129

A Language reference 131

B Semantics 139

List of Figures

3.1 Handshake protocol . . . 12
3.2 Term and process grammar . . . 16
3.3 Pattern matching grammar . . . 16
3.4 Messages and events for authentication . . . 21
3.5 Handshake protocol attack trace . . . 28
3.6 Handshake protocol - Initial simulator window . . . 31
3.7 Handshake protocol - Simulator window 1 . . . 33
3.8 Handshake protocol - Simulator window 2 . . . 34
3.9 Handshake protocol - Simulator window 3 . . . 34

4.1 Natural number grammar . . . 36
4.2 Enriched terms grammar . . . 38
4.3 Grammar for correspondence assertions . . . 51

A.1 Grammar for terms . . . 132
A.2 Grammar for declarations . . . 133
A.3 Grammar for `not`, queries, and lemmas . . . 134
A.4 Grammar for `not`, queries, and lemmas restricted after parsing . . . 135
A.5 Grammar for `nounif` . . . 136
A.6 Grammar for `clauses` . . . 136
A.7 Grammar for processes . . . 137

B.1 Semantics of process terms and patterns . . . 142
B.2 Semantics of processes . . . 143

Chapter 1

Introduction

This manual describes the ProVerif software package version 2.02. ProVerif is a tool for automatically analyzing the security of cryptographic protocols. Support is provided for, but not limited to, cryptographic primitives including: symmetric and asymmetric encryption; digital signatures; hash functions; bit-commitment; and non-interactive zero-knowledge proofs. ProVerif is capable of proving reachability properties, correspondence assertions, and observational equivalence. These capabilities are particularly useful to the computer security domain since they permit the analysis of secrecy and authentication properties. Moreover, emerging properties such as privacy, traceability, and verifiability can also be considered. Protocol analysis is considered with respect to an unbounded number of sessions and an unbounded message space. Moreover, the tool is capable of attack reconstruction: when a property cannot be proved, ProVerif tries to reconstruct an execution trace that falsifies the desired property.

1.1 Applications of ProVerif

The applicability of ProVerif has been widely demonstrated. Protocols from the literature have been successfully analyzed: flawed and corrected versions of Needham-Schroeder public-key [NS78, Low96] and shared key [NS78, BAN89, NS87]; Woo-Lam public-key [WL92, WL97] and shared-key [WL92, AN95, AN96, WL97, GJ03]; Denning-Sacco [DS81, AN96]; Yahalom [BAN89]; Otway-Rees [OR87, AN96, Pau98]; and Skeme [Kra96]. The resistance to password guessing attacks has been demonstrated for the password-based protocols EKE [BM92] and Augmented EKE [BM93].

ProVerif has also been used in more substantial case studies:

- Abadi & Blanchet [AB05b] use correspondence assertions to verify the certified email protocol [AGHP02].
- Abadi, Blanchet & Fournet [ABF07] analyze the JFK (Just Fast Keying) [ABB+04] protocol, which was one of the candidates to replace IKE as the key exchange protocol in IPSec, by combining manual proofs with ProVerif proofs of correspondences and equivalences.
- Blanchet & Chaudhuri [BC08] study the integrity of the Plutus file system [KRS+03] on untrusted storage, using correspondence assertions, resulting in the discovery, and subsequent fixing, of weaknesses in the initial system.
- Bhargavan *et al.* [BFGT06, BFG06, BFGS08] use ProVerif to analyze cryptographic protocol implementations written in F#; in particular, the Transport Layer Security (TLS) protocol has been studied in this manner [BCFZ08].
- Chen & Ryan [CR09] evaluate authentication protocols found in the Trusted Platform Module (TPM), a widely deployed hardware chip, and discovered vulnerabilities.
- Delaune, Kremer & Ryan [DKR09, KR05] and Backes, Hritcu & Maffei [BHM08] formalize and analyze privacy properties for electronic voting using observational equivalence.

- Delaune, Ryan & Smyth [DRS08] and Backes, Maffei & Unruh [BMU08] analyze the anonymity properties of the trusted computing scheme Direct Anonymous Attestation (DAA) [BCC04, SRC07] using observational equivalence.

- Küsters & Truderung [KT09, KT08] examine protocols with Diffie-Hellman exponentiation and XOR.

- Smyth, Ryan, Kremer & Kourjieh [SRKK10, SRK10] formalize and analyze verifiability properties for electronic voting using reachability.

- Bhargavan, Blanchet and Kobeissi verify Signal [KBB17] and TLS 1.3 [BBK17].

- Blanchet verifies the ARINC 823 avionic protocols [Bla17].

For further examples, please refer to: `http://proverif.inria.fr/proverif-users.html`.

1.2 Scope of this manual

This manual provides an introductory description of the ProVerif software package version 2.02. The remainder of this chapter covers software support (Section 1.3) and installation (Section 1.4). Chapter 2 provides an introduction to ProVerif aimed at new users, advanced users may skip this chapter without loss of continuity. Chapter 3 demonstrates the basic use of ProVerif. Chapter 4 provides a more complete coverage of the features of ProVerif. Chapter 5 demonstrates the applicability of ProVerif with a case study. Chapter 6 considers advanced topics and Chapter 7 concludes. For reference, the complete grammar of ProVerif is presented in Appendix A. This manual does not attempt to describe the theoretical foundations of the internal algorithms used by ProVerif since these are available elsewhere (see Chapter 7 for references); nor is the applied pi calculus [AF01, RS11, ABF17], which provides the basis for ProVerif, discussed.

1.3 Support

Software bugs and comments should be reported by e-mail to:

`proverif-dev@inria.fr`

User support, general discussion and new release announcements are provided by the ProVerif mailing list. To subscribe to the list, send an email to `sympa@inria.fr` with the subject "subscribe proverif" (without quotes). To post on the list, send an email to:

`proverif@inria.fr`

Non-members are not permitted to send messages to the mailing list.

1.4 Installation

ProVerif is compatible with the Linux, Mac, and Windows operating systems; it can be downloaded from:

`http://proverif.inria.fr/`

The remainder of this section covers installation on Linux, Mac, and Windows platforms.

1.4.1 Installation via OPAM

ProVerif has been developed using Objective Caml (OCaml) and OPAM is the package manager of OCaml. Installing via OPAM is the simplest, especially if you already have OPAM installed.

1. If you do not already have OPAM installed, download it from

   `https://opam.ocaml.org/`

   and install it.

2. If you already have OPAM installed, run

   ```
   opam update
   ```

   to make sure that you get the latest version of ProVerif.

3. Run

   ```
   opam depext conf-graphviz
   opam depext proverif
   opam install proverif
   ```

   The first line installs graphviz, if you do not already have it. You may also install it using the package manager of your Linux, OSX, or cygwin distribution, especially if opam fails to install it. It is needed only for the graphical display of attacks.

   The second line installs GTK+2 including development libraries, if you do not already have it. You may also install it using the package manager of your distribution. You may additionally need to install `pkgconfig` using the package manager of your distribution, if you do not already have it and it is not installed by `opam depext proverif`. (This happens in particular on some OSX installations.) GTK+2 is needed for the interactive simulator `proverif_interact`.

   The third line installs ProVerif itself and its OCaml dependencies. ProVerif executables are in `~/.opam/`⟨*switch*⟩`/bin`, which is in the `PATH`, examples are in `~/.opam/`⟨*switch*⟩`/doc/proverif`, and various helper files are in `~/.opam/`⟨*switch*⟩`/share/proverif`. The directory ⟨*switch*⟩ is the opam switch in which you installed ProVerif, by default `system`.

4. Download the documentation package `proverifdoc2.02.tar.gz` from

   `http://proverif.inria.fr/`

   and uncompress it (e.g. using `tar -xzf proverifdoc2.02.tar.gz` or using your favorite file archive tool). That gives you the manual and a few additional examples.

1.4.2 Installation from sources (Linux/Mac/cygwin)

1. On Mac OS X, you need to install XCode if you do not already have it. It can be downloaded from `https://developer.apple.com/xcode/`.

2. ProVerif has been developed using Objective Caml (OCaml), accordingly OCaml version 4.03 or higher is a prerequisite to installation and can be downloaded from `http://ocaml.org/`, or installed via the package manager of your distribution. OCaml provides a byte-code compiler (`ocamlc`) and a native-code compiler (`ocamlopt`). Although ProVerif does not strictly require the native-code compiler, it is highly recommended to achieve large performance gains.

3. The installation of graphviz is required if you want to have a graphical representation of the attacks that ProVerif might found. Graphviz can be downloaded from `http://graphviz.org` or installed via the package manager of your distribution.

4. The installation GTK+2.24 and LablGTK2 is required if you want to run the interactive simulator `proverif_interact`. Use the package manager of your distribution to install GTK+2 including its development libraries if you do not already have it and download `lablgtk-2.18.6.tar.gz` from

```
http://lablgtk.forge.ocamlcore.org/
```

and follow the installation instructions in their README file.

5. Download the source package `proverif2.02.tar.gz` and the documentation package `proverifdoc2.02.tar.gz` from

   ```
   http://proverif.inria.fr/
   ```

6. Decompress the archives:

   (a) using GNU tar

   ```
   tar -xzf proverif2.02.tar.gz
   tar -xzf proverifdoc2.02.tar.gz
   ```

   (b) using tar

   ```
   gunzip proverif2.02.tar.gz
   tar -xf proverif2.02.tar
   gunzip proverifdoc2.02.tar.gz
   tar -xf proverifdoc2.02.tar
   ```

   This will create a directory `proverif2.02` in the current directory.

7. You are now ready to build ProVerif:

   ```
   cd proverif2.02
   ./build
   ```

   (If you did not install LablGTK2, the compilation of `proverif_interact` fails, but the executables `proverif` and `proveriftotex` are still produced correctly, so you can use ProVerif normally, but cannot run the interactive simulator.)

8. ProVerif has now been successfully installed.

1.4.3 Installation from binaries (Windows)

Windows users may install ProVerif using the binary distribution, as described below. They may also install cygwin and install ProVerif from sources as explained in the previous section.

1. The installation of graphviz is required if you want to have a graphical representation of the attacks that ProVerif might found. Graphviz can be downloaded from `https://graphviz.gitlab.io/_pages/Download/Download_windows.html`. Make sure that the `bin` subdirectory of the Graphviz installation directory is in your PATH.

2. The installation GTK+2.24 is required if you want to run the interactive simulator `proverif_interact`. At

   ```
   ftp://ftp.gnome.org/pub/gnome/binaries/win32/gtk+/2.24/
   ```

   download `gtk+-bundle_2.24.10-20120208_win32.zip`, unzip it in the directory `C:\GTK`, and add `C:\GTK\bin` to your PATH.

3. Download the Windows binary package `proverifbsd2.02.tar.gz` and the documentation package `proverifdoc2.02.tar.gz` from

   ```
   http://proverif.inria.fr/
   ```

4. Decompress the `proverifbsd2.02.tar.gz` and `proverifdoc2.02.tar.gz` archives in the same directory using your favorite file archive tool (e.g. WinZip).

5. ProVerif has now been successfully installed in the directory where the file was extracted.

1.4.4 Emacs

If you use the emacs text editor for editing ProVerif input files, you can install the emacs mode provided with the ProVerif distribution.

1. Copy the file `emacs/proverif.el` (if you installed by OPAM in the switch ⟨*switch*⟩, the file `~/.opam/`⟨*switch*⟩`/share/proverif/emacs/proverif.el`) to a directory where Emacs will find it (that is, in your emacs load-path).

2. Add the following lines to your `.emacs` file:

```
(setq auto-mode-alist
       (cons '("\\.horn$" . proverif-horn-mode)
       (cons '("\\.horntype$" . proverif-horntype-mode)
       (cons '("\\.pv[l]?$" . proverif-pv-mode)
       (cons '("\\.pi$" . proverif-pi-mode) auto-mode-alist)))))
(autoload 'proverif-pv-mode "proverif" "Major mode for editing ProVerif code." t)
(autoload 'proverif-pi-mode "proverif" "Major mode for editing ProVerif code." t)
(autoload 'proverif-horn-mode "proverif" "Major mode for editing ProVerif code." t)
(autoload 'proverif-horntype-mode "proverif" "Major mode for editing ProVerif code." t)
```

1.4.5 Atom

There is also a ProVerif mode for the text editor Atom (`https://atom.io/`), by Vincent Cheval. It can be downloaded from the Atom web site; the package name is `language-proverif`.

1.5 Copyright

ProVerif software (source) is distributed under the GNU general public license. For details see:

`http://proverif.inria.fr/LICENSEGPL`

The Windows binary distribution is under BSD license, for details see:

`http://proverif.inria.fr/LICENSEBSD`

Chapter 2

Getting started

This chapter provides a basic introduction to ProVerif and is aimed at new users; experienced users may choose to skip this chapter. ProVerif is a command-line tool which can be executed using the syntax:

`./proverif [options]` ⟨*filename*⟩

where `./proverif` is ProVerif's binary; ⟨*filename*⟩ is the input file; and command-line parameters `[options]` will be discussed later (Section 6.6.1). ProVerif can handle input files encoded in several languages. The typed pi calculus is currently considered to be state-of-the-art and files of this sort are denoted by the file extension `.pv`. This manual will focus on protocols encoded in the typed pi calculus. (For the interested reader, other input formats are mentioned in Section 6.6.1 and in `docs/manual-untyped.pdf`.) The pi calculus is designed for representing concurrent processes that interact using communications channels such as the Internet.

ProVerif is capable of proving reachability properties, correspondence assertions, and observational equivalence. This chapter will demonstrate the use of reachability properties and correspondence assertions in a very basic manner. The true power of ProVerif will be discussed in the remainder of this manual.

Reachability properties. Let us consider the ProVerif script:

```
1  (* hello.pv: Hello World Script *)
2
3  free c:channel.
4
5  free Cocks:bitstring [private].
6  free RSA:bitstring [private].
9
10 process
11   out(c,RSA);
12   0
```

Line 1 contains the comment *"hello.pv: Hello World Script"*; comments are enclosed by *(* comment *)*. Line 3 declares the *free name* c of type channel which will later be used for public channel communication. Lines 5 and 6 declare the free names Cocks and RSA of type bitstring, the keyword [**private**] excludes the names from the attacker's knowledge. Line 10 declares the start of the *main* process. Line 11 outputs the name RSA on the *channel* c. Finally, the termination of the process is denoted by 0 on Line 12.

Names may be of any type, but we explicitly distinguish names of type channel from other types, since the former may be used as a communications channel for message input/output. The concept of bound and free names is similar to local and global scope in programming languages; that is, free names are globally known, whereas bound names are local to a process. By default, free names are known by the attacker. Free names that are not known by the attacker must be declared *private* with the addition of the keyword [**private**]. The message output on Line 11 is broadcast using a *public channel* because the channel name c is a free name; whereas, if c were a bound name or explicitly excluded from the

attacker's knowledge, then the communication would be on a *private channel*. For convenience, the final line may be omitted and hence **out**(c,RSA) is an abbreviation of **out**(c,RSA);0.

Properties of the aforementioned script can be examined using ProVerif. For example, to test as to whether the names Cocks and RSA are available derivable by the attacker, the following lines can be included before the main process:

```
query attacker(RSA).
query attacker(Cocks).
```

Internally, ProVerif attempts to prove that a state in which the names Cocks and RSA are known to the attacker is unreachable (that is, it tests the queries **not attacker**(RSA) and **not attacker**(Cocks), and these queries are true when the names are *not* derivable by the attacker). This makes ProVerif suitable for proving the secrecy of terms in a protocol.

Executing ProVerif (`./proverif docs/hello.pv`) produces the output:

```
Process 0 (that is, the initial process):
{1}out(c, RSA)

-- Query not attacker(RSA[]) in process 0.
Translating the process into Horn clauses...
Completing...
Starting query not attacker(RSA[])
goal reachable: attacker(RSA[])

Derivation:

1. The message RSA[] may be sent to the attacker at output {1}.
attacker(RSA[]).

2. By 1, attacker(RSA[]).
The goal is reached, represented in the following fact:
attacker(RSA[]).


A more detailed output of the traces is available with
  set traceDisplay = long.

out(c, ~M) with ~M = RSA at {1}

The attacker has the message ~M = RSA.
A trace has been found.
RESULT not attacker(RSA[]) is false.
-- Query not attacker(Cocks[]) in process 0.
Translating the process into Horn clauses...
Completing...
Starting query not attacker(Cocks[])
RESULT not attacker(Cocks[]) is true.

--------------------------------------------------------------
Verification summary:

Query not attacker(RSA[]) is false.

Query not attacker(Cocks[]) is true.

--------------------------------------------------------------
```

As can be interpreted from `RESULT not attacker:(Cocks[]) is true`, the attacker has not been able to obtain the free name Cocks. The attacker has, however, been able to obtain the free name RSA as denoted by the `RESULT not attacker:(RSA[]) is false.` ProVerif is also able to provide an attack trace. In this instance, the trace is very short and denoted by

```
out(c, ~M) with ~M = RSA at {1}

The attacker has the message ~M = RSA.
```

which means that the name RSA is output on channel c at *point* {*1*} in the process and stored by the attacker in `~M`, where point {1} is annotated on Line 2 of the output. ProVerif concludes the trace by saying that the attacker has RSA. ProVerif also provides an English language description of the *derivation* denoted by

```
1. The message RSA[] may be sent to the attacker at output {1}.
attacker(RSA[]).

2. By 1, attacker(RSA[]).
The goal is reached, represented in the following fact:
attacker(RSA[]).
```

A derivation is the ProVerif internal representation of how the attacker may break the desired property, here may obtain RSA. It generally corresponds to an attack as in the example above, but may sometimes correspond to a false attack because of the internal approximations made by ProVerif. In contrast, when ProVerif presents a trace, it always corresponds to a real attack. See Section 3.3 for more details. The output ends with a summary of the results for all queries.

Correspondence assertions. Let us now consider an extended variant `docs/hello_ext.pv` of the script:

```
1  (* hello_ext.pv: Hello Extended World Script *)
2
3  free c:channel.
4
5  free Cocks:bitstring [private].
6  free RSA:bitstring [private].
7
8  event evCocks.
9  event evRSA.
10
11 query event(evCocks) ==> event(evRSA).
12
13 process
14   out(c,RSA);
15   in(c,x:bitstring);
16   if x = Cocks then
17     event evCocks;
18     event evRSA
19   else
20     event evRSA
```

Lines 1-7 should be familiar. Lines 8-9 declare events evCocks and evRSA. Intuition suggests that Line 11 is some form of query. Lines 13-14 should again be standard. Line 15 contains a message input of type bitstring on channel c which it binds to the variable x. Lines 16-20 denote an if-then-else statement; the body of the then branch can be found on Lines 17-18 and the else branch on Line 20. We remark that the code presented is a shorthand for the more verbose

```
if x = Cocks then event evCocks;event evRSA;0 else event evRSA;0
```

where 0 denotes the end of a branch (termination of a process). The statement **event** evCocks (similarly **event** evRSA) declares an event and the query

query event(evCocks) ==> **event**(evRSA)

is true if and only if, for all executions of the protocol, if the event evCocks has been executed, then the event evRSA has also been executed before. Executing the script produces the output:

```
Process 0 (that is, the initial process):
{1}out(c, RSA);
{2}in(c, x: bitstring);
{3}if (x = Cocks) then
    {4}event evCocks;
    {5}event evRSA
else
    {6}event evRSA

-- Query event(evCocks) ==> event(evRSA) in process 0.
Translating the process into Horn clauses...
Completing...
Starting query event(evCocks) ==> event(evRSA)
RESULT event(evCocks) ==> event(evRSA) is true.

--------------------------------------------------------------
Verification summary:

Query event(evCocks) ==> event(evRSA) is true.

--------------------------------------------------------------
```

As expected, it is not possible to witness the event evCocks without having previously executed the event evRSA and hence the correspondence **event**(evCocks) ==> **event**(evRSA) is true. In fact, a stronger property is true: the event evCocks is unreachable. The reader can verify this claim with the addition of **query event**(evCocks). (The authors remark that writing code with unreachable points is a common source of errors for new users. Advice on avoiding such pitfalls will be presented in Section 4.3.1.)

Chapter 3

Using ProVerif

The primary goal of ProVerif is the verification of cryptographic protocols. Cryptographic protocols are concurrent programs which interact using public communication channels such as the Internet to achieve some security-related objective. These channels are assumed to be controlled by a very powerful environment which captures an attacker with "Dolev-Yao" capabilities [DY83]. Since the attacker has complete control of the communication channels, the attacker may: read, modify, delete, and inject messages. The attacker is also able to manipulate data, for example: compute the ith element of a tuple; and decrypt messages if it has the necessary keys. The environment also captures the behavior of dishonest participants; it follows that only honest participants need to be modeled. ProVerif's input language allows such cryptographic protocols and associated security objectives to be encoded in a formal manner, allowing ProVerif to automatically verify claimed security properties. Cryptography is assumed to be perfect; that is, the attacker is only able to perform cryptographic operations when in possession of the required keys. In other words, it cannot apply any polynomial-time algorithm, but is restricted to apply only the cryptographic primitives specified by the user. The relationships between cryptographic primitives are captured using rewrite rules and/or an equational theory.

In this chapter, we demonstrate how to use ProVerif for verifying cryptographic protocols, by considering a naïve handshake protocol (Figure 3.1) as an example. Section 3.1 discusses how cryptographic protocols are encoded within ProVerif's input language, a variant of the applied pi calculus [AF01, RS11] which supports types; Section 3.2 shows the security properties that can be proved by ProVerif; and Section 3.3 explains how to understand ProVerif's output.

3.1 Modeling protocols

A ProVerif model of a protocol, written in the tool's input language (the typed pi calculus), can be divided into three parts. The *declarations* formalize the behavior of cryptographic primitives (Section 3.1.1); and their use is demonstrated on the handshake protocol (Section 3.1.2). Process *macros* (Section 3.1.3) allow sub-processes to be defined, in order to ease development; and finally, the protocol itself can be encoded as a *main* process (Section 3.1.4), with the use of macros.

3.1.1 Declarations

Processes are equipped with a finite set of types, free names, and constructors (function symbols) which are associated with a finite set of destructors. The language is strongly typed and user-defined types are declared as

type t.

All free names appearing within an input file must be declared using the syntax

free $n : t$.

where n is a name and t is its type. Several free names of the same type t can be declared by

free $n_1, \ldots, n_k : t$.

Figure 3.1 Handshake protocol

A naïve handshake protocol between client A and server B is illustrated below. It is assumed that each principal has a public/private key pair, and that the client A knows the server B's public key pk(skB). The aim of the protocol is for the client A to share the secret s with the server B. The protocol proceeds as follows. On request from a client A, server B generates a fresh symmetric key k (session key), pairs it with his identity (public key pk(skB)), signs it with his secret key skB and encrypts it using his client's public key pk(skA). That is, the server sends the message aenc(sign((pk(skB),k),skB),pk(skA)). When A receives this message, she decrypts it using her secret key skA, verifies the digital signature made by B using his public key pk(skB), and extracts the session key k. A uses this key to symmetrically encrypt the secret s. The rationale behind the protocol is that A receives the signature asymmetrically encrypted with her public key and hence she should be the only one able to decrypt its content. Moreover, the digital signature should ensure that B is the originator of the message. The messages sent are illustrated as follows:

$$\begin{array}{lcll} A & \rightarrow & B & : \text{pk(skA)} \\ B & \rightarrow & A & : \text{aenc(sign((pk(skB),k),skB),pk(skA))} \\ A & \rightarrow & B & : \text{senc(s,k)} \end{array}$$

Note that protocol narrations (as above) are useful, but lack clarity. For example, they do not specify any checks which should be made by the participants during the execution of the protocol. Such checks include verifying digital signatures and ensuring that encrypted messages are correctly formed. Failure of these checks typically results in the participant aborting the protocol. These details will be explicitly stated when protocols are encoded for ProVerif. (For further discussion on protocol specification, see [AN96, Aba00].)

Informally, the three properties we would like this protocol to provide are:

1. Secrecy: the value s is known only to A and B.
2. Authentication of A to B: if B reaches the end of the protocol and he believes he has shared the key k with A, then A was indeed his interlocutor and she has shared k.
3. Authentication of B to A: if A reaches the end of the protocol with shared key k, then B proposed k for use by A.

However, the protocol is vulnerable to a *man-in-the-middle* attack (illustrated below). If a dishonest participant I starts a session with B, then I is able to impersonate B in a subsequent session the client A starts with B. At the end of the protocol, A believes that she shares the secret s with B, while she actually shares s with I.

$$\begin{array}{lcll} I & \rightarrow & B & : \text{pk(skI)} \\ B & \rightarrow & I & : \text{aenc(sign((pk(skB),k),skB),pk(skI))} \\ A & \rightarrow & B & : \text{pk(skA)} \\ I & \rightarrow & A & : \text{aenc(sign((pk(skB),k),skB),pk(skA))} \\ A & \rightarrow & B & : \text{senc(s,k)} \end{array}$$

The protocol can easily be corrected by adding the identity of the intended client:

$$\begin{array}{lcll} A & \rightarrow & B & : \text{pk(skA)} \\ B & \rightarrow & A & : \text{aenc(sign((pk(skA),pk(skB),k),skB),pk(skA))} \\ A & \rightarrow & B & : \text{senc(s,k)} \end{array}$$

With this correction, I is not able to re-use the signed key from B in her session with A.

The syntax **channel** c. is a synonym for **free** c: channel. By default, free names are known by the attacker. Free names that are not known by the attacker must be declared *private*:

free $n : t$ [**private**].

Constructors (function symbols) are used to build terms modeling primitives used by cryptographic protocols; for example: one-way hash functions, encryptions, and digital signatures. Constructors are defined by

fun $f(t_1, \ldots, t_n) : t$.

where f is a constructor of arity n, t is its return type, and $t_1, \ldots, t_n$ are the types of its arguments. Constructors are available to the attacker unless they are declared private:

fun $f(t_1, \ldots, t_n) : t$ [**private**].

Private constructors can be useful for modeling tables of keys stored by the server (see Section 6.7.3), for example.

The relationships between cryptographic primitives are captured by destructors which are used to manipulate terms formed by constructors. Destructors are modeled using rewrite rules of the form:

reduc **forall** $x_{1,1} : t_{1,1}, \ldots, x_{1,n_1} : t_{1,n_1};\ g(M_{1,1}, \ldots, M_{1,k}) = M_{1,0}$;
...
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, $M_{1,1}, \ldots, M_{1,k}$ 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. 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 no rule can be applied, the destructor fails, and the process blocks (except for the **let** process, see Section 3.1.4). This behavior corresponds to real world application of cryptographic primitives which include sufficient redundancy to detect scenarios in which an operation fails. For example, in practice, encrypted messages may be assumed to come with sufficient redundancy to discover when the 'wrong' key is used for decryption. It follows that destructors capture the behavior of cryptographic primitives which can visibly fail.

When several variables have the same type, we can avoid repeating their type in the declaration, writing for instance:

reduc **forall** $x, y : t, z : t';\ g(M_1, \ldots, M_k) = M_0$.

Destructors must be deterministic, that is, for each terms $(M_1, \ldots, M_k)$ given as argument to g, when several rewrite rules apply, they must all yield the same result and, in the rewrite rules, the variables that occur in $M_{i,0}$ must also occur in $M_{i,1}, \ldots, M_{i,k}$, so that the result of $g(M_1, \ldots, M_k)$ is entirely determined.

In a similar manner to constructors, destructors may be declared private by appending [**private**]. The generic mechanism by which primitives are encoded permits the modeling of various cryptographic operators.

3.1.2 Example: Declaring cryptographic primitives for the handshake protocol

We now formalize the basic cryptographic primitives used by the handshake protocol.

Symmetric encryption. For symmetric encryption, we define the type key and consider the binary constructor senc which takes arguments of type bitstring, key and returns a bitstring.

```
1 type key.
2
3 fun senc(bitstring, key): bitstring.
```

Note that the type bitstring is built-in, and hence, need not be declared as a user-defined type. The type key is not built-in and hence we declare it on Line 1. To model the decryption operation, we introduce the destructor:

```
4 reduc forall m: bitstring, k: key; sdec(senc(m,k),k) = m.
```

where m represents the message and k represents the symmetric key.

Asymmetric encryption. For asymmetric cryptography, we consider the unary constructor pk, which takes an argument of type skey (private key) and returns a pkey (public key), to capture the notion of constructing a key pair. Decryption is captured in a similar manner to symmetric cryptography with a public/private key pair used in place of a symmetric key.

```
5  type skey.
6  type pkey.
7
8  fun pk(skey): pkey.
9  fun aenc(bitstring, pkey):bitstring.
10
11 reduc forall m: bitstring, k: skey; adec(aenc(m, pk(k)),k) = m.
```

Digital signatures. In a similar manner to asymmetric encryption, digital signatures rely on a pair of signing keys of types sskey (private signing key) and spkey (public signing key). We will consider digital signatures with message recovery:

```
12 type sskey.
13 type spkey.
14
15 fun spk(sskey): spkey.
16 fun sign(bitstring, sskey): bitstring.
17
18 reduc forall m: bitstring, k: sskey; getmess(sign(m,k)) = m.
19 reduc forall m: bitstring, k: sskey; checksign(sign(m,k),spk(k)) = m.
```

The constructors spk, for creating public keys, and sign, for constructing signatures, are standard. The destructors permit message recovery and signature verification. The destructor getmess allows the attacker to get the message m from the signature, even without having the key. The destructor checksign checks the signature, and returns m only when the signature is correct. Honest processes typically use only checksign. This model of signatures assumes that the signature is always accompanied with the message m. It is also possible to model signatures that do not reveal the message m, see Section 4.2.5.

Tuples and typing. For convenience, ProVerif has built-in support for tupling. A tuple of length $n > 1$ is defined as $(M_1, \ldots, M_n)$ where $M_1, \ldots, M_n$ are terms of any type. Once in possession of a tuple, the attacker has the ability to recover the ith element. The inverse is also true: if the attacker is in possession of terms $M_1, \ldots, M_n$, then it can construct the tuple $(M_1, \ldots, M_n)$. Tuples are always of type bitstring. Accordingly, constructors that take arguments of type bitstring may be applied to tuples. Note that the term (M) is not a tuple and is equivalent to M. (Parentheses are needed to override the default precedence of infix operators.) It follows that (M) and M have the same type and that tuples of arity one do not exist.

3.1.3 Process macros

To facilitate development, protocols need not be encoded into a single main process (as we did in Chapter 2). Instead, *sub-processes* may be specified in the declarations using macros of the form

let $R(x_1 : t_1, \ldots, x_n : t_n) = P$.

where R is the macro name, P is the sub-process being defined, and $x_1, \ldots, x_n$, of types $t_1, \ldots, t_n$ respectively, are the free variables of P. The macro expansion $R(M_1, \ldots, M_n)$ will then expand to P with M_1 substituted for x_1, ..., M_n substituted for x_n. As an example, consider a variant `docs/hello_var.pv` of `docs/hello.pv` (previously presented in Chapter 2):

```
free c:channel.

free Cocks:bitstring [private].
free RSA:bitstring [private].
query attacker(Cocks).

let R(x:bitstring) = out(c,x);0.

let R'(y:bitstring)= 0.

process R(RSA) | R'(Cocks)
```

By inspection of ProVerif's output (see Section 3.3 for details on ProVerif's output), one can observe that this process is identical to the one in which the macro definitions are omitted and are instead expanded upon in the main process. It follows immediately that macros are only an encoding which we find particularly useful for development.

3.1.4 Processes

The basic grammar of the language is presented in Figure 3.2; advanced features will be discussed in Chapter 4; and the complete grammar is presented in Appendix A for reference.

Terms M, N consist of names a, b, c, k, m, n, s; variables x, y, z; tuples $(M_1, \ldots, M_j)$ where j is the arity of the tuple; and constructor/destructor application, denoted $h(M_1, \ldots, M_k)$ where k is the arity of h and arguments $M_1, \ldots, M_k$ have the required types. Some functions use the infix notation: $M = N$ for equality, $M <> N$ for disequality (both equality and disequality work modulo an equational theory; they take two arguments of the same type and return a result of type bool), M && M for the boolean conjunction, M || M for the boolean disjunction. We use **not**(M) for the boolean negation. In boolean operations, all values different from true (modulo an equational theory) are considered as false. Furthermore, if the first argument of M && M is not true, then the second argument is not evaluated and the result is false. Similarly, if the first argument of M || M is true, then the second argument is not evaluated and the result is true.

Processes P, Q are defined as follows. The null process 0 does nothing; $P \mid Q$ is the parallel composition of processes P and Q, used to represent participants of a protocol running in parallel; and the replication $!P$ is the infinite composition $P \mid P \mid \ldots$, which is often used to capture an unbounded number of sessions. Name restriction **new** $n : t$; P binds name n of type t inside P, the introduction of restricted names (or private names) is useful to capture both fresh random numbers (modeling nonces and keys, for example) and private channels. Communication is captured by message input and message output. The process **in**$(M, x : t)$; P awaits a message of type t from channel M and then behaves as P with the received message bound to the variable x; that is, every free occurrence of x in P refers to the message received. The process **out**(M, N); P is ready to send N on channel M and then run P. In both of these cases, we may omit P when it is 0. The conditional **if** M **then** P **else** Q is standard: it runs P when the boolean term M evaluates to true, it runs Q when M evaluates to some other value. It executes nothing when the term M fails (for instance, when M contains a destructor for which no rewrite rule applies). For example, **if** $M = N$ **then** P **else** Q tests equality of M and N. For convenience, conditionals may be abbreviated as **if** M **then** P when Q is the null process. The power of destructors can be capitalized upon by **let** $x = M$ **in** P **else** Q statements where M may contain destructors. When this statement is encountered during process execution, there are two possible outcomes. If the term M does not fail (that is, for all destructors in M, matching rewrite rules exist), then x is bound to M and the P branch is taken; otherwise (rather than blocking), the Q branch is taken. (In particular, when M never fails, the P branch will always be executed with x bound to M.) For convenience, the statement **let** $x = M$ **in** P **else** Q may be abbreviated as **let** $x = M$ **in** P when Q is the null process. Finally, we have $R(M_1, \ldots, M_n)$, denoting the use of the macro R with terms $M_1, \ldots, M_n$ as arguments.

Figure 3.2 Term and process grammar

$M, N ::=$	terms
a, b, c, k, m, n, s	names
x, y, z	variables
$(M_1, \ldots, M_k)$	tuple
$h(M_1, \ldots, M_k)$	constructor/destructor application
$M = N$	term equality
$M <> N$	term disequality
M && M	conjunction
$M \mid\mid M$	disjunction
$\mathbf{not}(M)$	negation
$P, Q ::=$	processes
0	null process
$P \mid Q$	parallel composition
$!P$	replication
$\mathbf{new}\ n : t;\ P$	name restriction
$\mathbf{in}(M, x : t);\ P$	message input
$\mathbf{out}(M, N);\ P$	message output
$\mathbf{if}\ M\ \mathbf{then}\ P\ \mathbf{else}\ Q$	conditional
$\mathbf{let}\ x = M\ \mathbf{in}\ P\ \mathbf{else}\ Q$	term evaluation
$R(M_1, \ldots, M_k)$	macro usage

Figure 3.3 Pattern matching grammar

$T ::=$	patterns
$x : t$	typed variable
x	variable without explicit type
$(T_1, ..., T_n)$	tuple
$=M$	equality test

Pattern matching.

For convenience, ProVerif supports pattern matching and we extend the grammar to include patterns (Figure 3.3). The variable pattern $x : t$ matches any term of type t and binds the matched term to x. The variable pattern x is similar, but can be used only when the type of x can be inferred from the context. The tuple pattern $(T_1, \ldots, T_n)$ matches tuples $(M_1, \ldots, M_n)$ where each component M_i ($i \in \{1, \ldots, n\}$) is recursively matched with T_i. Finally, the pattern =M matches terms N where $M = N$. (This is equivalent to an equality test.)

To make use of patterns, the grammar for processes is modified. We omit the rule $\mathbf{in}(M, x : t);\ P$ and instead consider $\mathbf{in}(M, T);\ P$ which awaits a message matching the pattern T and then behaves as P with the free variables of T bound inside P. Similarly, we replace $\mathbf{let}\ x = M\ \mathbf{in}\ P\ \mathbf{else}\ Q$ with the more general $\mathbf{let}\ T = M\ \mathbf{in}\ P\ \mathbf{else}\ Q$. (Note that $\mathbf{let}\ x = M\ \mathbf{in}\ P\ \mathbf{else}\ Q$ is a particular case in which the type of x is inferred from M; users may also write $\mathbf{let}\ x : t = M\ \mathbf{in}\ P\ \mathbf{else}\ Q$ where t is the type of M, ProVerif will produce an error if there is a type mismatch.)

Scope and binding.

Bracketing must be used to avoid ambiguities in the way processes are written down. For example, the process $!P \mid Q$ might be interpreted as $!(P \mid Q)$, or as $(!P) \mid Q$. These processes are different. To avoid too much bracketing, we adopt conventions about the precedence of process operators. The binary parallel process $P \mid Q$ binds most closely; followed by the binary processes $\mathbf{if}\ M\ \mathbf{then}\ P\ \mathbf{else}\ Q$, $\mathbf{let}\ x = M\ \mathbf{in}\ P\ \mathbf{else}\ Q$; finally, unary processes bind least closely. It follows that $!P \mid Q$ is interpreted

as $!(P \mid Q)$. Users should pay particular attention to ProVerif warning messages since these typically arise from misunderstanding ProVerif's binding conventions. For example, consider the process

```
new n:t; out(c,n) | new n:t; in(c,x:t); 0 | if x = n then 0 | out(c,n)
```

which produces the message "Warning: identifier n rebound." Moreover, the process will never perform the final **out**(c,n) because the process is bracketed as follows:

```
new n:t;(out(c,n) | new n:t; (in(c,x:t); 0 | if x = n then (0 | out(c,n))))
```

and hence the final output is guarded by a conditional which can never be satisfied. The authors recommend the distinct naming of names and variables to avoid confusion. New users may like to refer to the output produced by ProVerif to ensure that they have defined processes correctly (see also Section 3.3). Another possible ambiguity arises because of the convention of omitting **else** 0 in the if-then-else construct (and similarly for let-in-else): it is not clear which **if** the **else** applies to in the expression:

if $M = M'$ **then if** $N = N'$ **then** P **else** Q

In this instance, we adopt the convention that the else branch belongs to the closest if and hence the statement should be interpreted as **if** $M = M'$ **then** (**if** $N = N'$ **then** P **else** Q). The convention is similar for let-in-else.

Remarks about syntax

The restrictions on identifiers (Figure 3.2) for constructors/destructors h, names a, b, c, k, m, n, s, types t, and variables x, y, z are completely relaxed. Formally, we do not distinguish between identifiers and let identifiers 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. Note that identifiers are case sensitive. Comments can be included in input files and are surrounded by `(*` and `*)`. Nested comments are not supported.

Reserved words. The following is a list of keywords in the ProVerif language; accordingly, they cannot be used as identifiers.

among, **axiom**, **channel**, **choice**, **clauses**, **const**, **def**, **diff**, **do**, **elimtrue**, **else**, **equation**, **equivalence**, **event**, **expand**, **fail**, **for**, **forall**, **foreach**, **free**, **fun**, **get**, **if**, **implementation**, **in**, **inj-event**, **insert**, **lemma**, **let**, **letfun**, **new**, **noninterf**, **not**, **nounif**, **or**, **otherwise**, **out**, **param**, **phase**, **pred**, **proba**, **process**, **proof**, **public_vars**, **putbegin**, **query**, **reduc**, **restriction**, **secret**, **set**, **suchthat**, **sync**, **table**, **then**, **type**, **weaksecret**, **yield**.

ProVerif also has built-in types bitstring, bool, time and constants true, false of type bool; although these identifiers can be reused as identifiers, the authors strongly discourage this practice.

3.1.5 Example: handshake protocol

We are now ready to present an encoding of the handshake protocol, available in `docs/ex_handshake.pv` (for brevity, we omit function/type declarations and destructors, for details see Section 3.1.1):

```
1 free c:channel.
2
3 free s:bitstring [private].
4 query attacker(s).
5
6 let clientA(pkA:pkey,skA:skey,pkB:spkey) =
7   out(c,pkA);
8   in(c,x:bitstring);
9   let y = adec(x,skA) in
```

```
10    let (=pkB,k:key) = checksign(y,pkB) in
11    out(c,senc(s,k)).
12
13  let serverB(pkB:spkey,skB:sskey) =
14    in(c,pkX:pkey);
15    new k:key;
16    out(c,aenc(sign((pkB,k),skB),pkX));
17    in(c,x:bitstring);
18    let z = sdec(x,k) in
19    0.
20
21  process
22    new skA:skey;
23    new skB:sskey;
24    let pkA = pk(skA) in out(c,pkA);
25    let pkB = spk(skB) in out(c,pkB);
26    ( (!clientA(pkA,skA,pkB)) | (!serverB(pkB,skB)) )
```

The first line declares the public channel c. Lines 3-4 should be familiar from Chapter 2 and further details will be given in Section 3.2. The client process is defined by the macro starting on Line 6 and the server process is defined by the macro starting on Line 13. The main process generates the private asymmetric key skA and the private signing key skB for principals A, B respectively (Lines 22-23). The public key parts pk(skA), spk(skB) are derived and then output on the public communications channel c (Lines 24-25), ensuring that they are available to the attacker. (Observe that this is done using handles pkA, pkB for convenience.) The main process also instantiates multiple copies of the client and server macros with the relevant parameters representing multiple sessions of the roles.

We assume that the server B is willing to run the protocol with any other principal; the choice of her interlocutor will be made by the environment. This is captured by modeling the first input **in**(c,pkX:pkey) to serverB as his client's public key pkX (Line 14). The client A on the other hand only wishes to share his secret s with the server B; accordingly, B's public key is hard-coded into the process clientA. We additionally assume that each principal is willing to engage in an unbounded number of sessions and hence clientA(pkA,skA,pkB) and serverB(pkB,skB) are under replication.

The client and server processes correspond exactly to the description presented in Figure 3.1 and we will now describe the details of our encoding. On request from a client, server B starts the protocol by selecting a fresh key k and outputting aenc(sign((pkB,k),skB),pkX) (Line 16); that is, her signature on the key k paired with her identity spk(skB) and encrypted for his client using her public key pkX. Meanwhile, the client A awaits the input of his interlocutor's signature on the pair (pkB,k) encrypted using his public key (Line 8). A verifies that the ciphertext is correctly formed using the destructor adec on Line 9, which will visibly fail if x is not a message asymmetrically encrypted for the client; that is, the (omitted) else branch of the statement will be evaluated because there is no corresponding rewrite rule. The statement **let** (=pkB,k:key) = checksign(y,pkB) **in** on Line 10 uses destructors and pattern matching with type checking to verify that y is a signature under skB containing a pair, where the first element is the server's public signing key and the second is a symmetric key k. If y is not a correct signature, then the (omitted) else branch of the statement will be evaluated because there is no corresponding rewrite rule, so the client halts. Finally, the server inputs a bitstring x and recovers the cleartext as variable z. (Observe that the failure of decryption is again detectable.) Note that the variable z in the server process is not used.

3.2 Security properties

The ProVerif tool is able to prove reachability properties, correspondence assertions, and observational equivalence. In this section, we will demonstrate how to prove the security properties of the handshake protocol. A more complete coverage of the properties that ProVerif can prove is presented in Section 4.3.

3.2.1 Reachability and secrecy

Proving reachability properties is ProVerif's most basic capability. The tool allows the investigation of which terms are available to an attacker; and hence (syntactic) secrecy of terms can be evaluated with respect to a model. To test secrecy of the term M in the model, the following query is included in the input file before the main process:

query attacker(M).

where M is a ground term, without destructors, containing free names (possibly private and hence not initially known to the attacker). We have already demonstrated the use of secrecy queries on our handshake protocol (see the code in Section 3.1.5).

3.2.2 Correspondence assertions, events, and authentication

Correspondence assertions [WL93] are used to capture relationships between events which can be expressed in the form *"if an event e has been executed, then event e′ has been previously executed."* Moreover, these events may contain arguments, which allow relationships between the arguments of events to be studied. To reason with correspondence assertions, we annotate processes with *events*, which mark important stages reached by the protocol but do not otherwise affect behavior. Accordingly, we extend the grammar for processes to include events denoted

event $e(M_1, \ldots, M_n)$; P

Importantly, the attacker's knowledge is not extended by the terms $M_1, \ldots, M_n$ following the execution of **event** $e(M_1, \ldots, M_n)$; hence, the execution of the process Q after inserting events is the execution of Q without events from the perspective of the attacker. All events must be declared (in the list of declarations in the input file) in the form **event** $e(t_1, \ldots, t_n)$. where $t_1, \ldots, t_n$ are the types of the event arguments. Relationships between events may now be specified as correspondence assertions.

Correspondence

The syntax to query a basic correspondence assertion is:

query $x_1 : t_1, \ldots, x_n : t_n$; **event**$(e(M_1, \ldots, M_j)) \implies$ **event**$(e'(N_1, \ldots, N_k))$.

where $M_1, \ldots, M_j, N_1, \ldots, N_k$ are terms built by the application of constructors to the variables $x_1, \ldots,$ x_n of types $t_1, \ldots, t_n$ and e, e' are declared as events. The query is satisfied if, for each occurrence of the event $e(M_1, \ldots, M_j)$, there is a previous execution of $e'(N_1, \ldots, N_k)$. Moreover, the parameterization of the events must satisfy any relationships defined by $M_1, \ldots, M_j, N_1, \ldots, N_k$; that is, the variables $x_1, \ldots, x_n$ have the same value in $M_1, \ldots, M_j$ and in $N_1, \ldots, N_k$.

In such a query, the variables that occur before the arrow ==> (that is, in $M_1, \ldots, M_j$) are universally quantified, while the variables that occur after the arrow ==> (in $N_1, \ldots, N_k$) but not before are existentially quantified. For instance,

query $x : t_1, y : t_2, z : t_3$; **event**$(e(x, y)) \implies$ **event**$(e'(y, z))$.

means that, for all x, y, for each occurrence of $e(x, y)$, there is a previous occurrence of $e'(y, z)$ for some z.

Injective correspondence

The definition of correspondence we have just discussed is insufficient to capture authentication in cases where a one-to-one relationship between the number of protocol runs performed by each participant is desired. Consider, for example, a financial transaction in which the server requests payment from the client; the server should complete the transaction only once for each transaction started by the client. (If this were not the case, the client could be charged for several transactions, even if the client only started one.) The situation is similar for access control and other scenarios. Injective correspondence assertions capture the one-to-one relationship and are denoted:

query $x_1 : t_1, \ldots, x_n : t_n$; **inj−event**$(e(M_1, \ldots, M_j)) \implies$ **inj−event**$(e'(N_1, \ldots, N_k))$.

Informally, this correspondence asserts that, for each occurrence of the event $e(M_1, \ldots, M_j)$, there is a distinct earlier occurrence of the event $e'(N_1, \ldots, N_k)$. It follows immediately that the number of occurrences of $e'(N_1, \ldots, N_k)$ is greater than, or equal to, the number of occurrences of $e(M_1, \ldots, M_j)$. Note that using **inj−event** or **event** before the arrow ==> does not change the meaning of the query. It is only important after the arrow.

3.2.3 Example: Secrecy and authentication in the handshake protocol

Authentication can be captured using correspondence assertions (additional applications of correspondence assertions were discussed in §1.1). Recall that in addition to the secrecy property mentioned for the handshake protocol in Figure 3.1, there were also authentication properties. The protocol is intended to ensure that, if client A thinks she executes the protocol with server B, then she really does so, and vice versa. When we say 'she thinks' that she executes it with B, we mean that the data she receives indicates that fact. Accordingly, we declare the events:

- **event** acceptsClient(key), which is used by the client to record the belief that she has accepted to run the protocol with the server B and the supplied symmetric key.
- **event** acceptsServer(key,pkey), which is used to record the fact that the server considers he has accepted to run the protocol with a client, with the proposed key supplied as the first argument and the client's public key as the second.
- **event** termClient(key,pkey), which means the client believes she has terminated a protocol run using the symmetric key supplied as the first argument and the client's public key as the second.
- **event** termServer(key), which denotes the server's belief that he has terminated a protocol run with the client A with the symmetric key supplied as the first argument.

Recall that the client is only willing to share her secret with the server B; it follows that, if she completes the protocol, then she believes she has done so with B and hence authentication of B to A should hold. In contrast, server B is willing to run the protocol with any client (that is, he is willing to learn secrets from many clients), and hence at the end of the protocol he only expects authentication of A to B to hold, if he believes A was indeed his interlocutor (so termServer(x) is executed only when pkX = pkA). We can now formalize the two authentication properties (given in Figure 3.1) for the handshake protocol. They are, respectively:

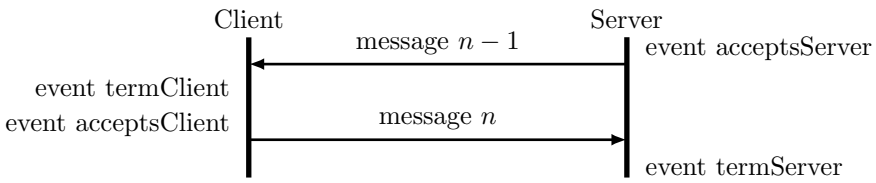
```
query x:key,y:spkey;event(termClient(x,y))==>event(acceptsServer(x,y)).
query x:key;inj-event(termServer(x))==>inj-event(acceptsClient(x)).
```

The subtle difference between the two correspondence assertions is due to the differing authentication properties expected by participants A and B. The first correspondence is not injective because the protocol does not allow the client to learn whether the messages she received are fresh: the message from the server to the client may be replayed, leading to several client sessions for a single server session. The revised ProVerif encoding with annotations and correspondence assertions is presented below and in the file `docs/ex_handshake_annotated.pv` (cryptographic declarations have been omitted for brevity):

```
1  free c:channel.
2
3  free s:bitstring [private].
4  query attacker(s).
5
6  event acceptsClient(key).
7  event acceptsServer(key,pkey).
8  event termClient(key,pkey).
9  event termServer(key).
10
11 query x:key,y:pkey; event(termClient(x,y))==>event(acceptsServer(x,y)).
12 query x:key; inj-event(termServer(x))==>inj-event(acceptsClient(x)).
13
```

```
14 let clientA(pkA:pkey,skA:skey,pkB:spkey) =
15   out(c,pkA);
16   in(c,x:bitstring);
17   let y = adec(x,skA) in
18   let (=pkB,k:key) = checksign(y,pkB) in
19   event acceptsClient(k);
20   out(c,senc(s,k));
21   event termClient(k,pkA).
22
23 let serverB(pkB:spkey,skB:sskey,pkA:pkey) =
24   in(c,pkX:pkey);
25   new k:key;
26   event acceptsServer(k,pkX);
27   out(c,aenc(sign((pkB,k),skB),pkX));
28   in(c,x:bitstring);
29   let z = sdec(x,k) in
30   if pkX = pkA then event termServer(k).
31
32 process
33   new skA:skey;
34   new skB:sskey;
35   let pkA = pk(skA) in out(c,pkA);
36   let pkB = spk(skB) in out(c,pkB);
37   ( (!clientA(pkA,skA,pkB)) | (!serverB(pkB,skB,pkA)) )
```

Figure 3.4 Messages and events for authentication



There is generally some flexibility in the placement of events in a process, but not all choices are correct. For example, in order to prove authentication in our handshake protocol, we consider the property

```
query x:key; inj-event(termServer(x))==>inj-event(acceptsClient(x)).
```

and the event termServer is placed when the server terminates (typically at the end of the protocol), while acceptsClient is placed when the client accepts (typically before the client sends its last message). Therefore, when the last message, message n, is from the client to the server, the placement of events follows Figure 3.4: the last message sent by the client is message n, so acceptsClient is placed before the client sends message n, and termServer is placed after the server receives message n. The last message sent by the server is message $n-1$, so acceptsServer is placed before the server sends message $n-1$, and termClient is placed after the client receives message $n-1$ (any position after that reception is fine). More generally, the event that occurs before the arrow ==> can be placed at the end of the protocol, but the event that occurs after the arrow ==> must be followed by at least one message output. Otherwise, the whole protocol can be executed without executing the latter event, so the correspondence certainly does not hold.

One can also note that moving an event that occurs before the arrow ==> towards the beginning of the protocol strengthens the correspondence property, and moving an event that occurs after the arrow ==> towards the end of the protocol also strengthens the correspondence property. Adding arguments to the events strengthens the correspondence property as well.

3.3 Understanding ProVerif output

The output produced by ProVerif is rather verbatim and can be overwhelming for new users. In essence the output is in the following format:

```
[Equations]
Process:
 [Process]

-- Query [Query]
Completing...
Starting query [Query]
goal [un]reachable: [Goal]
Abbreviations:
 ...

[Attack derivation]

A more detailed output of the traces is available with
  set traceDisplay = long.

[Attack trace]

RESULT [Query] [result].

--------------------------------------------------------------------------
Verification summary:

[Summary of verification results]

--------------------------------------------------------------------------
```

where [Equations] summarizes the internal representation of the equations given in the input file (if any) and [Process] presents the input process with all macros expanded and distinct identifiers assigned to unique names/variables; in addition, parts of the process are annotated with identifiers $\{n\}$ where $n \in \mathbb{N}^*$. (New users may like to refer to this interpreted process to ensure they have defined the scope of variables in the correct manner and to ensure they haven't inadvertently bound processes inside if-then-else/let-in-else statements.) ProVerif then begins to evaluate the [Query] provided by the user. Internally, ProVerif attempts to prove that a state in which a property is violated is unreachable; it follows that ProVerif shows the (un)reachability of some [Goal]. If a property is violated then ProVerif attempts to reconstruct an [Attack derivation] in English and an [Attack trace] in the applied pi calculus. ProVerif then reports whether the query was satisfied. Finally, ProVerif displays a summary of the verification results of all the queries in the file. For convenience, Linux and cygwin users may make use of the following command:

`./proverif` ⟨*filename*⟩`.pv | grep "RES"`

which reduces the output to the results of the queries.

3.3.1 Results

In order to understand the results correctly, it is important to understand the difference between the attack derivation and the attack trace. The attack derivation is an explanation of the actions that the attacker has to make in order to break the security property, in the internal representation of ProVerif. Because this internal representation uses abstractions, the derivation is not always executable in reality; for instance, it may require the repetition of certain actions that can in fact never be repeated, for instance because they are not under a replication. In contrast, the attack trace refers to the semantics of the applied pi calculus, and always corresponds to an executable trace of the considered process.

ProVerif can display three kinds of results:

- RESULT [Query] is true: The query is proved, there is no attack. In this case, ProVerif displays no attack derivation and no attack trace.
- RESULT [Query] is false: The query is false, ProVerif has discovered an attack against the desired security property. The attack trace is displayed just before the result (and an attack derivation is also displayed, but you should focus on the attack trace since it represents the real attack).
- RESULT [Query] cannot be proved: This is a "don't know" answer. ProVerif could not prove that the query is true and also could not find an attack that proves that the query is false. Since the problem of verifying protocols for an unbounded number of sessions is undecidable, this situation is unavoidable. Still, ProVerif gives some additional information that can be useful in order to determine whether the query is true. In particular, ProVerif displays an attack derivation. By manually inspecting the derivation, it is sometimes possible to reconstruct an attack. For observational equivalence properties, it may also display an attack trace, even if this trace does not prove that the observational equivalence does not hold. We will come back to this point when we deal with observational equivalence, in Section 4.3.2. Sources of incompleteness, which explain why ProVerif sometimes fails to prove properties that hold, will be discussed in Section 6.7.5.

Interpreting results. Understanding the internal manner in which ProVerif operates is useful to interpret the results output. Recall that ProVerif attempts to prove that a state in which a property is violated is unreachable. It follows that when ProVerif is supplied with **query attacker**(M)., that internally ProVerif attempts to show **not attacker**(M) and hence RESULT **not attacker**(M) is true. means that the secrecy of M is preserved by the protocol.

Error and warning messages. In case of a syntax error, ProVerif indicates the character position of the error (line and column numbers). Please use your text editor to find the position of the error. (The error messages can be interpreted by `emacs`.) In addition, ProVerif may provide various warning messages. The earlier grep command can be modified into `./proverif` ⟨*filename*⟩`.pv | egrep "RES|Err|War"` for more manageable output with notification of error/warnings, although a more complex command is required to read any associated messages. In this case, the command `./proverif` ⟨*filename*⟩`.pv | less` can be useful.

3.3.2 Example: ProVerif output for the handshake protocol

Executing the handshake protocol with `./proverif docs/ex_handshake_annotated.pv | grep "RES"` produces the following output:

```
RESULT not attacker(s[]) is false.
RESULT event(termClient(x_2,y_1)) ==> event(acceptsServer(x_2,y_1)) is false.
RESULT inj-event(termServer(x_2)) ==> inj-event(acceptsClient(x_2)) is true.
```

which informs us that authentication of A to B holds, but authentication of B to A and secrecy of s do not hold.

Analyzing attack traces.

By inspecting the output more closely, we can reconstruct the attack. For example, let us consider the query **query attacker**(s) which produces the following:

```
1 Process 0 (that is, the initial process):
2 {1}new skA: skey;
3 {2}new skB: sskey;
4 {3}let pkA: pkey = pk(skA) in
5 {4}out(c, pkA);
6 {5}let pkB: spkey = spk(skB) in
7 {6}out(c, pkB);
8 (
9     {7}!
```

```
10      {8}out(c, pkA);
11      {9}in(c, x: bitstring);
12      {10}let y: bitstring = adec(x,skA) in
13      {11}let (=pkB,k: key) = checksign(y,pkB) in
14      {12}event acceptsClient(k);
15      {13}out(c, senc(s,k));
16      {14}event termClient(k,pkA)
17  ) | (
18      {15}!
19      {16}in(c, pkX: pkey);
20      {17}new k_1: key;
21      {18}event acceptsServer(k_1,pkX);
22      {19}out(c, aenc(sign((pkB,k_1),skB),pkX));
23      {20}in(c, x_1: bitstring);
24      {21}let z: bitstring = sdec(x_1,k_1) in
25      {22}if (pkX = pkA) then
26      {23}event termServer(k_1)
27  )
28
29  -- Query not attacker(s[]) in process 0.
30  Completing...
31  Starting query not attacker(s[])
32  goal reachable: attacker(s[])
33
34  Derivation:
35  Abbreviations:
36  k_2 = k_1[pkX = pk(sk),!1 = @sid]
37
38  1. The attacker has some term sk.
39  attacker(sk).
40
41  2. By 1, the attacker may know sk.
42  Using the function pk the attacker may obtain pk(sk).
43  attacker(pk(sk)).
44
45  3. The message pk(sk) that the attacker may have by 2 may be received at
46  input {16}.
47  So the message aenc(sign((spk(skB[]),k_2),skB[]),pk(sk)) may be sent to the
48  attacker at output {19}.
49  attacker(aenc(sign((spk(skB[]),k_2),skB[]),pk(sk))).
50
51  4. By 3, the attacker may know aenc(sign((spk(skB[]),k_2),skB[]),pk(sk)).
52  By 1, the attacker may know sk.
53  Using the function adec the attacker may obtain sign((spk(skB[]),k_2),skB[]).
54  attacker(sign((spk(skB[]),k_2),skB[])).
55
56  5. By 4, the attacker may know sign((spk(skB[]),k_2),skB[]).
57  Using the function getmess the attacker may obtain (spk(skB[]),k_2).
58  attacker((spk(skB[]),k_2)).
59
60  6. By 5, the attacker may know (spk(skB[]),k_2).
61  Using the function 2-proj-2-tuple the attacker may obtain k_2.
62  attacker(k_2).
63
64  7. The message pk(skA[]) may be sent to the attacker at output {4}.
```

```
65 attacker(pk(skA[])).
66
67 8. By 4, the attacker may know sign((spk(skB[]),k_2),skB[]).
68 By 7, the attacker may know pk(skA[]).
69 Using the function aenc the attacker may obtain
70 aenc(sign((spk(skB[]),k_2),skB[]),pk(skA[])).
71 attacker(aenc(sign((spk(skB[]),k_2),skB[]),pk(skA[]))).
72
73 9. The message aenc(sign((spk(skB[]),k_2),skB[]),pk(skA[])) that the attacker
74 may have by 8 may be received at input {9}.
75 So the message senc(s[],k_2) may be sent to the attacker at output {13}.
76 attacker(senc(s[],k_2)).
77
78 10. By 9, the attacker may know senc(s[],k_2).
79 By 6, the attacker may know k_2.
80 Using the function sdec the attacker may obtain s[].
81 attacker(s[]).
82
83 11. By 10, attacker(s[]).
84 The goal is reached, represented in the following fact:
85 attacker(s[]).
86
87
88 A more detailed output of the traces is available with
89   set traceDisplay = long.
90
91 new skA: skey creating skA_1 at {1}
92
93 new skB: sskey creating skB_1 at {2}
94
95 out(c, ~M) with ~M = pk(skA_1) at {4}
96
97 out(c, ~M_1) with ~M_1 = spk(skB_1) at {6}
98
99 out(c, ~M_2) with ~M_2 = pk(skA_1) at {8} in copy a
100
101 in(c, pk(a_1)) at {16} in copy a_2
102
103 new k_1: key creating k_2 at {17} in copy a_2
104
105 event acceptsServer(k_2,pk(a_1)) at {18} in copy a_2
106
107 out(c, ~M_3) with ~M_3 = aenc(sign((spk(skB_1),k_2),skB_1),pk(a_1)) at {19}
108 in copy a_2
109
110 in(c, aenc(adec(~M_3,a_1),~M)) with aenc(adec(~M_3,a_1),~M) =
111 aenc(sign((spk(skB_1),k_2),skB_1),pk(skA_1)) at {9} in copy a
112
113 event acceptsClient(k_2) at {12} in copy a
114
115 out(c, ~M_4) with ~M_4 = senc(s,k_2) at {13} in copy a
116
117 event termClient(k_2,pk(skA_1)) at {14} in copy a
118
119 The attacker has the message
```

```
120 sdec(~M_4,2-proj-2-tuple(getmess(adec(~M_3,a_1)))) = s.
121 A trace has been found.
122 RESULT not attacker(s[]) is false.
123
```

ProVerif first outputs its internal representation of the process under consideration. Then, it handles each query in turn. The output regarding the query **query attacker**(s) can be split into three main parts:

- From "Abbreviations" to "A more detailed...", a description of the derivation that leads to the fact **attacker**(s).
- After "A more detailed..." until "A trace has been found", a description of the corresponding attack trace.
- Finally, the "RESULT" line concludes: the property is false, there is an attack in which the attacker gets s.

Let us first explain the derivation. It starts with a list of abbreviations: these abbreviations give names to some subterms, in order to display them more briefly; such abbreviations are used for the internal representation of names (keys, nonces, ...), which can sometimes be large terms that represent simple atomic data. Next, the description of the derivation itself starts. It is a numbered list of steps, here from 1 to 10. Each step corresponds to one action of the process or of the attacker. After an English description of the step, ProVerif displays the fact that is derived thanks to this step, here **attacker**(M) for some term M, meaning that the attacker has M.

- In step 1, the attacker chooses any value sk in its knowledge (which it is going to use as its secret key).
- In step 2, the attacker uses the knowledge of sk obtained at step 1 ("By 1") to compute the corresponding public key pk(sk) using function pk.
- Step 3 is a step of the process. Input {16} (the numbers between braces refer to program points also written between braces in the description of the process, so input {16} is the input of Line 19) receives the message pk(sk) from the attacker, and output {19} (the one at Line 22) replies with aenc(sign((spk(skB[]),k_2),skB[]),pk(sk)). Note that k_2 is an abbreviation for k_2 = k_1[pkX = pk(sk),!1 = @sid], as listed at the beginning of the derivation. It designates the key k_2 generated by the **new** at Line 20, in session @sid (the number of the copy generated by the replication at Line 18, designated by !1, that is, the first replication), when the key pkX received by the input at Line 19 is pk(sk). ProVerif displays skB[] instead of skB when skB is a name without argument (that is, a free name or a name chosen under no replication and no input). In other words, the attacker starts a session of the server B with its own public key and gets the corresponding message aenc(sign((spk(skB[]),k_2),skB[]),pk(sk)).
- Steps 4 to 6 are again applications of functions by the attacker to perform its internal computations: the attacker decrypts the message aenc(sign((spk(skB[]),k_2),skB[]),pk(sk)) received at step 3 and gets the signed message, so it obtains sign((spk(skB[]),k_2),skB[]) (step 4) and k_2 (step 6).
- Step 7 uses a step of the process: by the output {4} (the one at Line 5), the attacker gets pk(skA[]).
- At step 8, the attacker reencrypts sign((spk(skB[]),k_2),skB[]) with pk(skA[]).
- Step 9 is again a step of the process: the attacker sends aenc(sign((spk(skB[]),k_2),skB[]),pk(skA[])) (obtained at step 8) to input {9} (at Line 11) and gets the reply senc(s [], k_2). In other words, the attacker has obtained a correct message 2 for a session between A and B. It sends this message to A who replies with senc(s [], k_2) as if it was running a session with B.
- In step 10, the attacker decrypts senc(s [], k_2) since it has k_2 (by step 6), so it obtains s[].
- Finally, step 11 indicates that the query goal has been reached, that is, **attacker**(s[]).

As one can notice, this derivation corresponds exactly to the attack against the protocol outlined in Figure 3.1. The display of the derivation can be tuned by some settings: **set** abbreviateDerivation = false prevents the use of abbreviations for names and **set** explainDerivation = false switches to a display of the derivation by explicit references to the Horn clauses used internally by ProVerif instead of relating the derivation to the process. (See also Section 6.6.2 for details on these settings.)

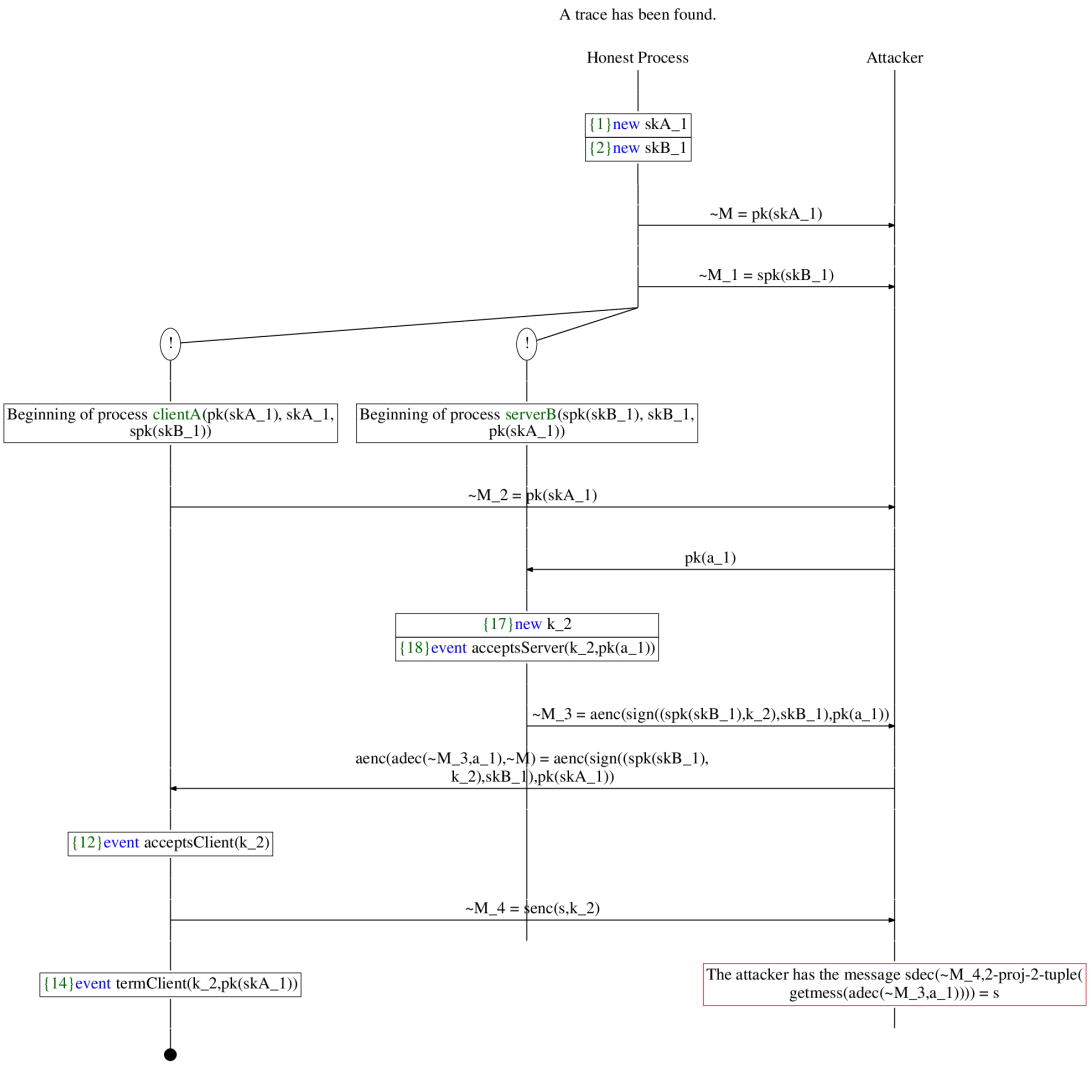
Next, ProVerif reconstructs a trace in the semantics of the pi calculus, corresponding to this derivation. This trace is presented as a sequence of inputs and outputs on public channels and of events. The internal reductions of the process are not displayed for brevity. (As mentioned in the output, it is possible to obtain a more detailed display with the state of the process and the knowledge of the attacker at each step by adding **set** traceDisplay = long. in your input file.) Each input, output, or event is followed by its location in the process "at $\{n\}$", which refers to the program point between braces in the process displayed at the beginning. When the process is under replication, several copies of the process may be generated. Each of these copies is named (by a name like "a_n"), and ProVerif indicates in which copy of the process the input, output, or event is executed. The name itself is unimportant, just the fact that the copy is the same or different is important: the presence of different names of copies for the same replication shows that several sessions are used. Let us explain the trace in the case of the handshake protocol:

- The first two **new** correspond to the creation of secret keys.
- The first two outputs correspond to the outputs of public keys, at outputs {4} (Line 5) and {6} (Line 7). The attacker stores these public keys in fresh variables ˜M and ˜M_1 respectively, so that it can reuse them later.
- The third output is the output of pkA at output {8} (Line 10), in a session of the client A named a.
- The next 4 steps correspond to a session of the server B (copy a_2) with the attacker: the attacker sends its public key pk(a_1) at the input {16} (Line 19). A fresh shared key k_2 is then created. The event acceptsServer is executed (Line 21), and the message aenc(sign((spk(skB_1), k_2), skB_1), pk(a_1)) is sent at output {19} (Line 22) and stored in variable ˜M_3, a fresh variable that can be used later by the attacker. These steps correspond to step 3 of the derivation above.
- The last 4 steps correspond to the end of the execution of the session a of the client A. The attacker computes aenc(adec(˜M_3,a_1),˜M)) and obtains the message aenc(sign((spk(skB_1),k_2),skB_1),pk(skA_1)), which it sends to the input {9} (Line 11). The event acceptsClient is executed (Line 14), the message senc(s,k_2) is sent at output {13} (Line 15) and stored in variable ˜M_4 and finally the event termClient is executed (Line 16). These steps correspond to step 9 of the derivation above.
- Finally, the attacker obtains s[] by computing sdec(˜M_4, 2−proj−2−tuple(getmess(adec(˜M_3, a_1)))).

This trace shows that there is an attack against the secrecy of s, it corresponds to the attack against the protocol outlined in Figure 3.1.

Another way to represent an attack found by ProVerif is by a graph. For instance, the attack explained previously is shown in Figure 3.5. To obtain such a graph, use the command-line option `-graph` or `-html` described in Section 6.6.1. The detailed version is built when **set** traceDisplay = long. has been added to the input `.pv` file. The graph starts always with two processes: the honest one, and the attacker. The progress of the attack is represented vertically. Parallel processes are represented by several columns. Replications of processes are denoted by nodes labeled by !, with a column for each created process. Processes fork when a parallel composition is reduced. The termination of a process is represented by a point. An output on a public channel is represented by a horizontal arrow from the process that makes the output to the attacker. The edge is labeled with an equality $X = M$ where M is the sent message and X is a fresh variable (or tuple of variables) in which the adversary stores it. An input on a public channel is represented by an arrow from the attacker to the receiving process, labeled with an equality $R = M$, where R is the computation performed by the attacker to obtain the sent message M. The

Figure 3.5 Handshake protocol attack trace



message M is omitted when it is exactly equal to R, for instance when R is a constant. A communication made on a private channel is represented by an arrow from the process that outputs the message to the process that receives it; this arrow is labeled with the message. Creation of nonces and other steps are represented in boxes. Information about the attack is written in red; the displayed information depends on the security property that is broken by the attack. The text "a trace has been found" is written at the top of the figure, possibly with assumptions necessary for the attack. When labels are too long to fit on arrows, a table of abbreviations appears at the top right of the figure.

Let us take a closer look at Figure 3.5. First, two new secret keys are created by the honest process. Then the corresponding public keys are sent on a public channel; the attacker receives them and stores them in ˜M and ˜M_1. Next, a parallel reduction is made. We obtain two processes which replicate themselves once each. The first process (clientA) sends its public key on a public channel, and the attacker receives it. Then the attacker sends the message pk(a_1), containing its own public key, to the second process serverB. This process then creates a new shared key k_2 and executes the event acceptsServer(k_2,pk(a_1)). It sends the message aenc(sign((spk(skB_1), k_2), skB_1), pk(a_1)) on a public channel; the attacker receives it and stores it in ˜M_3. The attacker computes aenc(adec(˜M_3,a_1),˜M)), that is, it decrypts and reencrypts the message, thus obtaining

aenc(sign((spk(skB_1),k_2),skB_1),pk(skA_1)). It sends that message to clientA. The process clientA executes the event acceptsClients(k_2) and sends the message senc(s,k_2). The attacker receives it and stores it in ~M_4. Finally, the attacker computes sdec(~M_4,2−proj−2−tuple(getmess(adec(~M_3,a_1)))), and obtains the secret s. This point is mentioned in the red box at the bottom right of the page. The process clientA executes the last event termClient, and terminates. This is the end of the attack. The line numbers of each step appear in green in boxes. The keywords are written in blue, while the names of processes are written in green.

For completeness, we present the complete formalization of the rectified protocol, which ProVerif can successfully verify, below and in the file `docs/ex_handshake_annotated_fixed.pv`.

```
1  (* Symmetric key encryption *)
2
3  type key.
4  fun senc(bitstring, key): bitstring.
5  reduc forall m: bitstring, k: key; sdec(senc(m,k),k) = m.
6
7
8  (* Asymmetric key encryption *)
9
10 type skey.
11 type pkey.
12
13 fun pk(skey): pkey.
14 fun aenc(bitstring, pkey): bitstring.
15
16 reduc forall m: bitstring, sk: skey; adec(aenc(m,pk(sk)),sk) = m.
17
18
19 (* Digital signatures *)
20
21 type sskey.
22 type spkey.
23
24 fun spk(sskey): spkey.
25 fun sign(bitstring, sskey): bitstring.
26
27 reduc forall m: bitstring, ssk: sskey; getmess(sign(m,ssk)) = m.
28 reduc forall m: bitstring, ssk: sskey; checksign(sign(m,ssk),spk(ssk)) = m.
29
30
31 free c:channel.
32
33 free s:bitstring [private].
34 query attacker(s).
35
36 event acceptsClient(key).
37 event acceptsServer(key,pkey).
38 event termClient(key,pkey).
39 event termServer(key).
40
41 query x:key,y:pkey; event(termClient(x,y))==>event(acceptsServer(x,y)).
42 query x:key; inj-event(termServer(x))==>inj-event(acceptsClient(x)).
43
44 let clientA(pkA:pkey,skA:skey,pkB:spkey) =
45   out(c,pkA);
46   in(c,x:bitstring);
```

```
47    let y = adec(x,skA) in
48    let (=pkA,=pkB,k:key) = checksign(y,pkB) in
49    event acceptsClient(k);
50    out(c,senc(s,k));
51    event termClient(k,pkA).
52
53  let serverB(pkB:spkey,skB:sskey,pkA:pkey) =
54    in(c,pkX:pkey);
55    new k:key;
56    event acceptsServer(k,pkX);
57    out(c,aenc(sign((pkX,pkB,k),skB),pkX));
58    in(c,x:bitstring);
59    let z = sdec(x,k) in
60    if pkX = pkA then event termServer(k).
61
62  process
63    new skA:skey;
64    new skB:sskey;
65    let pkA = pk(skA) in out(c,pkA);
66    let pkB = spk(skB) in out(c,pkB);
67    ( (!clientA(pkA,skA,pkB)) | (!serverB(pkB,skB,pkA)) )
```

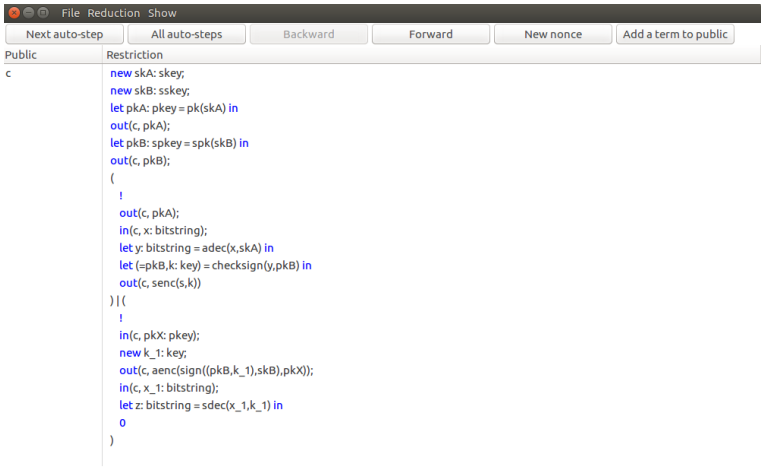
3.4 Interactive mode

As indicated in Section 1.4, ProVerif comes with a program `proverif_interact` which allows to simulate the execution of a process run. There are two ways to launch this program. By typing the name of the program. It then opens a file chooser dialog allowing to choose a `.pv` or `.pcv` file containing the description of the protocol. (`.pcv` files are for CryptoVerif compatibility, see Section 6.8. To choose a `.pcv` file, you first need to change the filter at the bottom right of the file chooser dialog.) The other way is by typing the name of the program, followed by the path of the `.pv` or `.pcv` file. In this case, the simulator starts directly. When the input file is correctly loaded, a window appears, as in Figure 3.6, where the loaded file is the model of the handshake protocol, available in `docs/ex_handshake.pv`.

3.4.1 Interface description

The simulator is made of a main window which allows to make reduction steps on running processes. This window contains several columns representing the current state of the run. The first column, titled "Public", contains all public elements of the current state. For example, after loading the file containing the handshake protocol, the channel c appears in the public column as expected, since c is declared public in the input file (see Figure 3.6). The last columns show processes that are currently running in parallel. To make a reduction step on a specific process, you can click on the head of the column representing the process to reduce. To allow the attacker to create a nonce, there is a button "New nonce", or an option in the "Reduction" menu, or a keyboard shortcut Ctrl+C. If the types are not ignored (by including **set** ignoreTypes = false in your input file, see Section 6.6.2), a dialog box opens and asks the type of the nonce. When a nonce is created, it is added to the public elements of the current state. To go one step backward, there is a button "Backward", or an option in the "Reduction" menu, or a keyboard shortcut Ctrl+B. The button "Forward", the option "Forward" of the "Reduction" menu, or the keyboard shortcut Ctrl+F allow the user to re-execute a step that has been undone by the "Backward" button. The button "Add a term to public " is explained in Section 3.4.5. The interface also allows to display a drawing of the current trace in a new window by clicking on "Display trace" in the "Show" menu, or by hitting Ctrl+D. Each time a new reduction step is made, the drawing is refreshed. The trace can be saved by selecting "Save File" in the "Save" menu, or hitting Ctrl+S. One of these formats: `.png, .pdf, .jpg or .eps`, must be used to save the file, and the name of the file with its extension must be given. Note that a more detailed version of the trace is available if

Figure 3.6 Handshake protocol - Initial simulator window



set traceDisplay = long. has been added to the input file. The main window and the menu also contains two other options: "Next auto-step" and "All auto-steps". We explain this functionality in the next section.

3.4.2 Manual and auto-reduction

There are two kinds of processes. The ones on which the first reduction can be done without the intervention of the user (called auto-reducible processes), and the ones that require the intervention of the user (called manually-reducible processes).

- The processes 0, $P \mid Q$, **new** $n : t; P$, **let** $x = M$ **in** P **else** Q, **if** M **then** P **else** Q, and **event** $e(M_1, \ldots, M_n); P$ are all auto-reducible.
- The process !P is manually reducible.
- The process **out**$(M, N); P$ is auto-reducible if the channel M is public, or the evaluation of the message N or of the channel M fails. Otherwise, it is a manually-reducible process.
- The process **in**$(M, x : T); P$ is auto-reducible if the evaluation of the channel M fails. Otherwise, it is a manually-reducible process.

When auto-reducible processes are running and you press the button "All auto-steps" (or if you select this option on the menu), it reduces all auto-reducible processes that are running. When you press the button "Next auto-step", it makes one step of reduction on the first auto-reducible process. Manually-reducible processes can be reduced only by clicking on the head of their column.

3.4.3 Execution of 0, $P \mid Q$, $!P$, new, let, if, and event

The reduction of 0 just removes the process. The reduction of $P \mid Q$ separates the process $P \mid Q$ into two processes P and Q (a column is added to the main window). The reduction of $!P$ adds a copy of P in a new column at the left of $!P$. The reduction of **new** $n : t; P$ creates a fresh nonce local to the process P. The reduction of **let** $x = M$ **in** P **else** Q evaluates M. If this evaluation succeeds, then the process becomes P with the result of M substituted for x. Otherwise, the process becomes Q. The reduction of **if** M **then** P **else** Q evaluates M. If M evaluates to true, then the process becomes P. If the evaluation of M succeeds and M evaluates to a value other than true, then the process becomes Q. If the evaluation of M fails, then the process is removed. The reduction of **event** $e(M_1, \ldots, M_n); P$ evaluates $M_1, \ldots, M_n$. If these evaluations succeed, the process becomes P. Otherwise, the process is removed. The user can display a column titled "Events", showing the list of executed events by selecting the item "Show/hide events" in the "Show" menu or using the keyboard shortcut Ctrl+E.

3.4.4 Execution of inputs and outputs

They are several possible kinds of inputs and outputs, depending on whether the process is auto-reducible or not, and on whether the channel is public or not. Let us first consider the case of **out**(M, N); P.

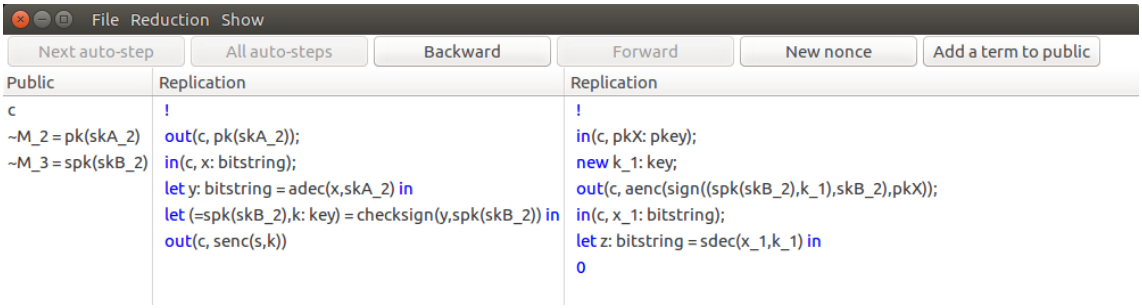
- If the process is auto-reducible because the evaluation of the channel M or of the message N fails, then the process is removed.
- If the evaluations of the message N and the channel M succeed and the channel M is public, then the output is made as explained in Section 3.1.4. The message is added to the public elements of the current state. It is displayed as follows ˜M_i = N, where ˜M_i is a new binder: this binder can then be used to designate the term N in the computations that the adversary makes in the rest of the execution. Such computations are called *recipes*. They are terms built from the binders ˜M_i, the nonces created by the adversary, the names that are initially public, and application of public functions to recipes. In the general case, the public elements of the current state are represented in the form *binder* = *recipe* = *message*, where the recipe is the computation that the adversary makes to obtain the corresponding message, and the binder can be used to designate that message in future recipes. To lighten the display, the binder is omitted when it is equal to the recipe, and the recipe is omitted when it is equal to the message itself.
- If the evaluations of the message N and the channel M succeed but the channel M is not known to be public (this case is displayed "Output (private)" in the head of the column), then there are two possibilities.
  - Prove that the channel is in fact public, and make a public communication. To do so, a recipe using public elements of the current state must be given. If this recipe is evaluated as equal to the channel, a public output on this channel is made.
  - Make a private communication on this channel between two processes. If this choice has been made, the list of all the input processes on the same channel appears in the main window. The user chooses the process that will receive the output message. If there is no such process, the reduction is not possible and an error message appears.

Let us now consider the case of **in**$(M,x : T)$; P.

- If the evaluation of the channel M fails, then the process is removed.
- If the evaluation of the channel M succeeds and the channel is public, then a pop-up window opens, and the user gives the message to send on the channel. The message is given in the form of a recipe, which can contain recipes of public elements of the current state, and applications of public functions. In case the recipe is wrongly typed, if types are ignored (the default), then a warning message box appears, allowing the user to choose to continue or go back. If types are not ignored (the input file contains **set** ignoreTypes = false), an error message box appears, and a new message must be given.
- If the evaluation of the channel M succeeds and the channel is not known to be public (this case is displayed "Input (private)" in the head of the column), then the program works similarly to the case of a private output. There are again two possibilities: prove that the channel is public by giving a recipe and make an input from the adversary, or choose an output process to make a private communication between these processes as explained above.

In addition to the public functions explicitly defined in the input file, recipes can also contain projection functions. The syntax for projections associated to tuples differs depending on whether types are ignored or not. If types are ignored (the default), then the i-th projection of a tuple of arity m is written $i-$proj$-m-$tuple. Otherwise, when the input file contains **set** ignoreTypes = false, $i-$proj$-<type_1>-\ldots-<type_m>-$tuple is the i-th projection of a tuple of arity m, when $<type_n>$ is the type of the n-th argument of the tuple. For instance, 2−proj−channel−bitstring−tuple is the second projection of a pair with arguments of type channel and bitstring, so 2−proj−channel−bitstring−tuple((c, m)) = m, where c is a channel and m is a bitstring. The i-th projection of a previously defined data constructor f (see Section 4.1.2) is written $i-$proj$-f$.

Figure 3.7 Handshake protocol - Simulator window 1



3.4.5 Button "Add a term to public"

Please recall that the elements in public are of the form *binder* = *recipe* = *message* (see Section 3.4.4 for more information on public elements). Clicking the button "Add a term to public" allows the user to add a public term to the current state computed by attacker. The user gives the recipe that the attacker uses to compute this term. It is then evaluated. If the evaluation fails, an error message appears. If the evaluation succeeds, an entry ˜M_i = *recipe* = t is added to the column "Public", where t is the result of the evaluation of the recipe and ˜M_i is a fresh binder associated to it. ˜M_i can then be used in future recipes in order to represent the term t.

3.4.6 Execution of insert and get

You can ignore this section if you do not use tables, defined in Section 4.1.5. The constructs **insert** and **get** respectively insert an element in a table and read a table.

The process **insert** $d(M_1, \ldots, M_n)$; P is auto-reducible if it is the only process or if the evaluation of one of the M_i fails. To insert an element, just click on the head of the column representing the **insert** process to reduce. If the evaluation succeeds, the element is inserted and appears in the column "Tables". Otherwise, the process is removed. The user can display a column titled "Tables", containing all elements of tables obtained by **insert** steps, by selecting the item "Show/hide tables" in the "Show" menu or using the keyboard shortcut Ctrl+T.

The process **get** $d(T_1, \ldots, T_n)$ **suchthat** M **in** P **else** Q is never auto-reducible. To **get** an element from a table, click on the head of the column to reduce. Three cases are possible, depending on the set of terms in the table d that match the patterns $T_1, \ldots T_n$ and satisfy the condition M. First, if there is no such term, then the **else** branch of the **get** is executed. Second, if there is only one such term, then this term is selected, and the **in** branch is executed with the variables of $T_1, \ldots T_n$ instantiated to match this term, as explained in Section 4.1.5. Or third, if there are several such terms, then a window showing all the possible terms is opened. To make the reduction, double-click on the chosen term.

3.4.7 Handshake run in interactive mode

Let us see how to execute a trace similar to the one represented in Figure 3.5 starting from Figure 3.6.

- First, a click on the "All auto-steps" button will lead to the situation represented in Figure 3.7: the honest process first creates two secret keys, then output a first public key after a **let**, and then a second one after another **let** on channel c. The attacker stores these public keys in fresh variables ˜M_2 and ˜M_3. A parallel reduction is then made after that.

- The first process ClientA can now be replicated, by clicking "Replication" at the top of its column. Three processes are obtained. The first process can make an output by clicking on "Next auto-step".

- The process ServerB is then replicated by clicking on the column representing the third process. A click on "New nonce" allows the attacker to create his secret key n, which is added to the public

Figure 3.8 Handshake protocol - Simulator window 2

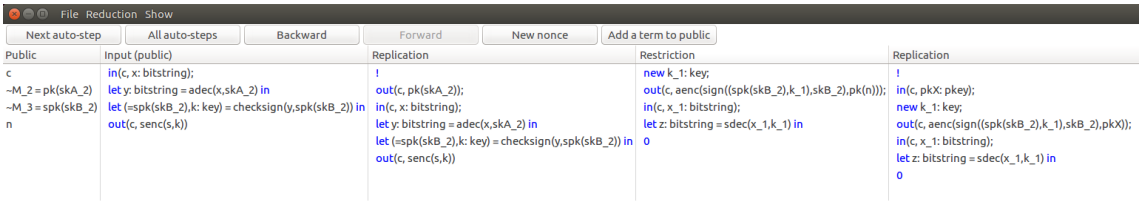
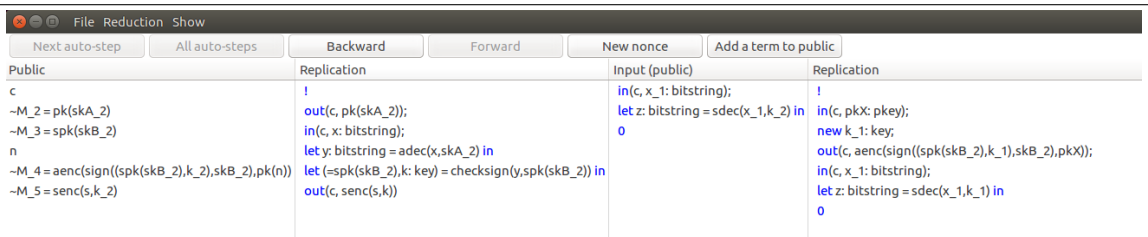


Figure 3.9 Handshake protocol - Simulator window 3



elements of the current state. The message pk(n) can then be input on channel c by clicking on the same column and giving pk(n) as recipe. The result is shown in Figure 3.8.

- A new click on the third process creates a fresh key k_2. Another click sends the message aenc(sign(spk(skB_2), k_2), skB_2, pk(n)), and the attacker stores this message in a fresh variable ˜M_4.
- The message aenc(adec(˜M_4, n),˜M_2) can then be input on channel c, by clicking on the first process and giving aenc(adec(˜M_4, n),˜M_2) as recipe.
- A click on the "All auto-steps" makes all possible reductions on the first process, leading to the output of the message senc(s, k_2) stored by the attacker in a variable ˜M_5. It leads to the window represented in Figure 3.9, and to a trace similar to the one represented in Figure 3.5.
- Finally, by clicking the button "Add a term to public" and giving the recipe sdec(˜M_5, 2−proj−2−tuple(getmess(adec(˜M_4,n)))), the attacker computes this recipe and obtains the secret s. The secret s is then added to the set of public terms.

3.4.8 Advanced features

If the process representing by the input file contains subterms of the form **choice**[L,R] or **diff**[L,R] (see Section 4.3.2), a pop-up window will ask the user to choose either the first or the second component of **choice**, or the biprocess (process with **choice**[L,R]). If the user choses the first or second component, all instances of **choice** inside the process will then be replaced accordingly. Otherwise, the tool runs the processes using the semantics of biprocesses. If the input file is made to test the equivalence between two processes P_1 and P_2 (see Section 4.3.2), a pop-up window will ask the user to choose to emulate either P_1 or P_2.

The processes **let** ... **suchthat** ... (see Section 6.3) and **sync** (see Section 4.1.7) are not supported yet. Passive adversaries (the setting **set attacker** = passive., see Section 6.6.2) and key compromise (the setting **set** keyCompromise = approx. or **set** keyCompromise = strict., see Section 6.6.2) are not supported either. The simulator always simulates an active adversary without key compromise, even if different settings are present.

The command line options `-lib [filename]` (see Section 6.6.1), and `-commandGraph` (used to define the command for the creation of the graph trace from the dot file generated by the simulator) can be used.

Chapter 4

Language features

In the previous chapter, the basic features of the language were introduced; we will now provide a more complete coverage of the language features. These features will be used in Chapter 5 to study the Needham-Schroeder public key protocol as a case study. More advanced features of the language will be discussed in Chapter 6 and the complete input grammar is presented in Appendix A for reference; the features presented in this chapter should be sufficient for most users.

4.1 Primitives and modeling features

In Section 3.1.1, we introduced the basic components of the declarations of the language and how to model processes; this section will develop our earlier presentation.

4.1.1 Constants

A constant may be defined as a function of arity 0, for example "**fun** $c() : t$." ProVerif also provides a specific construct for constants:

```
const c : t.
```

where c is the name of the constant and t is its type. Several constants of the same type t can be declared by

```
const c_1, ..., c_k : t.
```

4.1.2 Data constructors and type conversion

Constructors **fun** $f(t_1, \ldots, t_n) : t.$ may be declared as items of data by appending [**data**], that is,

```
fun f(t_1, ..., t_n) : t [data].
```

A constructor declared as data is similar to a tuple: the attacker can construct and decompose data constructors. In other words, declaring a data constructor f as above implicitly declares n destructors that map $f(x_1, \ldots, x_n)$ to x_i, where $i \in \{1, \ldots, n\}$. One can inverse a data constructor by pattern-matching: the pattern $f(T_1, \ldots, T_n)$ is added as pattern in the grammar of Figure 3.3. The type of $T_1, \ldots, T_n$ is the type of the arguments of f, so when T_i is a variable, its type can be omitted. For example, with the declarations

```
type key.
type host.
fun keyhost(key, host): bitstring [data].
```

we can write

```
let keyhost(k,h) = x in ...
```

Constructors declared **data** cannot be declared **private**.

One application of data constructors is type conversion. As discussed in Section 3.1.1, the type system occasionally makes it difficult to apply functions to arguments due to type mismatches. This can be overcome with type conversion. A type converter is simply a special type of data constructor defined as follows:

fun $tc(t) : t'$ [**typeConverter**].

where the type converter tc takes input of type t and returns a result of type t'. Observe that, since the constructor is a data constructor, the attacker may recover term M from the term $tc(M)$. Intuitively, the keyword **typeConverter** means that the function is the identity function, and so has no effect except changing the type. By default, types are used for typechecking the protocol but during protocol verification, ProVerif ignores types. The **typeConverter** functions are thus removed. (This behavior allows ProVerif to detect type flaw attacks, in which the attacker mixes data of different types. This behavior can be changed by the setting **set** ignoreTypes = ... as discussed in Section 6.6.2.)

The reverse type conversion, from t' to t, should be performed by pattern-matching:

let $tc(x) = M$ **in** ...

where M is of type t' and x is of type t. This construct is allowed since type converters are data constructors. When one defines a type converter $tc(t) : t'$ from type t to t', all elements of type t can be converted to type t', but the only elements of type t' that can be converted to type t are the elements of the form $tc(M)$. Hence, for instance, it is reasonable to define a type converter from a type key representing 128-bit keys to type bitstring, but not in the other direction, since all 128-bit keys are bitstrings but only some bitstrings are 128-bit keys.

4.1.3 Natural numbers

Natural numbers are natively supported and have the built-in type nat. Internally, ProVerif models natural numbers following the Peano axioms, that is, it considers a constant 0 of type nat and a data constructor for successor. As such, all natural numbers are terms and can be used with other user-defined functions. A term is said to be *a natural number* if it is the constant 0 or the application of the successor to a natural number. The grammar of terms (Figure 3.2) is extended in Figure 4.1 to consider the built-in infix functions manipulating natural numbers.

Figure 4.1 Natural number grammar

$M, N ::=$	terms
...	
i	natural number ($i \in \mathbb{N}$)
$M + i$	addition ($i \in \mathbb{N}$)
$i + M$	addition ($i \in \mathbb{N}$)
$M - i$	subtraction ($i \in \mathbb{N}$)
$M > N$	greater
$M < N$	smaller
$M >= N$	greater or equal
$M <= N$	smaller or equal

Finally, ProVerif has a built-in boolean function is_nat checking whether a term is a natural number of not, that is, is_nat(M) returns true if and only if M is equal modulo the equational theory to a natural number.

Note that addition between two arbitrary terms is not allowed. The order relations $>, <, >=, <=$ are internally represented by boolean destructor functions that compare the value of two natural numbers. As such, M > N returns true (resp. false) if M and N are both natural numbers and M is strictly greater than (resp. smaller or equal to) N. Note that M > N fails if M or N is not a natural number. Similarly, the subtraction is internally represented by a destructor function and for instance, $M - i$ fails if M is a natural number strictly smaller than i. It corresponds to the fact that negative numbers are not allowed in ProVerif.

Restrictions. Since natural numbers are represented with a constant 0 and a data constructor successor, the attacker can generate all natural numbers. Therefore, ProVerif does not allow the declaration of new names with the type nat, i.e., **new** k:nat, since it would allow a process to generate a term declared as a natural number but that does not satisfy the Peano axioms. Similarly, user defined constructors cannot have nat as their return type. However, this restriction does not apply to destructors. Finally, all functions can have nat as argument type. For example, the following declarations and process are allowed.

```
type key.

free c:channel.

free s:bitstring [private].

fun ienc(nat,key):bitstring.
fun idec(bitstring,key):nat
reduc forall x:nat, y:key; idec(ienc(x+1,y),y) = x.

query attacker(s).

process
  new k:key; (
    out(c,ienc(2,k))
    | in(c,x:nat); in(c,y:bitstring); if x + 3 > idec(y,k) then out(c,s)
  )
```

The function idec is allowed to have nat as return type as it is declared as a destructor. In this example, the query is false since the attacker can obtain s by inputting any natural number for x. Note that the test **if** x + 3 > idec(y,k) **then** ... is not equivalent to **if** x > idec(y,k) − 3 **then** .... Indeed, in the latter, ProVerif first evaluates the terms x and idec(y,k) − 3 before comparing their values. In our example, idec(y,k) − 3 will always fail since the only case where the evaluation of idec(y,k) would not fail is when y is equal to ienc(2,k). In such a case, idec(y,k) would be evaluated to 1 but then the evaluation of 1 − 3 would fail. Hence, the query **attacker**(s) is true for the following process:

```
process
  new k:key; (
    out(c,ienc(2,k))
    | in(c,x:nat); in(c,y:bitstring); if x > idec(y,k) - 3 then out(c,s)
  )
```

4.1.4 Enriched terms

For greater flexibility, we redefine our grammar for terms (Figures 3.2 and 4.1) to include restrictions, conditionals, and term evaluations as presented in Figure 4.2. The behavior of enriched terms will now be discussed. Names, variables, tuples, and constructor/destructor application are defined as standard. The term **new** $a : t$; M constructs a new name a of type t and then evaluates the enriched term M. The term **if** M **then** N **else** N' is defined as N if the condition M is equal to true and N' when M does not fail but is not equal to true. If M fails, or the else branch is omitted and M is not equal to true, then the term **if** M **then** N **else** N' fails (like when no rewrite rule matches in the evaluation of a destructor). Similarly, **let** $T = M$ **in** N **else** N' is defined as N if the pattern T is matched by M, and the variables of T are bound by this pattern-matching. As before, if the pattern is not matched, then the enriched term is defined as N'; and when the else branch is omitted, the term fails. The term **event** $e(M_1, \ldots, M_n)$; M executes the event $e(M_1, \ldots, M_n)$ and then evaluates the enriched term M. The use of enriched terms will be demonstrated in the Needham-Schroeder case study in Section 5.3.

ProVerif's internal encoding for *enriched terms*. Enriched terms are a convenient tool for the end user; internally, ProVerif handles such constructs by encoding them: the conditional **if** M **then** N **else** N'

Figure 4.2 Enriched terms grammar

$M, N ::=$	enriched terms
a, b, c, k, m, n, s	names
x, y, z	variables
$(M_1, \ldots, M_j)$	tuple
$h(M_1, \ldots, M_j)$	constructor/destructor application
i	natural number ($i \in \mathbb{N}$)
$M + i$	addition ($i \in \mathbb{N}$)
$i + M$	addition ($i \in \mathbb{N}$)
$M - i$	subtraction ($i \in \mathbb{N}$)
$M > N$	greater
$M < N$	smaller
$M >= N$	greater or equal
$M <= N$	smaller or equal
$M = N$	term equality
$M <> N$	term disequality
M && M	conjunction
M || M	disjunction
not(M)	negation
new $a : t$; M	name restriction
if M **then** N **else** N'	conditional
let $T = M$ **in** N **else** N'	term evaluation
event $e(M_1, \ldots, M_n)$; M	event

is encoded as a special destructor also displayed as **if** M **then** N **else** N'; the restriction **new** $a : t$; M is expanded into a process; the term evaluation **let** $T = M$ **in** N **else** N' is encoded as a mix of processes and special destructors. As an example, let us consider the following process.

```
free c:channel.

free A:bitstring.
free B:bitstring.

process
  in(c, (x:bitstring,y:bitstring));
  if x = A || x = B then
  let z = (if y = A then new n:bitstring; (x,n) else (x,y)) in
  out(c,z)
```

The process takes as input a pair of bitstrings x,y and checks that either x=A or x=B. The term evaluation **let** z = (**if** y = A **then new** n:bitstring; (x,n) **else** (x,y)) **in** is defined using the enriched term **if** y = A **then new** n:bitstring; (x,n) **else** (x,y) which evaluates to the tuple (x,n) where n is a new name of type bitstring if y=A; or (x,y) otherwise. (Note that brackets have only been added for readability.) Internally, ProVerif encodes the above main process as:

```
in(c, (x: bitstring,y: bitstring));
if ((x = A) || (x = B)) then
new n: bitstring;
let z: bitstring = (if (y = A) then (x,n) else (x,y)) in
out(c, z)
```

This encoding sometimes has visible consequences on the behavior of ProVerif. Note that this process was obtained by beautifying the output produced by ProVerif (see Section 3.3 for details on ProVerif output).

4.1.5 Tables and key distribution

ProVerif provides tables (or databases) for persistent storage. Tables must be specified in the declarations in the following form:

table $d(t_1, \ldots, t_n)$.

where d is the name of the table which takes records of type $t_1, \ldots, t_n$. Processes may populate and access tables, but deletion is forbidden. Note that tables are not accessible by the attacker. Accordingly, the grammar for processes is extended:

insert $d(M_1, \ldots, M_n)$; P	insert record
get $d(T_1, \ldots, T_n)$ **in** P **else** Q	read record
get $d(T_1, \ldots, T_n)$ **suchthat** M **in** P **else** Q	read record

The process **insert** $d(M_1, \ldots, M_n)$; P inserts the record $M_1, \ldots, M_n$ into the table d and then executes P; when P is the 0 process, it may be omitted. The process **get** $d(T_1, \ldots, T_n)$ **in** P **else** Q attempts to retrieve a record in accordance with patterns $T_1, \ldots, T_n$. When several records can be matched, one possibility is chosen (but ProVerif considers all possibilities when reasoning) and the process P is evaluated with the free variables of $T_1, \ldots, T_n$ bound inside P. When no such record is found, the process Q is executed. The else branch can be omitted; in this case, when no suitable record is found, the process blocks. The **get** process also has a richer form **get** $d(T_1, \ldots, T_n)$ **suchthat** M **in** P **else** Q; in this case, the retrieved record is required to satisfy the condition M in addition to matching the patterns $T_1, \ldots, T_n$. The grammar for enriched terms is extended similarly:

insert $d(M_1, \ldots, M_n)$; M	insert record
get $d(T_1, \ldots, T_n)$ **in** N **else** N'	read record
get $d(T_1, \ldots, T_n)$ **suchthat** M **in** N **else** N'	read record

When the **else** branch of **get** is omitted in an enriched term, it equivalent to **else fail**.

The use of tables for key management will be demonstrated in the Needham-Schroeder public key protocol case study (Chapter 5).

As a side remark, tables can be encoded using private channels. We provide a specific construct since it is frequently used, it can be analyzed precisely by ProVerif (more precisely than some other uses of private channels), and it is probably easier to understand for users that are not used to the pi calculus.

4.1.6 Phases

Many protocols can be broken into phases, and their security properties can be formulated in terms of these phases. Typically, for instance, if a protocol discloses a session key after the conclusion of a session, then the secrecy of the data exchanged during that session may be compromised but not its authenticity. To enable modeling of protocols with several phases the syntax for processes is supplemented with a phase prefix **phase** t; P, where t is a positive integer. Observe that all processes are under phase 0 by default and hence the instruction **phase** 0 is not allowed. Intuitively, t represents a global clock, and the process **phase** t; P is active only during phase t. A process with phases is executed as follows. First, all instructions under phase 0 are executed, that is, all instructions not under phase $i \geq 1$. Then, during a stage transition from phase 0 to phase 1, all processes which have not yet reached phase $i \geq 1$ are discarded and the process may then execute instructions under phase 1, but not under phase $i \geq 2$. More generally, when changing from phase n to phase $n+1$, all processes which have not reached a phase $i \geq n+1$ are discarded and instructions under phase $n+1$, but not for phase $i \geq n+2$, are executed. It follows from our description that it is not necessary for all instructions of a particular phase to be executed prior to phase transition. Moreover, processes may communicate only if they are under the same phase.

Phases can be used, for example, to prove forward secrecy properties: the goal is to show that, even if some participants get corrupted (so their secret keys are leaked to the attacker), the secrets exchanged in sessions that took place before the corruption are preserved. Corruption can be modeled in ProVerif by outputting the secret keys of the corrupted participants in phase 1; the secrets of the sessions run in phase 0 should be preserved. This is done for the fixed handshake protocol of the previous chapter in the following example (file `docs/ex_handshake_forward_secrecy_skB.pv`):

```
free c:channel.

free s:bitstring [private].
query attacker(s).

let clientA(pkA:pkey,skA:skey,pkB:spkey) =
  out(c,pkA);
  in(c,x:bitstring);
  let y = adec(x,skA) in
  let (=pkA,=pkB,k:key) = checksign(y,pkB) in
  out(c,senc(s,k)).

let serverB(pkB:spkey,skB:sskey,pkA:pkey) =
  in(c,pkX:pkey);
  new k:key;
  out(c,aenc(sign((pkX,pkB,k),skB),pkX));
  in(c,x:bitstring);
  let z = sdec(x,k).

process
  new skA:skey;
  new skB:sskey;
  let pkA = pk(skA) in out(c,pkA);
  let pkB = spk(skB) in out(c,pkB);
  ( (!clientA(pkA,skA,pkB)) | (!serverB(pkB,skB,pkA)) |
  phase 1; out(c, skB) )
```

The secret key skB of the server B is leaked in phase 1 (last line). The secrecy of s is still preserved in this example: the attacker can impersonate B in phase 1, but cannot decrypt messages of sessions run in phase 0. (Note that one could hope for a stronger model: this model does not consider sessions that are running precisely when the key is leaked. While the attacker can simulate B in phase 1, the model above does not run A in phase 1; one could easily add a model of A in phase 1 if desired.) In contrast, if the secret key of the client A is leaked, then the secrecy of s is not preserved: the attacker can decrypt the messages of previous sessions by using skA, and thus obtain s.

4.1.7 Synchronization

The synchronization command **sync** t [*tag*] introduces a global synchronization [BS16], which has some similarity with phases.

The synchronization level t must be a positive integer. Synchronizations **sync** t cannot occur under replications. Synchronizations with the same level t and the same tag *tag* are considered as the "same synchronization", that is, synchronizations with the same level t and the same tag *tag* are allowed only in different branches of **if**, **let**, **let** ... **suchthat**, **get**. Since only one of these branches will be executed at runtime, at most one synchronization with a given level t and tag *tag* can be reached.

The global synchronizations must be executed in increasing order of level t. The process waits until **sync** t commands with all existing tags at level t are reached before executing the synchronization t. More precisely, assuming t is the smallest synchronization level that occurs in the initial process and has not been executed yet, if the initial process contains commands **sync** t with tags $tag_1, \ldots, tag_n$, then the process waits until it reaches exactly commands **sync** t with tags $tag_1, \ldots, tag_n$, then it executes the synchronization t and continues after the **sync** t commands. So, in contrast to phases, processes are never discarded by synchronization, but the process may block in case some synchronizations cannot be reached or are discarded for instance by a test that fails above them.

The tags of synchronizations are determined as follows:

- The user can specify the tag of the synchronization by writing **sync** t [*tag*]. When the user omits the tag and just writes **sync** t, ProVerif gives it a fresh tag.

- When a synchronization occurs inside a process macro and the process macro is expanded, a tag prefix is added to all synchronizations inside the process macro. The prefix p is specified by writing [**sync**: tag prefix p] at the expansion of the process macro. For instance:

```
let P(x:bitstring)=
   sync 1 [T];
   out(c,x).

process
    P(a) [sync: tag prefix T1] | P(b) [sync:tag prefix T2]
```

  yields the process

```
sync 1 [T1_T]; out(c,a) | sync 1 [T2_T]; out(c,b)
```

  (The prefix is separated from the tag by an underscore.) When the indication [**sync**: tag prefix p] is omitted, ProVerif chooses a fresh prefix. One can tell ProVerif not to add a prefix, that is, leave the tags of synchronizations unchanged, by writing [**sync**: no tag prefix] instead of [**sync**: tag prefix p].

Therefore, when all tags of synchronizations and tag prefixes of process macros are omitted, all synchronizations in the resulting process have distinct tags. This is suitable when these synchronizations occur in parallel processes.

When synchronizations occur in branches of tests, one typically wants them to have the same tag (because otherwise the synchronization would block). So one would write for instance

```
if ... then (... sync 1 [T]; ...) else (... sync 1 [T]; ...)
```

or

```
if ... then (... P(...) [sync: tag prefix T])
      else (... P(...) [sync: tag prefix T])
```

Synchronizations cannot be used with phases. Synchronizations are implemented in ProVerif by translating them into outputs and inputs; the translated process is displayed by ProVerif. Further discussion of synchronization with an example can be found in Section 4.3.2, page 60.

4.2 Further cryptographic operators

In Section 3.1.1, we introduced how to model the relationships between cryptographic operations and in Section 3.1.2 we considered the formalization of basic cryptographic primitives needed to model the handshake protocol. This section will consider more advanced formalisms and provide a small library of cryptographic primitives.

4.2.1 Extended destructors

We introduce an extended way to define the behaviour of destructors [CB13].

fun $g(t_1, \ldots, t_k) : t$
reduc **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** $x_{m,1} : t_{m,1}, \ldots, x_{m,n_m} : t_{m,n_m}; g(M_{m,1}, \ldots, M_{m,k}) = M_{m,0}$.

This declaration should be seen as a sequence of rewrite rules rather than as 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, **forall** $x_{1,1} : t_{1,1}, \ldots, x_{1,n_1} : t_{1,n_1};\ g(M_{1,1}, \ldots, M_{1,k}) = M_{1,0}$. If this rewrite rule is applicable, then the term $g(N_1, \ldots, N_n)$ is reduced according to that rewrite rule. Otherwise, ProVerif tries 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 with the definition of Section 3.1.1.

```
fun eq(bitstring,bitstring):bool
    reduc forall x: bitstring; eq(x,x) = true
    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.

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 **fail**.

```
fun test(bool,bitstring,bitstring):bitstring
reduc
  forall x: bitstring, y:bitstring; test(true,x,y) = x
  otherwise forall c: bool, x: bitstring, y:bitstring; test(c,x,y) = y
  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 fail**. It indicates that x can be any message or even the special value **fail**. Relying on this new declaration of variables, the destructor test could have been defined as follows:

```
fun test(bool,bitstring,bitstring):bitstring
reduc
  forall x: bitstring, y:bitstring; test(true,x,y) = x
  otherwise forall c: bool or fail, x: bitstring, y:bitstring;
    test(c,x,y) = y.
```

A variant of this test destructor is the following one:

```
fun test'(bool,bitstring,bitstring):bitstring
reduc
  forall x: bitstring or fail, y:bitstring or fail ; test'(true,x,y) = x
  otherwise forall c: bool, x: bitstring or fail, y:bitstring or fail;
    test'(c,x,y) = y.
```

This destructor returns its second argument when the first argument c is true, its third argument when the first argument c does not fail but is not true, and fails otherwise. With this definition, when the first argument is true, test' returns the second argument even when the third argument fails (which models that the third argument does not need to be evaluated in this case). Symmetrically, when the first argument does not fail but is not true, test' returns the third argument even when the second argument fails. In contrast, the previous destructor test fails when its second or third arguments fail.

It is also possible to transform the special failure value **fail** into a non-failure value c0 by a destructor:

```
const c0: bitstring.
fun catchfail(bitstring): bitstring
reduc
  forall x:bitstring; catchfail(x) = x
  otherwise catchfail(fail) = c0.
```

Such a destructor is used internally by ProVerif.

4.2.2 Equations

Certain cryptographic primitives, such as the Diffie-Hellman key agreement, cannot be encoded as destructors, because they require algebraic relations between terms. Accordingly, ProVerif provides an alternative model for cryptographic primitives, namely equations. The relationships between constructors are captured using equations of the form

equation forall $x_1 : t_1, \ldots, x_n : t_n$; $M = N$.

where M, N are terms built from the application of (defined) constructor symbols to the variables $x_1, \ldots, x_n$ of type $t_1, \ldots, t_n$. Note that when no variables are required (that is, when terms M, N are constants) **forall** $x_1 : t_1, \ldots, x_n : t_n$; may be omitted.

More generally, one can declare several equations at once, as follows:

equation **forall** $x_{1,1} : t_{1,1}, \ldots, x_{1,n_1} : t_{1,n_1}$; $M_1 = N_1$;
...
forall $x_{m,1} : t_{m,1}, \ldots, x_{m,n_m} : t_{m,n_m}$; $M_m = N_m$ *option* .

where *option* can either be empty, [convergent], or [linear]. When an option [convergent] or [linear] is present, it means that the group of equations is convergent (the equations, oriented from left to right, form a convergent rewrite system) or linear (each variable occurs at most once in the left-hand and once in the right-hand side of each equation), respectively. In this case, this group of equations must use function symbols that appear in no other equation. ProVerif checks that the convergent or linear option is correct. However, in case ProVerif cannot prove termination of the rewrite system associated to equations declared [convergent], it just displays a warning, and continues assuming that the rewrite system terminates. Indeed, ProVerif's algorithm for proving termination is obviously not complete, so the rewrite system may terminate and ProVerif not be able to prove it. The main interest of the [convergent] option is then to bypass the verification of termination of the rewrite system.

Performance. It should be noted that destructors are more efficient than equations. The use of destructors is therefore advocated where possible.

Limitations. ProVerif does not support all equations. It must be possible to split the set of equations into two kinds of equations that do not share constructor symbols: convergent equations and linear equations. Convergent equations are equations that, when oriented from left to right, form a convergent (that is, terminating and confluent) rewriting system. Linear equations are equations such that each variable occurs at most once in the left-hand side and at most once in the right-hand side. When ProVerif cannot split the equations into convergent equations and linear equations, an error message is displayed.

Moreover, even when the equations can be split as above, it may happen that the pre-treatment of equations by ProVerif does not terminate. Essentially, ProVerif computes rewrite rules that encode the equations and it requires that, when $M_1, \ldots, M_n$ are in normal form, the normal form of $f(M_1, \ldots, M_n)$ can be computed by a single rewrite step. For some equations, this constraint implies generating an infinite number of rewrite rules, so in this case ProVerif does not terminate. For instance, associativity cannot be handled by ProVerif for this reason, which prevents the modeling of primitives such as XOR (exclusive or) or groups. Another example that leads to non-termination for the same reason is the equation $f(g(x)) = g(f(x))$. In the obtained rewrite rules, all variables that occur in the right-hand side must also occur in the left-hand side.

It is also worth noting that, because ProVerif orients equations from left to right when it builds the rewrite system, the orientation in which the equations are written may influence the success or failure of ProVerif (even if the semantics of the equation obviously does not depend on the orientation). Informally, the equations should be written with the most complex term on the left and the simplest one on the right.

Even with these limitations, many practical primitives can be modeled by equations in ProVerif, as illustrated below.

Diffie-Hellman key agreement. The Diffie-Hellman key agreement relies on modular exponentiation in a cyclic group G of prime order q; let g be a generator of G. A principal A chooses a random exponent a in $\mathbb{Z}_q^*$, and sends g^a to B. Similarly, B chooses a random exponent b, and sends g^b to A. Then A computes $(g^b)^a$ and B computes $(g^a)^b$. These two keys are equal, since $(g^b)^a = (g^a)^b$, and cannot be obtained by a passive attacker who has g^a and g^b but neither a nor b.

We model the Diffie-Hellman key agreement as follows:

```
1 type G.
2 type exponent.
```

```
const g: G [data].
fun exp(G, exponent): G.

equation forall x: exponent, y: exponent; exp(exp(g,x),y) = exp(exp(g,y),x).
```

The elements of G have type G, the exponents have type exponent, g is the generator g, and exp models modular exponentiation exp(x,y) = x^y. The equation means that $(g^x)^y = (g^y)^x$.

This model of Diffie-Hellman key agreement is limited in that it just takes into account the equation needed for the protocol to work, while there exist other equations, coming from the multiplicative group $\mathbb{Z}_q^*$. A more complete model is out of scope of the current treatment of equations in ProVerif, because it requires an associative function symbol, but extensions have been proposed to handle it [KT09].

Symmetric encryption. We model a symmetric encryption scheme for which one cannot distinguish whether decryption succeeds or not. We consider the binary constructors senc and sdec, the arguments of which are of types bitstring and key.

```
type key.

fun senc(bitstring, key): bitstring.
fun sdec(bitstring, key): bitstring.
```

To model the properties of decryption, we introduce the equations:

```
equation forall m: bitstring, k: key; sdec(senc(m,k),k) = m.
equation forall m: bitstring, k: key; senc(sdec(m,k),k) = m.
```

where k represents the symmetric key and m represents the message. The first equation is standard: it expresses that, by decrypting the ciphertext with the correct key, one gets the cleartext. The second equation might seem more surprising. It implies that encryption and decryption are two inverse bijections; it is satisfied by block ciphers, for instance. One can also note that this equation is necessary to make sure that one cannot distinguish whether decryption succeeds or not: without this equation, sdec(M,k) succeeds if and only if senc(sdec(M,k),k) = M.

Trapdoor commitments. As a more involved example, let us consider trapdoor commitments [DDKS17]. Trapdoor commitments are commitments that can be opened to a different value than the one initially committed, using a trapdoor. We represent a trapdoor commitment of message m with randomness r and trapdoor td by $tdcommit(m, r, td)$. The normal opening of the commitment returns the message m, so we have the equation

$$open(tdcommit(m, r, td), r) = m$$

To change the message, we use the equation:

$$tdcommit(m_2, f(m_1, r, td, m_2), td) = tdcommit(m_1, r, td)$$

These equations, oriented from left to right, are not convergent. We need to complete them to obtain a convergent system, with the following equations:

$$open(tdcommit(m_1, r, td), f(m_1, r, td, m_2)) = m_2$$
$$f(m_1, f(m, r, td, m_1), td, m_2) = f(m, r, td, m_2)$$

These equations are convergent, but ProVerif is unable to show termination, so it fails to handle the equations if they are given separately. We can bypass the termination check by giving the equations together and indicating that they are convergent, as follows:

```
type trapdoor.
type rand.

fun tdcommit(bitstring, rand, trapdoor): bitstring.
```

```
fun open(bitstring, rand): bitstring.
fun f(bitstring, rand, trapdoor, bitstring): rand.

equation forall m: bitstring, r: rand, td: trapdoor;
   open(tdcommit(m,r,td),r) = m;
   forall m1: bitstring, m2: bitstring, r: rand, td: trapdoor;
   tdcommit(m2,f(m1,r,td,m2),td) = tdcommit(m1,r,td);
   forall m1: bitstring, m2: bitstring, r: rand, td: trapdoor;
   open(tdcommit(m1,r,td),f(m1,r,td,m2)) = m2;
   forall m: bitstring, m1: bitstring, m2: bitstring, r: rand, td: trapdoor;
   f(m1, f(m,r,td,m1), td, m2) = f(m, r, td, m2) [convergent].
```

ProVerif still displays a warning because it cannot prove that the equations terminate:

```
Warning: the following equations
open(tdcommit(m,r,td),r) = m
tdcommit(m2,f(m1,r_7,td_8,m2),td_8) = tdcommit(m1,r_7,td_8)
open(tdcommit(m1_9,r_11,td_12),f(m1_9,r_11,td_12,m2_10)) = m2_10
f(m1_14,f(m_13,r_16,td_17,m1_14),td_17,m2_15) = f(m_13,r_16,td_17,m2_15)
are declared convergent. I could not prove termination.
I assume that they really terminate.
Expect problems (such as ProVerif going into a loop) if they do not!
```

but it accepts the equations.

4.2.3 Function macros

Sometimes, terms that consist of more than just a constructor or destructor application are repeated many times. ProVerif provides a macro mechanism in order to define a function symbol that represents that term and avoid the repetition. Function macros are defined by the following declaration:

letfun $f(x_1 : t_1$ [**or fail**]$, \ldots, x_j : t_j$ [**or fail**]$) = M$.

where the macro f takes arguments $x_1, \ldots, x_j$ of types $t_1, \ldots, t_j$ and evaluates to the enriched term M (see Figure 4.2). The type of the function macro f is inferred from the type of M. The optional **or fail** after the type of each argument allows the user to control the behavior of the function macro in case some of its arguments fail:

- If **or fail** is absent and the argument fails, the function macro fails as well. For instance, with the definitions

  ```
  fun h(): t
  reduc h() = fail.

  letfun f(x:t) =
    let y = x in c0 else c1.
  ```

  h() is **fail** and f(h()) returns **fail** and f never returns c1.

- If **or fail** is present and the argument fails, the failure value is passed to the function macro, which may for instance catch it and return some non-failure result. For instance, with the same definition of h as above and the following definition of f

  ```
  letfun f(x:t or fail) =
    let y = x in c0 else c1.
  ```

  f(h()) returns c1.

Function macros can be used as constructors/destructors h in terms (see Figure 4.2). The applicability of function macros will be demonstrated by the following example.

Probabilistic asymmetric encryption. Recall that asymmetric cryptography makes use of the unary constructor `pk`, which takes an argument of type `skey` (private key) and returns a `pkey` (public key). Since the constructors of ProVerif always represent deterministic functions, we model probabilistic encryption by considering a constructor that takes the random coins used inside the encryption algorithm as an additional argument, so probabilistic asymmetric encryption is modeled by a ternary constructor internal_aenc, which takes as arguments a message of type `bitstring`, a public key of type `pkey`, and random coins of type `coins`. When encryption is used properly, the random coins must be freshly chosen at each encryption, so that the encryption of x under y is modeled by **new** r: coins; internal_aenc(x,y,r). In order to avoid writing this code at each encryption, we can define a function macro aenc, which expands to this code, as shown below. Decryption is defined in the usual way.

```
type skey.
type pkey.
type coins.

fun pk(skey): pkey.
fun internal_aenc(bitstring, pkey, coins):bitstring.

reduc forall m:bitstring, k:skey, r:coins;
  adec(internal_aenc(m,pk(k),r),k)=m.

letfun aenc(x:bitstring, y:pkey) = new r: coins; internal_aenc(x,y,r).
```

Observe that the use of probabilistic cryptography increases the complexity of the model due to the additional names introduced. This may slow down the analysis process.

4.2.4 Process macros with fail

Much like function macros above, process macros may also be declared with arguments of type t **or fail**:

let $p(x_1 : t_1$ [**or fail**],...,$x_j : t_j$ [**or fail**]) = P.

The optional **or fail** after the type of each argument allows the user to control the behavior of the process in case some of its arguments fail:

- If **or fail** is absent and the argument fails, the process blocks. For instance, with the definitions

  ```
  fun h(): t
  reduc h() = fail.

  let p(x:t) =
     let y = x in out(c, c0) else out(c, c1).
  ```

  p(h()) does nothing and p never outputs c1.

- If **or fail** is present and the argument fails, the failure value is passed to the process, which may for instance catch it and continue to run. For instance, with the same definition of h as above and the following definition of p

  ```
  let p(x:t or fail) =
     let y = x in out(c, c0) else out(c, c1).
  ```

  p(h()) outputs c1 on channel c.

4.2.5 Suitable formalizations of cryptographic primitives

In this section, we present various formalizations of basic cryptographic primitives, and relate them to the assumptions on these primitives. We would like to stress that we make *no computational soundness claims*: ProVerif relies on the symbolic, Dolev-Yao model of cryptography; its results do not apply to the computational model, at least not directly. If you want to obtain proofs of protocols in the computational

model, you should use other tools, for instance CryptoVerif (`http://cryptoverif.inria.fr`). Still, even in the symbolic model, some formalizations correspond better than others to certain assumptions on primitives. The goal of this section is to help you find the best formalization for your primitives.

Hash functions. A hash function is represented as a unary constructor h with no associated destructor or equations. The constructor takes as input, and returns, a bitstring. Accordingly, we define:

```
fun h(bitstring): bitstring.
```

The absence of any associated destructor or equational theory captures pre-image resistance, second pre-image resistance and collision resistance properties of cryptographic hash functions. In fact, far stronger properties are ensured: this model of hash functions is close to the random oracle model.

Symmetric encryption. The most basic formalization of symmetric encryption is the one based on decryption as a destructor, given in Section 3.1.2. However, formalizations that are closer to practical cryptographic schemes are as follows:

1. For block ciphers, which are deterministic, bijective encryption schemes, a better formalization is the one based on equations and given in Section 4.2.2.

2. Other symmetric encryption schemes are probabilistic. This can be formalized in a way similar to what was presented for probabilistic public-key encryption in Section 4.2.3.

   ```
   type key.
   type coins.

   fun internal_senc(bitstring, key, coins):bitstring.

   reduc forall m:bitstring, k:key, r:coins;
         sdec(internal_senc(m,k,r),k) = m.

   letfun senc(x:bitstring, y:key) = new r: coins; internal_senc(x,y,r).
   ```

   As shown in [CHW06], for protocols that do not test equality of ciphertexts, for secrecy and authentication, one can use the simpler, deterministic model of Section 3.1.2. However, for observational equivalence properties, or for protocols that test equality of ciphertexts, using the probabilistic model does make a difference.

   Note that these encryption schemes generally leak the length of the cleartext. (The length of the ciphertext depends on the length of the cleartext.) This is not taken into account in this formalization, and generally difficult to take into account in formal protocol provers, because it requires arithmetic manipulations. For some protocols, one can argue that this is not a problem, for example when the length of the messages is fixed in the protocol, so it is a priori known to the attacker. Block ciphers are not concerned by this comment since they encrypt data of fixed length.

   Also note that, in this formalization, encryption is authenticated. In this respect, this formalization is close to IND-CPA and INT-CTXT symmetric encryption. So it does not make sense to add a MAC (message authentication code) to such an encryption, as one often does to obtain authenticated encryption from unauthenticated encryption: the MAC is already included in the encryption here. If desired, it is sometimes possible to model malleability properties of some encryption schemes, by adding the appropriate equations. However, it is difficult to model general unauthenticated encryption (IND-CPA encryption) in formal protocol provers.

   In this formalization, encryption hides the encryption key. If one wants to model an encryption scheme that does not conceal the key, one can add the following destructor [ABCL09]:

   ```
   reduc forall m:bitstring, k:key, r:coins, m':bitstring, r':coins;
         samekey(internal_senc(m,k,r), internal_senc(m',k,r')) = true.
   ```

This destructor allows the attacker to test whether two ciphertexts have been built with the same key. The presence of such a destructor makes no difference for reachability properties (secrecy, correspondences) since it does not enable the attacker to construct terms that it could not construct otherwise. However, it does make a difference for observational equivalence properties. (Note that it would obviously be a serious mistake to give out the encryption key to the attacker, in order to model a scheme that does not conceal the key.)

Asymmetric encryption. A basic, deterministic model of asymmetric encryption has been given in Section 3.1.2. However, cryptographically secure asymmetric encryption schemes must be probabilistic. So a better model for asymmetric encryption is the probabilistic one given in Section 4.2.3. As shown in [CHW06], for protocols that do not test equality of ciphertexts, for secrecy and authentication, one can use the simpler, deterministic model of Section 3.1.2. However, for observational equivalence properties, or for protocols that test equality of ciphertexts, using the probabilistic model does make a difference.

It is also possible to model that the encryption leaks the key. Since the encryption key is public, we can do this simply by giving the key to the attacker:

```
reduc forall m:bitstring,pk:pkey,r:coins;getkey(internal_aenc(m,pk,r)) = pk.
```

The previous models consider a unary constructor pk that computes the public key from the secret key. An alternative (and equivalent) formalism for asymmetric encryption considers the unary constructors pk', sk' which take arguments of type seed', to capture the notion of constructing a key pair from some seed.

```
type seed'.
type pkey'.
type skey'.

fun pk'(seed'):pkey'.
fun sk'(seed'):skey'.

fun aenc'(bitstring, pkey'):bitstring.
reduc forall m:bitstring, k:seed'; adec'(aenc'(m,pk'(k)),sk'(k)) = m.
```

The addition of single quotes (') is only for distinction between the different formalizations. We have given here the deterministic version, a probabilistic version is obviously also possible.

Digital signatures. The Handbook of Applied Cryptography defines four different classes of digital signature schemes [MvOV96, Figure 11.1], we explain how to model these four classes. Deterministic signatures with message recovery were already modeled in Section 3.1.2. Probabilistic signatures with message recovery can be modeled as follows, using the same ideas as for asymmetric encryption:

```
type sskey.
type spkey.
type scoins.

fun spk(sskey):spkey.
fun internal_sign(bitstring, sskey, scoins):bitstring.
reduc forall m:bitstring, k:sskey, r:scoins;
      getmess(internal_sign(m,k,r)) = m.
reduc forall m:bitstring, k:sskey, r:scoins;
      checksign(internal_sign(m,k,r),spk(k)) = m.

letfun sign(m:bitstring, k:sskey) = new r:scoins; internal_sign(m,k,r).
```

There also exist signatures that do not allow message recovery, named digital signatures with appendix in [MvOV96]. Here is a model of such signatures in the deterministic case:

```
type sskey'.
type spkey'.
```

```
fun spk'(sskey'):spkey'.
fun sign'(bitstring, sskey'):bitstring.
reduc forall m:bitstring,k:sskey'; checksign'(sign'(m,k),spk'(k),m) = true.
```

For such signatures, the message must be given when verifying the signature, and signature verification just returns true when it succeeds. Note that these signatures hide the message as if it were encrypted; this is often a stronger property than desired. If one wants to model that these signatures do not hide the message, then one can reintroduce a destructor that leaks the message:

```
reduc forall m:bitstring,k:sskey'; getmess'(sign'(m,k)) = m.
```

Only the adversary should use this destructor; it may be an overapproximation of the capabilities of the adversary, since the message may not be fully recoverable from the signature. Probabilistic signatures with appendix can also be modeled by combining the models given above.

It is also possible to model that the signature leaks the key. Obviously, we must not leak the secret key, but we can leak the corresponding public key using the following destructor:

```
reduc forall m:bitstring,k:sskey,r:scoins;
  getkey(internal_sign(m,k,r)) = spk(k).
```

This model is for probabilistic signatures; it can be straightforwardly adapted to deterministic signatures.

Finally, as for asymmetric encryption, we can also consider unary constructors pk', sk' which take arguments of type seed', to capture the notion of constructing a key pair from some seed. We leave the construction of these models to the reader.

Message authentication codes. Message authentication codes (MACs) can be formalized by a constructor with no associated destructor or equation, much like a keyed hash function:

```
type mkey.

fun mac(bitstring, mkey):bitstring.
```

This model is strong: it considers the MAC essentially as a random oracle. It is probably the best possible model if the MAC is assumed to be a pseudo-random function (PRF). If the MAC is assumed to be unforgeable (UF-CMA), then one can add a destructor that leaks the MACed message:

```
reduc forall m:bitstring, k:mkey; get_message(mac(m,k)) = m.
```

Only the adversary should use this destructor; it may be an overapproximation of the capabilities of the adversary, since the message may not be fully recoverable from the MAC. We also remind the reader that using MACs in conjunction with symmetric encryption is generally useless in ProVerif since the basic encryption is already authenticated.

Other primitives. A simple model of Diffie-Hellman key agreements is given in Section 4.2.2, bit-commitment and blind signatures are formalized in [KR05, DKR09], trapdoor commitments are formalized in Section 4.2.2, and non-interactive zero-knowledge proofs are formalized in [BMU08]. Since defining correct models for cryptographic primitives is difficult, we recommend reusing existing definitions, such as the ones given in this manual.

4.3 Further security properties

In Section 3.2, the basic security properties that ProVerif is able to prove were introduced. In this section, we generalize our earlier presentation and introduce further security properties.

ProVerif is sound, but not complete. ProVerif's ability to reason with reachability, correspondences, and observational equivalence is sound (sometimes called correct); that is, when ProVerif says that a property is satisfied, then the model really does guarantee that property. However, ProVerif is not complete; that is, ProVerif may not be capable of proving a property that holds. Sources of incompleteness are detailed in Section 6.7.5.

4.3.1 Complex correspondence assertions, secrecy, and events

In Section 3.2.2, we demonstrated how to model correspondence assertions of the form: *"if an event e has been executed, then event e' has been previously executed."* We will now generalize these assertions considerably. The syntax for correspondence assertions is revised as follows:

query $x_1 : t_1, \ldots, x_n : t_n$; q .

where the query q is constructed by the grammar presented in Figure 4.3, such that all terms appearing in q are built by the application of constructors to the variables $x_1, \ldots, x_n$ of types $t_1, \ldots, t_n$ and all events appearing in q have been declared with the appropriate type. Equalities, disequalities, and inequalities are not allowed before an arrow ==> or alone as single fact in the query. If q or a subquery of q is of the form F ==> H and H contains an injective event, then F must be an injective event. If F is a non-injective event, it is automatically transformed into an injective event by ProVerif. The indication **public_vars** $y_1, \ldots, y_m$, when present, means that $y_1, \ldots, y_m$ are public, that is, the adversary has read access to them. The identifiers $y_1, \ldots, y_m$ must correspond to bound variables or names inside the considered process. (Variables or names bound inside enriched terms are not allowed because the expansion of terms may modify the conditions under which they are defined.) ProVerif then outputs them on public channels as soon as they are defined, to give their value to the adversary. This is mainly useful for compatibility with CryptoVerif. We will explain the meaning of these queries through many examples.

Reachability

This corresponds to the case in which the query q is just a fact F. Such a query is in fact an abbreviation for F ==> false, that is, **not** F. In other words, ProVerif tests whether F holds, but returns the following results:

- "RESULT **not** F is true." when F never holds.
- "RESULT **not** F is false." when there exists a trace in which F holds, and ProVerif displays such a trace.
- "RESULT **not** F cannot be proved." when ProVerif cannot decide either way.

For instance, we have seen **query attacker**(M) before: this query tests the secrecy of the term M and ProVerif returns "RESULT **not** attacker(M) is true." when M is secret, that is, the attacker cannot reconstruct M. When phases (see Section 4.1.6) are used, this query returns "RESULT **not** attacker(M) is true." when M is secret in all phases, or equivalently in the last phase. When M contains variables, they must be declared with their type at the beginning of the query, and ProVerif returns "RESULT **not** attacker(M) is true." when all instances of M are secret.

We can test secrecy in a specific phase n by **query attacker**(M) **phase** n. which returns "RESULT **not** attacker(M) phase n is true." when M is secret in phase n, that is, the attacker cannot reconstruct M in phase n.

We can also test whether the protocol sends a term M on a channel N (during the last phase if phases are used) by **query mess**(N, M). This query returns "RESULT **not** mess(N,M) is true." when the message M is never sent on channel N. We can also specify which phase should be considered by **query mess**(N, M) **phase** n. This query is intended for use when the channel N is private (the attacker does not have it). When the attacker has the channel N, this query is equivalent to **query attacker**(M).

Similarly, we can test whether the element $(M_1, \ldots, M_n)$ is present in table d by **query table**($d(M_1, \ldots, M_n)$).

ProVerif can also evaluate the reachability of events within a model using the following query:

query $x_1 : t_1, \ldots, x_n : t_n$; **event** $(e(M_1, \ldots, M_k))$.

This query returns "RESULT **not** event($e(M_1, \ldots, M_k)$) is true." when the event is not reachable. Such queries are useful for debugging purposes, for example, to detect unreachable branches of a model. With reference to the "Hello World" script (`docs/hello_ext.pv`) in Chapter 2, one could examine as to whether the else branch is reachable.

Figure 4.3 Grammar for correspondence assertions

q ::=	query
$cq\ pv$	reachability or correspondence
secret $x\ pv$ [*options*]	secrecy
pv ::=	public variables
	empty
public_vars $y_1, \ldots, y_m$	public variables
cq ::=	reachability or correspondence query
F_1 && ... && F_n	reachability
F_1 && ... && F_n ==> H	correspondence
H ::=	hypothesis
F	fact
$M\ op\ N$	constraint with $op \in \{<, <=, >, >=, =; <>\}$
H && H	conjunction
H || H	disjunction
false	constant false
(F ==> H)	nested correspondence
F ::=	fact
AF	action fact
AF@t	action fact executed at time t
AF ::=	action fact
attacker(M)	the attacker has M (in any phase)
attacker(M) **phase** n	the attacker has M in phase n
mess(N, M)	M is sent on channel N (in the last phase)
mess(N, M) **phase** n	M is sent on channel N in phase n
table($d(M_1, \ldots, M_n)$)	the element $M_1, \ldots, M_n$ is in table d (in any phase)
table($d(M_1, \ldots, M_n)$) **phase** n	the element $M_1, \ldots, M_n$ is in table d in phase n
event($e(M_1, \ldots, M_n)$)	non-injective event
inj−event($e(M_1, \ldots, M_n)$)	injective event

More generally, such a query can be F_1 && ... && F_n, which is in fact an abbreviation for F_1 && ... && F_n ==> false, that is, **not** (F_1 && ... && F_n): ProVerif tries to prove that $F_1, \ldots, F_n$ are not simultaneously reachable. The similar query with **inj−event** instead of **event** is useless: it has the same meaning as the one with **event**. Injective events are useful only for correspondences described below. Equalities, disequalities, and inequalities are not allowed in reachability queries as mentioned above.

Basic correspondences

Basic correspondences are queries $q = F_1$ && ... && F_n ==> H where H does not contain nested correspondences. They mean that, if F_1, ..., F_n hold, then H also holds. We have seen such correspondences in Section 3.2.2. We can extend them to conjunctions and disjunctions of events in H. For instance,

```
query event(e0) ==> event(e1) && event(e2).
```

means that, if e_0 has been executed, then e_1 and e_2 have been executed. Similarly,

```
query event(e0) ==> event(e1) || event(e2).
```

means that, if e_0 has been executed, then e_1 or e_2 has been executed. If the correspondence F ==> H holds, F is an event, and H contains events, then the events in H must be executed before the event F (or at the same time as F in case an event in H may be equal to F). This property is proved by stopping the execution of the process just after the event F.

Conjunctions and disjunctions can be combined:

```
query event(e0) ==> event(e1) || (event(e2) && event(e3)).
```

means that, if e_0 has been executed, then either e_1 has been executed, or e_2 and e_3 have been executed. The conjunction has higher priority than the disjunction, but one should use parentheses to disambiguate the expressions. The events can of course have arguments, and can also be injective events. For instance,

```
query inj-event(e0) ==> event(e1) || (inj-event(e2) && event(e3)).
```

means that each execution of e_0 corresponds to either an execution of e_1 (perhaps the same execution of e_1 for different executions of e_0), or to a *distinct* execution of e_2 and an execution of e_3. Note that using **inj−event** or **event** before the arrow ==> does not change anything, since **event** is automatically changed into **inj−event** before ==> when there is **inj−event** after the arrow ==>.

Conjunctions are also allowed before the arrow ==>. For instance,

```
event(e1(M1)) && ... && event(en(Mn)) ==> H
```

means that, if events $e_1(M_1)$, ..., $e_n(M_n)$ are executed, then H holds. When there are several injective events before the arrow ==>, the query means that for each tuple of executed injective events before the arrow, there are distinct injective events after the arrow. For instance, the query

```
inj-event(e1) && inj-event(e2) ==> inj-event(e3)
```

requires that if event e_1 is executed n_1 times and event e_2 is executed n_2 times, then event e_3 is executed at least $n_1 \times n_2$ times.

Correspondences may also involve the knowledge of the attacker or the messages sent on channels. For instance,

```
query attacker(M) ==> event(e1).
```

means that, when the attacker knows M, the event e_1 has been executed. Conversely,

```
query event(e1) ==> attacker(M).
```

means that, when event e_1 has been executed, the attacker knows M. (In practice, ProVerif may have more difficulties proving the latter correspondence. Technically, ProVerif needs to conclude **attacker**(M) from facts that occur in the hypothesis of a clause that concludes **event**(e_1); this hypothesis may get simplified during the resolution process in a way that makes the desired facts disappear.)

One may also use equalities, disequalities, and inequalities after the arrow ==>. For instance, assuming a free name a,

```
query x:t; event(e(x)) ==> x = a.
```

means that the event e(x) can be executed only when x is a. Similarly,

```
query x:t, y:t'; event(e(x)) ==> event(e'(y)) && x = f(y)
```

means that, when the event e(x) is executed, the event **event**(e'(y)) has been executed and x = f(y). Using disequalities,

```
query x:t; event(e(x)) ==> x <> a.
```

means that the event e(x) can be executed only when x is different from a.

Nested correspondences

The grammar permits the construction of *nested correspondences*, that is, correspondences F_1 && ... && F_n ==> H in which some of the events H are replaced with correspondences. Such correspondences allow us to order events. More precisely, in order to explain a nested correspondence

F_1 && ... && F_n ==> H, let us define a hypothesis H_s by replacing all arrows ==> of H with conjunctions &&. The nested correspondence F_1 && ... && F_n ==> H holds if and only if the basic correspondence F_1 && ... && F_n ==> H_s holds and additionally, for each F' ==> H' that occurs in F_1 && ... && F_n ==> H, if F' is an event, then the events of H' have been executed before F' (or at the same time as F' in case events in H' may be equal to F').[1] For example,

$$\mathbf{event}(e_0) \implies (\mathbf{event}(e_1) \implies (\mathbf{event}(e_2) \implies \mathbf{event}(e_3)))$$

is true when, if the event e_0 has been executed, then events e_3, e_2, e_1 have been previously executed in that order and before e_0. In contrast, the correspondence

$$\mathbf{event}(e_0) \implies (\mathbf{event}(e_1) \implies \mathbf{event}(e_2)) \;\&\&\; (\mathbf{event}(e_3) \implies \mathbf{event}(e_4))$$

holds when, if the event e_0 has been executed, then e_2 has been executed before e_1 and e_4 before e_3, and those occurrences of e_1 and e_3 have been executed before e_0.

Even if the grammar of correspondences does not explicitly require that facts F that occur before arrows in nested correspondences are events (or injective events), in practice they are because the only goal of nested correspondences is to order such events.

Our study of the JFK protocol, which can be found in the subdirectory `examples/pitype/jfk` (if you installed by OPAM in the switch ⟨*switch*⟩, the directory `~/.opam/`⟨*switch*⟩`/doc/proverif/examples/pitype/jfk`), provides several interesting examples of nested correspondence assertions used to prove the correct ordering of messages of the protocol.

ProVerif proves nested correspondences essentially by proving several correspondences. For instance, in order to prove

$$\mathbf{event}(e_0) \implies (\mathbf{event}(e_1) \implies \mathbf{event}(e_2))$$

where the events e_0, e_1, e_2 may have arguments, ProVerif proves that each execution of e_0 is preceded by the execution of an instance of e_1, and that, when e_0 is executed, each execution of that instance of e_1 is preceded by the execution of an instance of e_2.

A typical usage of nested correspondences is to order all messages in a protocol. One would like to prove a correspondence in the style:

$$\begin{array}{l}\mathbf{inj\text{-}event}(e_{\mathrm{end}}) \implies \\ \quad (\mathbf{inj\text{-}event}(e_n) \implies \ldots \implies (\mathbf{inj\text{-}event}(e_1) \implies \mathbf{inj\text{-}event}(e_0)))\end{array}$$

where e_0 means that the first message of the protocol has been sent, e_i ($i > 0$) means that the i-th message of the protocol has been received and the $(i+1)$-th has been sent, and finally e_{end} means that the last message of the protocol has been received. (These events have at least as arguments the messages of the protocol.)

Temporal correspondences

Correspondences and nested correspondences allow one to verify the order in which facts occur in execution traces. The grammar also permits to reason on the order of facts through time variables. In a query, each fact F can be associated with a variable t of type time with the construct $F@t$, meaning that the fact F is executed at time t. When several facts are associated with time variables $t, t', \ldots$, we can reason on the order in which these facts are executed using equalities, inequalities, and disequalities between the time variables, *e.g.* $t < t'$, $t = t'$, ... For example, in our study of the Yubikey protocol, which can be found in the file `examples/pitype/lemma/yubikey-less-axioms-time.pv` (if you installed by OPAM in the switch ⟨*switch*⟩, the file `~/.opam/`⟨*switch*⟩`/doc/proverif/examples/pitype/lemma/yubikey-less-axioms-time.pv`), a server executes the event Login(pid,k,i,x) every time it accepts a connection from a Yubikey with identity pid and key k. The value i is the value of the server's

[1] Although the meaning of a basic correspondence such as $\mathbf{event}(e_0)$ ==> $\mathbf{event}(e_1)$ is similar to a logical implication, the meaning of a nested correspondence such as $\mathbf{event}(e_0)$ ==> ($\mathbf{event}(e_1)$ ==> $\mathbf{event}(e_2)$) is very different from the logical formula $\mathbf{event}(e_0) \Rightarrow (\mathbf{event}(e_1) \Rightarrow \mathbf{event}(e_2))$ in classical logic, which would mean $(\mathbf{event}(e_0) \wedge \mathbf{event}(e_1)) \Rightarrow \mathbf{event}(e_2)$. The nested correspondence $\mathbf{event}(e_0)$ ==> ($\mathbf{event}(e_1)$ ==> $\mathbf{event}(e_2)$) rather means that, if e_0 is executed, then some instance of e_1 is executed (before e_0), and if that instance of e_1 is executed, then an instance of e_2 is executed (before e_1). So the nested correspondence is similar to an abbreviation for the two correspondences $\mathbf{event}(e_0)$ ==> $\mathbf{event}(\sigma e_1)$ and $\mathbf{event}(\sigma e_1)$ ==> $\mathbf{event}(\sigma e_2)$ for some substitution σ.

counter and x is the value of the Yubikey's counter sent to the server. The following query ensures that the server never executes two login events at different times with the same value for the identity, the key, and Yubikey's counter.

```
query t:time, t':time, pid:bitstring, k:bitstring, i:nat, i':nat,
  x:nat, x':nat;
    event(Login(pid,k,i,x))@t && event(Login(pid,k,i',x))@t' ==> t = t'.
```

Formally, the query is true when, if two Login events are executed with the same key, identity, and Yubikey's counter value, then the two events are executed at the same time. Since the semantics of ProVerif's calculus only allows events to be executed one at a time, it also implies that the two events are equal, i.e., $i = i'$.

Using temporal correspondences allows one to be more precise than basic correspondences. For example, the following query is not equivalent to the previous one.

```
query pid:bitstring, k:bitstring, i:nat, i':nat, x:nat, x':nat;
    event(Login(pid,k,i,x)) && event(Login(pid,k,i',x)) ==> i = i'.
```

Indeed, an execution trace where the Login event is executed twice with the same arguments (at different times) would satisfy this query but not the former one.

Temporal variables can be used to compare facts from both the premise and the conclusion of the query. For example,

$$\textbf{event}(e_0)\ \&\&\ \textbf{event}(e_1)@t_1 \implies \textbf{event}(e_2)@t_2\ \&\&\ t_2 < t_1$$

is true when if two events e_0 and e_1 are executed then the event e_2 must have been executed strictly before the event e_1.

Note that temporal variables can be used in combination with injective events and nested correspondences, although they overlap in some cases. For example, the query

$$\textbf{event}(e_0) \implies (\textbf{event}(e_1) \implies \textbf{event}(e_2))$$

is equivalent to the query

$$\textbf{event}(e_0) \implies \textbf{event}(e_1)@t_1\ \&\&\ \textbf{event}(e_2)@t_2\ \&\&\ t_2 <= t_1$$

The grammar of correspondences also allows attacker, message, and table facts to be associated with time variables but ProVerif is in practice more successful in proving correspondence queries containing mainly events. Note that the type time can only be used in queries and cannot be used in declarations of processes, function symbols, names, ...

Secrecy

The query **query secret** x provides an alternative way to test secrecy to **query attacker**(M). The latter query is meant to test whether the attacker can compute the term M, built from free names. The query **query secret** x can test the secrecy of the bound name or variable x. The identifier x must correspond to a bound variable or name inside the considered process. (Variables or names bound inside enriched terms are not allowed because the expansion of terms may modify the conditions under which they are defined.) This query comes in two flavors:

- **query secret** x, **query secret** x [reachability], or **query secret** x [pv_reachability] tests whether the attacker can compute a value stored in the variable x or equal to the bound name x.
- **query secret** x [real_or_random] or **query secret** x [pv_real_or_random] tests whether the attacker can distinguish each value of x from a fresh name (representing a fresh random value). This query is in fact encoded as an observational query between processes that differ only by terms. Such queries are explained in the next section.

This query is designed for compatibility with CryptoVerif: the options that start with pv_ apply only to ProVerif; those that start with cv_ apply only to CryptoVerif and are ignored by ProVerif; the others apply to both tools. The various options make it possible to test, in each tool, whether the attacker can compute the value of x or whether it can distinguish it from a fresh random value. (The former is the default in ProVerif while the latter is the default in CryptoVerif.)

4.3.2 Observational equivalence

The notion of indistinguishability is a powerful concept which allows us to reason about complex properties that cannot be expressed as reachability or correspondence properties. The notion of indistinguishability is generally named *observational equivalence* in the formal model. Intuitively, two processes P and Q are observationally equivalent, written $P \approx Q$, when an active attacker cannot distinguish P from Q. Formal definitions can be found in [AF01, BAF08]. Using this notion, one can for instance specify that a process P follows its specification Q by saying that $P \approx Q$. ProVerif can prove some observational equivalences, but not all of them because their proof is complex. In this section, we present the queries that enable us to prove observational equivalences using ProVerif.

Strong secrecy

A first class of equivalences that ProVerif can prove is strong secrecy. Strong secrecy means that the attacker is unable to distinguish when the secret changes. In other words, the value of the secret should not affect the observable behavior of the protocol. Such a notion is useful to capture the attacker's ability to learn partial information about the secret: when the attacker learns the first component of a pair, for instance, the whole pair is secret in the sense of reachability (the attacker cannot reconstruct the whole pair because it does not have the second component), but it is not secret in the sense of strong secrecy (the attacker can notice changes in the value of the pair, since it has its first component). The concept is particularly important when the secret consists of known values. Consider for instance a process P that uses a boolean b. The variable b can take two values, *true* or *false*, which are both known to the attacker, so it is not secret in the sense of reachability. However, one may express that b is strongly secret by saying that $P\{\mathit{true}/b\} \approx P\{\mathit{false}/b\}$: the attacker cannot determine whether b is *true* or *false*. ($\{\mathit{true}/b\}$ denotes the substitution that replaces b with *true*.)

The strong secrecy of values $x_1, \ldots, x_n$ is denoted by

noninterf $x_1, \ldots, x_n$.

When the process under consideration is P, this query is true if and only if

$$P\{M_1/x_1, \ldots, M_n/x_n\} \approx P\{M'_1/x_1, \ldots, M'_n/x_n\}$$

for all terms $M_1, \ldots, M_n, M'_1, \ldots, M'_n$. ($\{M_1/x_1, \ldots, M_n/x_n\}$ denotes the substitution that replaces x_1 with M_1, ..., x_n with M_n.) In other words, the attacker cannot distinguish changes in the values of $x_1, \ldots, x_n$. The values $x_1, \ldots, x_n$ must be free names of P, declared by **free** $x_i : t_i$ [**private**]. This point is particularly important: if $x_1, \ldots, x_n$ do not occur in P or occur as bound names or variables in P, the query **noninterf** $x_1, \ldots, x_n$ holds trivially, because $P\{M_1/x_1, \ldots, M_n/x_n\} = P\{M'_1/x_1, \ldots, M'_n/x_n\}$! To express secrecy of bound names or variables, one can use **choice**, described below. In the equivalence above, the attacker is permitted to replace the values $x_1, \ldots, x_n$ with any term $M_1, \ldots, M_n, M'_1, \ldots, M'_n$ it can build, that is, any term that can be built from public free names, public constructors, and fresh names created by the attacker. These terms cannot contain bound names (or private free names).

For instance, this strong secrecy query can be used to show the secrecy of a payload sent encrypted under a session key. Here is a trivial example of a such situation, in which we use a previously shared long-term key k as session key (file `docs/ex_noninterf1.pv`).

```
1  free c: channel.
2
3  (* Shared key encryption *)
4  type key.
5  fun senc(bitstring,key): bitstring.
6  reduc forall x: bitstring, y: key; sdec(senc(x,y),y) = x.
7
8  (* The shared key *)
9  free k:key [private].
10
11 (* Query *)
12 free secret_msg: bitstring [private].
```

```
noninterf secret_msg.

process (!out(c, senc(secret_msg, k))) |
  (!in(c, x:bitstring); let s = sdec(x, k) in 0)
```

One can also specify the set of terms in which $M_1, \ldots, M_n, M'_1, \ldots, M'_n$ are taken, using a variant of the **noninterf** query:

noninterf x_1 **among** $(M_{1,1}, \ldots, M_{1,k_1})$,
$\ldots$,
x_n **among** $(M_{n,1}, \ldots, M_{n,k_n})$.

This query is true if and only if

$$P\{M_1/x_1, \ldots, M_n/x_n\} \approx P\{M'_1/x_1, \ldots, M'_n/x_n\}$$

for all terms $M_1, M'_1 \in \{M_{1,1}, \ldots, M_{1,k_1}\}$, $\ldots$, $M_n, M'_n \in \{M_{n,1}, \ldots, M_{n,k_n}\}$. Obviously, the terms $M_{j,1}, \ldots, M_{j,k_j}$ must have the same type as x_j. For instance, the secrecy of a boolean b could be expressed by **noninterf** b **among** (true, false).

Consider the following example (`docs/ex_noninterf2.pv`) in which the attacker is asked to distinguish between sessions which output x $\in$ {n, h(n)}, where n is a private name.

```
free c: channel.

fun h(bitstring): bitstring.

free x,n: bitstring [private].
noninterf x among (n, h(n)).

process out(c, x)
```

Note that **free** x,n: bitstring [**private**]. is a convenient shorthand for

```
free x: bitstring [private].
free n: bitstring [private].
```

More complex examples can be found in subdirectory `examples/pitype/noninterf` (if you installed by OPAM in the switch ⟨*switch*⟩, the directory `~/.opam/`⟨*switch*⟩`/doc/proverif/examples/pitype/noninterf`).

Off-line guessing attacks

Protocols may rely upon *weak secrets*, that is, values with low entropy, such as human-memorable passwords. Protocols which rely upon weak secrets are often subject to off-line guessing attacks, whereby an attacker passively observes, or actively participates in, an execution of the protocol and then has the ability to verify if a guessed value is indeed the weak secret without further interaction with the protocol. This makes it possible for the attacker to enumerate a dictionary of passwords, verify each of them, and find the correct one. The absence of off-line guessing attacks against a name n can be tested by the query:

weaksecret n.

where n is declared as a private free name: **free** $n : t$ [**private**]. ProVerif then tries to prove that the attacker cannot distinguish a correct guess of the secret from an incorrect guess. This can be written formally as an observational equivalence

P | **phase** 1; **out**(c, n) $\approx$ P | **phase** 1; **new** $n' : t$; **out**(c, n')

where P is the process under consideration and t is the type of n. In phase 0, the attacker interacts with the protocol P. In phase 1, the attacker can no longer interact with P, but it receives either the correct password n or a fresh (incorrect) password n', and it should not be able to distinguish between these two situations.

As an example, we will consider the naïve voting protocol introduced by Delaune & Jacquemard [DJ04]. The protocol proceeds as follows. The voter V constructs her ballot by encrypting her vote v with the public key of the administrator. The ballot is then sent to the administrator whom is able to decrypt the message and record the voter's vote, as modeled in the file `docs/ex_weaksecret.pv` shown below:

```
1  free c: channel.
2
3  type skey.
4  type pkey.
5
6  fun pk(skey): pkey.
7  fun aenc(bitstring, pkey): bitstring.
8
9  reduc forall m: bitstring, k: skey; adec(aenc(m,pk(k)),k) = m.
10
11 free v: bitstring [private].
12 weaksecret v.
13
14 let V(pkA:pkey) = out(c, aenc(v, pkA)).
15
16 let A(skA:skey) = in(c,x:bitstring); let v' = adec(x, skA) in 0.
17
18 process
19   new skA: skey;
20   let pkA = pk(skA) in
21   out(c, pkA);
22   ! (V(pkA) | A(skA))
```

The voter's vote is syntactically secret; however, if the attacker is assumed to know a small set of possible votes, then v can be deduced from the ballot. The off-line guessing attack can be thwarted by the use of a probabilistic public-key encryption scheme.

More examples regarding guessing attacks can be found in subdirectory `examples/pitype/weaksecr` (if you installed by OPAM in the switch ⟨*switch*⟩, the directory `~/.opam/`⟨*switch*⟩`/doc/proverif/examples/pitype/weaksecr`).

Observational equivalence between processes that differ only by terms

The most general class of equivalences that ProVerif 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 **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'. (The keyword **diff** is also allowed as a synonym for **choice**; **diff** is used in research papers.) For example, the secret ballot (privacy) property of an electronic voting protocol can be expressed as:

$$P(sk_A, v_1) \mid P(sk_B, v_2) \approx P(sk_A, v_2) \mid P(sk_B, v_1) \tag{4.1}$$

where P is the voter process, sk_A (respectively sk_B) is the voter's secret key and v_1 (respectively v_2) is the candidate for whom the voter wishes to vote for: one cannot distinguish the situation in which A votes for v_1 and B votes from v_2 from the situation in which A votes for v_2 and B votes for v_1. (The simpler equivalence $P(sk_A, v_1) \approx P(sk_A, v_2)$ typically does not hold because, if A is the only voter, one can know for whom she voted from the final result of the election.) The pair of processes (4.1) can be expressed as a single biprocess as follows:

$$P(sk_A, \mathbf{choice}[v_1, v_2]) \mid P(sk_B, \mathbf{choice}[v_2, v_1])$$

Accordingly, we extend our grammar for terms to include **choice**[M,N].

Unlike the previous security properties we have studied, there is no need to explicitly tell ProVerif that a script aims at verifying an observational equivalence, since this can be inferred from the occurrence of **choice**[M,N]. It should be noted that the analysis of observational equivalence is incompatible with other security properties, that is, scripts in which **choice**[M,N] appears cannot contain **query**, **noninterf**, nor **weaksecret**. (For this reason, you may have to write several distinct input files in order to prove several properties of the same protocol. You can use a preprocessor such as `m4` or `cpp` to generate all these files from a single master file.)

Example: Decisional Diffie-Hellman assumption The decisional Diffie-Hellman (DDH) assumption states that, given a cyclic group G of prime order q with generator g, (g^a, g^b, g^{ab}) is computationally indistinguishable from (g^a, g^b, g^c), where a, b, c are random elements from $\mathbb{Z}_q^*$. A formal counterpart of this property can be expressed as an equivalence using the ProVerif script below (file `docs/dh-fs.pv`).

```
free d: channel.

type G.
type exponent.

const g: G [data].
fun exp(G, exponent): G.

equation forall x: exponent, y: exponent; exp(exp(g,x),y) = exp(exp(g,y),x).

process
  new a: exponent; new b: exponent; new c: exponent;
  out(d, (exp(g,a), exp(g,b), choice[exp(exp(g,a),b), exp(g,c)]))
```

ProVerif succeeds in proving this equivalence. Intuitively, this result shows that our model of the Diffie-Hellman key agreement is stronger than the Decisional Diffie-Hellman assumption.

Observe that the biprocess **out**(d,(exp(g,a),exp(g,b),**choice**[exp(exp(g,a),b),exp(g,c)])) is equivalent to

```
out(choice[d,d],(choice[exp(g,a),exp(g,a)], choice[exp(g,b),exp(g,b)],
   choice[exp(exp(g,a),b),exp(g,c)])).
```

That is, **choice**[M,M] may be abbreviated as M; it follows immediately that the **choice** operator is only needed to model the terms that are different in the pair of processes.

Real-or-random secrecy In the computational model, one generally expresses the secrecy of a value x by saying that x is indistinguishable from a fresh random value. One can express a similar idea in the formal model using observational equivalence. For instance, this notion can be used for proving secrecy of a session key k, as in the following variant of the fixed handshake protocol of Chapter 3 (file `docs/ex_handshake_RoR.pv`).

```
free c:channel.

let clientA(pkA:pkey,skA:skey,pkB:spkey) =
  out(c,pkA);
  in(c,x:bitstring);
  let y = adec(x,skA) in
  let (=pkA,=pkB,k:key) = checksign(y,pkB) in
  new random:key;
  out(c, choice[k,random]).

let serverB(pkB:spkey,skB:sskey,pkA:pkey) =
  in(c,pkX:pkey);
  new k:key;
  out(c,aenc(sign((pkX,pkB,k),skB),pkX)).
```

```
15
16 process
17   new skA:skey;
18   new skB:sskey;
19   let pkA = pk(skA) in out(c,pkA);
20   let pkB = spk(skB) in out(c,pkB);
21   ( (!clientA(pkA,skA,pkB)) | (!serverB(pkB,skB,pkA)) )
```

In Line 9, one outputs to the attacker either the real key (k) or a random key (random), and the equivalence holds when the attacker cannot distinguish these two situations. As ProVerif finds, the equivalence does not hold in this example, because of a replay attack: the attacker can replay the message from the server B to the client A, which leads several sessions of the client to have the same key k. The attacker can distinguish this situation from a situation in which the key is a fresh random number (random) generated at each session of the client. Another example can be found in Section 5.4.2.

When the observational equivalence proof fails on the biprocess given by the user, ProVerif tries to simplify that biprocess by transforming as far as possible tests that occur in subprocesses into tests done inside terms, which increases the chances of success of the proof. The proof is then retried on the simplified process(es). This simplification of biprocesses can be turned off by the setting **set** simplifyProcess = false. (See Section 6.6.2 for details on this setting.) More complex examples using **choice** can be found in subdirectory `examples/pitype/choice` (if you installed by OPAM in the switch ⟨*switch*⟩, the directory `~/.opam/`⟨*switch*⟩`/doc/proverif/examples/pitype/choice`).

Remarks The absence of off-line guessing attacks can also be expressed using **choice**:

P | **phase** 1; **new** $n' : t$; **out**(c, **choice**[n,n'])

This is how ProVerif handles guessing attacks internally, but using **weaksecret** is generally more convenient in practice. (For instance, one can query for the secrecy of several weak secrets in the same ProVerif script.)

Strong secrecy **noninterf** $x_1, \ldots, x_n$ can also be formalized using **choice**, by inputting two messages x'_i, x''_i for each $i \leq n$ and defining x_i by **let** $x_i =$ **choice**$[x'_i, x''_i]$ before starting the protocol itself (possibly in an earlier phase than the protocol). However, the query **noninterf** is typically much more efficient than **choice**. On the other hand, in the presence of equations that can be applied to the secrets, **noninterf** commonly leads to false attacks. So we recommend trying with **noninterf** for properties that can be expressed with it, especially when there is no equation, and using **choice** in the presence of equations or for properties that cannot be expressed using **noninterf**.

Strong secrecy with **among** can also be encoded using **choice**. That may require many equivalences when the sets are large, even if some examples are very easy to encode. For instance, the query **noninterf** b **among** (true, false) can also be encoded as **let** b = **choice**[true, false] **in** P where P is the protocol under consideration.

Static equivalence [AF01] is an equivalence between frames, that is, substitutions with hidden names

$\phi =$ **new** $n_1 : t_1$; ... **new** $n_k : t_k$; $\{M_1/x_1, \ldots, M_l/x_l\}$
$\phi' =$ **new** $n'_1 : t'_1$; ... **new** $n'_{k'} : t'_{k'}$; $\{M'_1/x_1, \ldots, M'_l/x_l\}$

Static equivalence corresponds to the case in which the attacker receives either the messages $M_1, \ldots, M_l$ or $M'_1, \ldots, M'_l$, and should not be able to distinguish between these two situations; static equivalence can be expressed by the observational equivalence

new $n_1 : t_1$; ... **new** $n_k : t_k$; **out**(c, $(M_1, \ldots, M_l)$)
$\approx$
new $n'_1 : t'_1$; ... **new** $n'_{k'} : t'_{k'}$; **out**(c, $(M'_1, \ldots, M'_l)$)

which can always be written using **choice**:

new $n_1 : t_1$; ... **new** $n_k : t_k$; **new** $n'_1 : t'_1$; ... **new** $n'_{k'} : t'_{k'}$;
out(c, (**choice**$[M_1, M'_1]$, ..., **choice**$[M_l, M'_l]$))

The Diffie-Hellman example above is an example of static equivalence.

Internally, ProVerif proves a property much stronger than observational equivalence of P and Q. In fact, it shows that for each reachable test, the same branch of the test is taken both in P and in Q; for each reachable destructor application, the destructor application either succeeds both in P and Q or fails both in P and Q; for each reachable configuration with an input and an output on private channels, the channels are equal in P and in Q, or different in P and in Q. In other words, it shows that each reduction step is executed in the same way in P and Q. Because this property is stronger than observational equivalence, we may have "false attacks" in which this property is wrong but observational equivalence in fact holds. When ProVerif does not manage to prove observational equivalence, it tries to reconstruct an attack against the stronger property, that is, it provides a trace of P and Q that arrives at a point at which P and Q reduce in a different way. This trace explains why the proof fails, and may also enable the user to understand if observational equivalence really does not hold, but it does not provide a proof that observational equivalence does not hold. That is why ProVerif never concludes "RESULT [Query] is false" for observational equivalences; when the proof fails, it just concludes "RESULT [Query] cannot be proved".

Observational equivalence with synchronizations Synchronizations (see Section 4.1.7) can help proving equivalences with **choice**, because they allow swapping data between processes at the synchronization points [BS16]. The following toy example illustrates this point:

```
free c: channel.
free m,n: bitstring.

process
    (
        out(c,m);
        sync 1;
        out(c,choice[m,n])
    )|(
        sync 1;
        out(c,choice[n,m])
    )
```

The two processes represented by this biprocess are observationally equivalent, and this property is proved by swapping m and n in the second component of **choice** at the synchronization point. By default, ProVerif tries all possible swapping strategies in order to prove the equivalence. It is also possible to choose the swapping strategy in the input file by **set** swapping = "swapping stragegy"., or to choose it interactively by adding **set** interactiveSwapping = true. to the input file. In the latter case, ProVerif displays a description of the possible swappings and asks the user which swapping strategy to choose.

A swapping strategy is described as follows. The swapping strategies are permutations of the synchronizations, represented by their tag (given by the user or chosen automatically by ProVerif as explained in Section 4.1.7; for stability of the tags, when a swapping strategy is given, it is recommend that the user specifies the tags). They are denoted as follows:

$$tag_{1,1} \text{->} \ldots \text{->} tag_{1,n_1};\ldots;tag_{k,1} \text{->} \ldots \text{->} tag_{k,n_k}$$

which means that $tag_{i,j}$ has image $tag_{i,j+1}$ when $j < n_i$ and tag_{i,n_i} has image $tag_{i,1}$ by the permutation. (In other words, we give the cycles of the permutation.) When the tag of a synchronization does not appear in the swapping strategy, data is not swapped at that synchronization. For instance, the previous example may the rewritten:

```
free c: channel.
free m,n: bitstring.

set swapping = "tag1 -> tag2".

process
```

```
    (
        out(c,m);
        sync 1 [tag1];
        out(c,choice[m,n])
    )|(
        sync 1 [tag2];
        out(c,choice[n,m])
    )
```

with additional tags, and the swapping strategy is tag1 −> tag2.

When a synchronization is tagged with a tag that contains the string noswap, data is not swapped at that synchronization.

Swapping data at synchronizations point can help for instance proving ballot secrecy in e-voting protocols: as mentioned above, this property is proved by showing that the two processes represented by the biprocess

$$P(sk_A, \mathbf{choice}[v_1, v_2]) \mid P(sk_B, \mathbf{choice}[v_2, v_1])$$

are observationally equivalent, and proving this property often requires swapping the votes v_1 and v_2. This technique is illustrated on the FOO e-voting protocol in the file `examples/pitype/sync/foo.pv` of the documentation package `proverifdoc2.02.tar.gz`. Other examples appear in the directory `examples/pitype/sync/` in that package.

Observational equivalence between two processes

ProVerif can also prove equivalence $P \approx Q$ between two processes P and Q presented separately, using the following command (instead of **process** P)

equivalence P Q

where P and Q are processes that do not contain **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: the structure of the two processes must be fairly similar. Here is a toy example:

```
type key.
type macs.

fun mac(bitstring, key): macs.

free c: channel.

equivalence
   ! new k:key; ! new a: bitstring; out(c, mac(a, k))
   ! new k:key; new a: bitstring; out(c, mac(a, k))
```

The difference between the two processes is that the first process can use the same key k for sending several MACs, while the second one sends one MAC for each key k. Even though the structure of the two processes is slightly different (there is an additional replication in the first process), ProVerif manages to merge these two processes into a single biprocess:

```
!
new k_39: key;
!
new a_40: bitstring;
new k_41: key;
new a_42: bitstring;
out(c, choice[mac(a_40,k_39),mac(a_42,k_41)])
```

and to prove that the two processes are observationally equivalent.

When proving an equivalence by **equivalence** P Q, the processes P and Q must not contain synchronizations **sync** n (see Section 4.1.7).

Chapter 5

Needham-Schroeder public key protocol: Case Study

The Needham-Schroeder public key protocol [NS78] is intended to provide mutual authentication of two principals Alice A and Bob B. Although it is not stated in the original description, the protocol may also provide a secret session key shared between the participants. In addition to the two participants, we assume the existence of a trusted key server S.

The protocol proceeds as follows. Alice contacts the key server S and requests Bob's public key. The key server responds with the key pk(skB) paired with Bob's identity, signed using his private signing key for the purposes of authentication. Alice proceeds by generating a nonce Na, pairs it with her identity A, and sends the message encrypted with Bob's public key. On receipt, Bob decrypts the message to recover Na and the identity of his interlocutor A. Bob then establishes Alice's public key pk(skA) by requesting it to the key server S. Bob then generates his nonce Nb and sends the message (Na,Nb) encrypted for Alice. Finally, Alice replies with the message aenc(Nb, pk(skB)). The rationale behind the protocol is that, since only Bob can recover Na, only he can send message 6; and hence authentication of Bob should hold. Similarly, only Alice should be able to recover Nb; and hence authentication of Alice is expected on receipt of message 7. Moreover, it follows that Alice and Bob have established the shared secrets Na and Nb which can subsequently be used as session keys. The protocol can be summarized by the following narration:

$$
\begin{array}{llllll}
(1) & A & \rightarrow & S & : & \text{(A, B)} \\
(2) & S & \rightarrow & A & : & \text{sign}\,(\text{(B, pk(skB))}, \text{skS}) \\
(3) & A & \rightarrow & B & : & \text{aenc}(\text{(Na, A)}, \text{pk(skB)}) \\
(4) & B & \rightarrow & S & : & \text{(B, A)} \\
(5) & S & \rightarrow & B & : & \text{sign}\,(\text{(A, pk(skA))}, \text{skS}) \\
(6) & B & \rightarrow & A & : & \text{aenc}(\text{(Na, Nb)}, \text{pk(skA)}) \\
(7) & A & \rightarrow & B & : & \text{aenc(Nb, pk(skB))}
\end{array}
$$

Informally, the protocol is expected to satisfy the following properties:

1. Authentication of A to B: if B reaches the end of the protocol and he believes he has done so with A, then A has engaged in a session with B.

2. Authentication of B to A: similarly to the above.

3. Secrecy for A: if A reaches the end of the protocol with B, then the nonces Na and Nb that A has are secret; in particular, they are suitable for use as session keys for preserving the secrecy of an arbitrary term M in the symmetric encryption senc(M,K) where $\text{K} \in \{\text{Na}, \text{Nb}\}$.

4. Secrecy for B: similarly.

However, nearly two decades after the protocol's inception, Gavin Lowe discovered a man-in-the-middle attack [Low96]. An attacker I engages Alice in a legitimate session of the protocol; and in parallel, the attacker is able to impersonate Alice in a session with Bob. In practice, one may like to consider the

attacker to be a malicious retailer I whom Alice is willing to communicate with (presumably without the knowledge that the retailer is corrupt), and Bob is an honest institution (for example, a bank) whom Alice conducts legitimate business with. In this scenario, the honest bank B is duped by the malicious retailer I who is pertaining to be Alice. The protocol narration below describes the attack (with the omission of key distribution).

$$\begin{array}{lcl:l} A & \rightarrow & I & \text{aenc((Na,A), pk(skI))} \\ I & \rightarrow & B & \text{aenc((Na,A), pk(skB))} \\ B & \rightarrow & A & \text{aenc((Na,Nb), pk(skA))} \\ A & \rightarrow & I & \text{aenc(Nb, pk(skI))} \\ I & \rightarrow & B & \text{aenc(Nb, pk(skB))} \end{array}$$

Lowe fixes the protocol by the inclusion of Bob's identity in message 6; that is,

$$(6') \quad B \rightarrow A : \text{aenc((Na,Nb,B), pk(skA))}$$

This correction allows Alice to verify whom she is running the protocol with and prevents the attack. In the remainder of this chapter, we demonstrate how the Needham-Schroeder public key protocol can be analyzed using ProVerif with various degrees of complexity.

5.1 Simplified Needham-Schroeder protocol

We begin our study with the investigation of a simplistic variant which allows us to concentrate on the modeling process rather than the complexities of the protocol itself. Accordingly, we consider the essence of the protocol which is specified as follows:

$$\begin{array}{lcl:l} A & \rightarrow & B & \text{aenc((Na,pk(skA)), pk(skB))} \\ B & \rightarrow & A & \text{aenc((Na,Nb), pk(skA))} \\ A & \rightarrow & B & \text{aenc(Nb, pk(skB))} \end{array}$$

In this formalization, the role of the trusted key server is omitted and hence we assume that participants Alice and Bob are in possession of the necessary public keys prior to the execution of the protocol. In addition, Alice's identity is modeled using her public key.

5.1.1 Basic encoding

The declarations are standard, they specify a public channel c and constructors/destructors required to capture cryptographic primitives in the now familiar fashion:

```
free c: channel.

(* Public key encryption *)
type pkey.
type skey.

fun pk(skey): pkey.
fun aenc(bitstring, pkey): bitstring.
reduc forall x: bitstring, y: skey; adec(aenc(x, pk(y)),y) = x.

(* Signatures *)
type spkey.
type sskey.

fun spk(sskey): spkey.
fun sign(bitstring, sskey): bitstring.
reduc forall x: bitstring, y: sskey; getmess(sign(x,y)) = x.
reduc forall x: bitstring, y: sskey; checksign(sign(x,y), spk(y)) = x.
```

```
(* Shared key encryption *)
fun senc(bitstring,bitstring): bitstring.
reduc forall x: bitstring, y: bitstring; sdec(senc(x,y),y) = x.
```

Process macros for A and B can now be declared and the main process can also be specified:

```
let processA(pkB: pkey, skA: skey) =
  in(c, pkX: pkey);
  new Na: bitstring;
  out(c, aenc((Na, pk(skA)), pkX));
  in(c, m: bitstring);
  let (=Na, NX: bitstring) = adec(m, skA) in
  out(c, aenc(NX, pkX)).

let processB(pkA: pkey, skB: skey) =
  in(c, m: bitstring);
  let (NY: bitstring, pkY: pkey) = adec(m, skB) in
  new Nb: bitstring;
  out(c, aenc((NY, Nb), pkY));
  in(c, m3: bitstring);
  if Nb = adec(m3, skB) then 0.

process
  new skA: skey; let pkA = pk(skA) in out(c, pkA);
  new skB: skey; let pkB = pk(skB) in out(c, pkB);
  ( (!processA(pkB, skA)) | (!processB(pkA, skB)) )
```

The main process begins by constructing the private keys skA and skB for principals A and B respectively. The public parts pk(skA) and pk(skB) are then output on the public communication channel c, ensuring they are available to the attacker. (Observe that this is done using the handles pkA and pkB for convenience.) An unbounded number of instances of processA and processB are then instantiated (with the relevant parameters), representing A and B's willingness to participate in arbitrarily many sessions of the protocol.

We assume that Alice is willing to run the protocol with any other principal; the choice of her interlocutor will be made by the environment. This is captured by modeling the first input **in**(c, pkX: pkey) to processA as the interlocutor's public key pkX. The actual protocol then commences with Alice selecting her nonce Na, which she pairs with her identity pkA = pk(skA) and outputs the message encrypted with her interlocutor's public key pkX. Meanwhile, Bob awaits an input from his initiator; on receipt, Bob decrypts the message to recover his initiator's nonce NY and identity pkY. Bob then generates his nonce Nb and sends the message (NY,Nb) encrypted for the initiator using the key pkY. Next, if Alice believes she is talking to her interlocutor, that is, if the ciphertext she receives contains her nonce Na, then she replies with aenc(Nb, pk(skB)). (Recall that only the interlocutor who has the secret key corresponding to the public key part pkX should have been able to recover Na and hence if the ciphertext contains her nonce, then she believes authentication of her interlocutor holds.) Finally, if the ciphertext received by Bob contains his nonce Nb, then he believes that he has successfully completed the protocol with his initiator.

5.1.2 Security properties

Recall that the primary objective of the protocol is mutual authentication of the principals Alice and Bob. Accordingly, when A reaches the end of the protocol with the belief that she has done so with B, then B has indeed engaged in a session with A; and vice-versa for B. We declare the events:

- **event** beginAparam(pkey), which is used by Bob to record the belief that the initiator whose public key is supplied as a parameter has commenced a run of the protocol with him.

- **event** endAparam(pkey), which means that Alice believes she has successfully completed the protocol with Bob. This event is executed only when Alice believes she runs the protocol with Bob, that is, when pkX = pkB. Alice supplies her public key pk(skA) as the parameter.
- **event** beginBparam(pkey), which denotes Alice's intention to initiate the protocol with an interlocutor whose public key is supplied as a parameter.
- **event** endBparam(pkey), which records Bob's belief that he has completed the protocol with Alice. He supplies his public key pk(skB) as the parameter.

Intuitively, if Alice believes she has completed the protocol with Bob and hence executes the event endAparam(pk(skA)), then there should have been an earlier occurrence of the event beginAparam(pk(skA)), indicating that Bob started a session with Alice; moreover, the relationship should be injective. A similar property should hold for Bob.

In addition, we wish to test if, at the end of the protocol, the nonces Na and Nb are secret. These nonces are names created by **new** or variables such as NX and NY, while the standard secrecy queries of ProVerif deal with the secrecy of private free names. To solve this problem, we can use the following general technique: instead of directly testing the secrecy of the nonces, we use them as session keys in order to encrypt some free name and test the secrecy of that free name. For instance, in the process for Alice, we output senc(secretANa,Na) and we test the secrecy of secretANa: secretANa is secret if and only if the nonce Na that Alice has is secret. Similarly, we output senc(secretANb,NX) and we test the secrecy of secretANb: secretANb is secret if and only if NX (that is, the nonce Nb that Alice has) is secret. We proceed symmetrically for Bob using secretBNa and secretBNb. (Alternatively, we could also define a variable NaA to store the nonce Na that Alice has at the end of the protocol, and test its secrecy using the query **query secret** NaA. We can proceed similarly using NbA to store the nonce Nb on Alice's side, and NaB and NbB to store the nonces on Bob's side. This is done in the file `docs/NeedhamSchroederPK-var5.pv`.)

Observe that the use of four names secretANa, secretANb, secretBNa, secretBNb for secrecy queries allows us to analyze the precise point of failure; that is, we can study *secrecy for Alice* and *secrecy for Bob*. Moreover, we can analyze both nonces Na and Nb independently for each of Alice and Bob.

The corresponding ProVerif code annotated with events and additional code to model secrecy, along with the relevant queries, is presented as follows (file `docs/NeedhamSchroederPK-var1.pv`):

```
23 (* Authentication queries *)
24 event beginBparam(pkey).
25 event endBparam(pkey).
26 event beginAparam(pkey).
27 event endAparam(pkey).
28
29 query x: pkey; inj-event(endBparam(x)) ==> inj-event(beginBparam(x)).
30 query x: pkey; inj-event(endAparam(x)) ==> inj-event(beginAparam(x)).
31
32 (* Secrecy queries *)
33 free secretANa, secretANb, secretBNa, secretBNb: bitstring [private].
34
35 query attacker(secretANa);
36       attacker(secretANb);
37       attacker(secretBNa);
38       attacker(secretBNb).
39
40 (* Alice *)
41 let processA(pkB: pkey, skA: skey) =
42   in(c, pkX: pkey);
43   event beginBparam(pkX);
44   new Na: bitstring;
45   out(c, aenc((Na, pk(skA)), pkX));
46   in(c, m: bitstring);
```

```
47    let (=Na, NX: bitstring) = adec(m, skA) in
48    out(c, aenc(NX, pkX));
49    if pkX = pkB then
50    event endAparam(pk(skA));
51    out(c, senc(secretANa, Na));
52    out(c, senc(secretANb, NX)).
53
54  (* Bob *)
55  let processB(pkA: pkey, skB: skey) =
56    in(c, m: bitstring);
57    let (NY: bitstring, pkY: pkey) = adec(m, skB) in
58    event beginAparam(pkY);
59    new Nb: bitstring;
60    out(c, aenc((NY, Nb), pkY));
61    in(c, m3: bitstring);
62    if Nb = adec(m3, skB) then
63    if pkY = pkA then
64    event endBparam(pk(skB));
65    out(c, senc(secretBNa, NY));
66    out(c, senc(secretBNb, Nb)).
67
68  (* Main *)
69  process
70    new skA: skey; let pkA = pk(skA) in out(c, pkA);
71    new skB: skey; let pkB = pk(skB) in out(c, pkB);
72    ( (!processA(pkB, skA)) | (!processB(pkA, skB)) )
```

Analyzing the simplified Needham-Schroeder protocol. Executing the Needham-Schroeder protocol with the command `./proverif docs/NeedhamSchroederPK-var1.pv | grep "RES"` produces the output:

```
RESULT not attacker(secretANa[]) is true.
RESULT not attacker(secretANb[]) is true.
RESULT not attacker(secretBNa[]) is false.
RESULT not attacker(secretBNb[]) is false.
RESULT inj-event(endAparam(x_569)) ==> inj-event(beginAparam(x_569)) is true.
RESULT inj-event(endBparam(x_999)) ==> inj-event(beginBparam(x_999)) is false.
RESULT (even event(endBparam(x_1486)) ==> event(beginBparam(x_1486)) is false.)
```

As we would expect, this means that the authentication of B to A and secrecy for A hold; whereas authentication of A to B and secrecy for B are violated. Notice how the use of four independent queries for secrecy makes the task of evaluating the output easier. In addition, we learn

```
RESULT (even event(endBparam(x_1486)) ==> event(beginBparam(x_1486)) is false.)
```

which means that even the non-injective authentication of A to B is false; that is, Bob may end the protocol thinking he talks to Alice while Alice has never run the protocol with Bob. For the query **attacker**(secretBNa[]), ProVerif returns the following trace of an attack:

```
1  new skA creating skA_411 at {1}
2  out(c, pk(skA_411)) at {3}
3  new skB creating skB_412 at {4}
4  out(c, pk(skB_412)) at {6}
5  in(c, pk(a)) at {8} in copy a_408
6  event(beginBparam(pk(a))) at {9} in copy a_408
7  new Na creating Na_410 at {10} in copy a_408
8  out(c, aenc((Na_410,pk(skA_411)),pk(a))) at {11} in copy a_408
9  in(c, aenc((Na_410,pk(skA_411)),pk(skB_412))) at {20} in copy a_409
```

```
10 event(beginAparam(pk(skA_411))) at {22} in copy a_409
11 new Nb creating Nb_413 at {23} in copy a_409
12 out(c, aenc((Na_410,Nb_413),pk(skA_411))) at {24} in copy a_409
13 in(c, aenc((Na_410,Nb_413),pk(skA_411))) at {12} in copy a_408
14 out(c, aenc(Nb_413,pk(a))) at {14} in copy a_408
15 in(c, aenc(Nb_413,pk(skB_412))) at {25} in copy a_409
16 event(endBparam(pk(skB_412))) at {28} in copy a_409
17 out(c, senc(secretBNa,Na_410)) at {29} in copy a_409
18 out(c, senc(secretBNb,Nb_413)) at {30} in copy a_409
19 The attacker has the message secretBNa.
```

This trace corresponds to Lowe's attack. The first two **new** and outputs correspond to the creation of the secret keys and outputs of the public keys of A and B in the main process. Next, processA starts, inputting the public key pk(a) of its interlocutor: a has been generated by the attacker, so this interlocutor is dishonest. A then sends the first message of the protocol aenc((Na_410,pk(skA_411)),pk(a)) (Line 8 of the trace). This message is received by B after having been decrypted and reencrypted under pkB by the attacker. It looks like a message for a session between A and B, B replies with aenc((Na_410,Nb_413),pk(skA_411)) which is then received by A. A replies with aenc(Nb_413,pk(a)). This message is again received by B after having been decrypted and reencrypted under pkB by the attacker. B has then apparently concluded a session with A, so it sends senc(secretBNa,Na_410). The attacker has obtained Na_410 by decrypting the message aenc((Na_410,pk(skA_411)),pk(a)) (sent at Line 8 of the trace), so it can compute secretBNa, thus breaking secrecy. The traces found for the other queries are similar.

5.2 Full Needham-Schroeder protocol

In this section, we will present a model of the full protocol and will demonstrate the use of some ProVerif features. (A more generic model is presented in Section 5.3.) In this formalization, we preserve the types of the Needham-Schroeder protocol more closely. In particular, we model the type *nonce* (rather than bitstring) and we introduce the type *host* for participants identities. Accordingly, we make use of type conversion where necessary. Since the modeling process should now be familiar, we present the complete encoding, which can be found in the file `docs/NeedhamSchroederPK-var2.pv`, and then discuss particular aspects.

```
free c: channel.

(* Public key encryption *)
type pkey.
type skey.

fun pk(skey): pkey.
fun aenc(bitstring, pkey): bitstring.
reduc forall x: bitstring, y: skey; adec(aenc(x, pk(y)),y) = x.

(* Signatures *)
type spkey.
type sskey.

fun spk(sskey): spkey.
fun sign(bitstring, sskey): bitstring.
reduc forall x: bitstring, y: sskey; getmess(sign(x,y)) = x.
reduc forall x: bitstring, y: sskey; checksign(sign(x,y), spk(y)) = x.

(* Shared key encryption *)
type nonce.

```

```
23 fun senc(bitstring,nonce): bitstring.
24 reduc forall x: bitstring, y: nonce; sdec(senc(x,y),y) = x.
25
26 (* Type converter *)
27 fun nonce_to_bitstring(nonce): bitstring [data,typeConverter].
28
29 (* Two honest host names A and B *)
30 type host.
31 free A, B: host.
32
33 (* Key table *)
34 table keys(host, pkey).
35
36 (* Authentication queries *)
37 event beginBparam(host).
38 event endBparam(host).
39 event beginAparam(host).
40 event endAparam(host).
41
42 query x: host; inj-event(endBparam(x)) ==> inj-event(beginBparam(x)).
43 query x: host; inj-event(endAparam(x)) ==> inj-event(beginAparam(x)).
44
45 (* Secrecy queries *)
46 free secretANa, secretANb, secretBNa, secretBNb: bitstring [private].
47
48 query attacker(secretANa);
49       attacker(secretANb);
50       attacker(secretBNa);
51       attacker(secretBNb).
52
53 (* Alice *)
54 let processA(pkS: spkey, skA: skey, skB: skey) =
55   in(c, hostX: host);
56   event beginBparam(hostX);
57   out(c, (A, hostX));                                  (* msg 1 *)
58   in(c, ms: bitstring);                                (* msg 2 *)
59   let (pkX: pkey, =hostX) = checksign(ms, pkS) in
60   new Na: nonce;
61   out(c, aenc((Na, A), pkX));                          (* msg 3 *)
62   in(c, m: bitstring);                                 (* msg 6 *)
63   let (=Na, NX: nonce) = adec(m, skA) in
64   out(c, aenc(nonce_to_bitstring(NX), pkX));           (* msg 7 *)
65   if hostX = B then
66   event endAparam(A);
67   out(c, senc(secretANa, Na));
68   out(c, senc(secretANb, NX)).
69
70 (* Bob *)
71 let processB(pkS: spkey, skA: skey, skB: skey) =
72   in(c, m: bitstring);                                 (* msg 3 *)
73   let (NY: nonce, hostY: host) = adec(m, skB) in
74   event beginAparam(hostY);
75   out(c, (B, hostY));                                  (* msg 4 *)
76   in(c,ms: bitstring);                                 (* msg 5 *)
77   let (pkY: pkey,=hostY) = checksign(ms, pkS) in
```

```
  new Nb: nonce;
  out(c, aenc((NY, Nb), pkY));                    (* msg 6 *)
  in(c, m3: bitstring);                           (* msg 7 *)
  if nonce_to_bitstring(Nb) = adec(m3, skB) then
  if hostY = A then
  event endBparam(B);
  out(c, senc(secretBNa, NY));
  out(c, senc(secretBNb, Nb)).

(* Trusted key server *)
let processS(skS: sskey) =
  in(c,(a: host, b: host));
  get keys(=b, sb) in
  out(c,sign((sb,b), skS)).

(* Key registration *)
let processK =
  in(c, (h: host, k: pkey));
  if h <> A && h <> B then insert keys(h,k).

(* Main *)
process
  new skA: skey; let pkA = pk(skA) in out(c, pkA); insert keys(A, pkA);
  new skB: skey; let pkB = pk(skB) in out(c, pkB); insert keys(B, pkB);
  new skS: sskey; let pkS = spk(skS) in out(c, pkS);
  ( (!processA(pkS, skA, skB)) | (!processB(pkS, skA, skB)) |
    (!processS(skS)) | (!processK) )
```

This process uses a key table in order to relate host names and their public keys. The key table is declared by **table** keys(host, pkey). Keys are inserted in the key table in the main process (for the honest hosts A and B, by **insert** keys(A, pkA) and **insert** keys(B, pkB)) and in a key registration process processK for dishonest hosts. The key server processS looks up the key corresponding to host b by **get** keys(=b, sb) in order to build the corresponding certificate. Since Alice is willing to run the protocol with any other participant and she will request her interlocutor's public key from the key server, we must permit the attacker to register keys with the trusted key server (that is, insert keys into the key table). This behavior is captured by the key registration process processK. Observe that the conditional **if** h <> A && h <> B **then** prevents the attacker from changing the keys belonging to Alice and Bob. (Recall that when several records are matched by a get query, then one possibility is chosen, but ProVerif considers all possibilities when reasoning; without the conditional, the attacker can therefore effectively change the keys belonging to Alice and Bob.)

Evaluating security properties of the Needham-Schroeder protocol. Once again ProVerif is able to conclude that authentication of B to A and secrecy for A hold, whereas authentication of A to B and secrecy for B are violated. We omit analyzing the output produced by ProVerif and leave this as an exercise for the reader.

5.3 Generalized Needham-Schroeder protocol

In the previous section, we considered an undesirable restriction on the participants; namely that the initiator was played by Alice using the public key pk(skA) and the responder played by Bob using the public key pk(skB). In this section, we generalize our encoding. Additionally, we also model authentication as full agreement, that is, agreement on all protocol parameters. The reader will also notice that we use encrypt and decrypt instead of aenc and adec, and sencrypt and sdecrypt instead of senc and sdec. The following script can be found in the file `docs/NeedhamSchroederPK-var3.pv`.

```
1  (* Loops if types are ignored *)
2  set ignoreTypes = false.
3
4  free c: channel.
5
6  type host.
7  type nonce.
8  type pkey.
9  type skey.
10 type spkey.
11 type sskey.
12
13 fun nonce_to_bitstring(nonce): bitstring [data,typeConverter].
14
15 (* Public key encryption *)
16 fun pk(skey): pkey.
17 fun encrypt(bitstring, pkey): bitstring.
18 reduc forall x: bitstring, y: skey; decrypt(encrypt(x,pk(y)),y) = x.
19
20 (* Signatures *)
21 fun spk(sskey): spkey.
22 fun sign(bitstring, sskey): bitstring.
23 reduc forall m: bitstring, k: sskey; getmess(sign(m,k)) = m.
24 reduc forall m: bitstring, k: sskey; checksign(sign(m,k), spk(k)) = m.
25
26 (* Shared key encryption *)
27 fun sencrypt(bitstring,nonce): bitstring.
28 reduc forall x: bitstring, y: nonce; sdecrypt(sencrypt(x,y),y) = x.
29
30 (* Secrecy assumptions *)
31 not attacker(new skA).
32 not attacker(new skB).
33 not attacker(new skS).
34
35 (* 2 honest host names A and B *)
36 free A, B: host.
37
38 table keys(host, pkey).
39
40 (* Queries *)
41 free secretANa, secretANb, secretBNa, secretBNb: bitstring [private].
42 query attacker(secretANa);
43       attacker(secretANb);
44       attacker(secretBNa);
45       attacker(secretBNb).
46
47 event beginBparam(host, host).
48 event endBparam(host, host).
49 event beginAparam(host, host).
50 event endAparam(host, host).
51 event beginBfull(host, host, pkey, pkey, nonce, nonce).
52 event endBfull(host, host, pkey, pkey, nonce, nonce).
53 event beginAfull(host, host, pkey, pkey, nonce, nonce).
54 event endAfull(host, host, pkey, pkey, nonce, nonce).
55
```

```
56  query x: host, y: host;
57    inj-event(endBparam(x,y)) ==> inj-event(beginBparam(x,y)).
58
59  query x1: host, x2: host, x3: pkey, x4: pkey, x5: nonce, x6: nonce;
60    inj-event(endBfull(x1,x2,x3,x4,x5,x6))
61        ==> inj-event(beginBfull(x1,x2,x3,x4,x5,x6)).
62
63  query x: host, y: host;
64    inj-event(endAparam(x,y)) ==> inj-event(beginAparam(x,y)).
65
66  query x1: host, x2: host, x3: pkey, x4: pkey, x5: nonce, x6: nonce;
67    inj-event(endAfull(x1,x2,x3,x4,x5,x6))
68        ==> inj-event(beginAfull(x1,x2,x3,x4,x5,x6)).
69
70  (* Role of the initiator with identity xA and secret key skxA *)
71  let processInitiator(pkS: spkey, skA: skey, skB: skey) =
72    (* The attacker starts the initiator by choosing identity xA,
73       and its interlocutor xB0.
74       We check that xA is honest (i.e. is A or B)
75       and get its corresponding key. *)
76    in(c, (xA: host, hostX: host));
77    if xA = A || xA = B then
78    let skxA = if xA = A then skA else skB in
79    let pkxA = pk(skxA) in
80    (* Real start of the role *)
81    event beginBparam(xA, hostX);
82    (* Message 1: Get the public key certificate for the other host *)
83    out(c, (xA, hostX));
84    (* Message 2 *)
85    in(c, ms: bitstring);
86    let (pkX: pkey, =hostX) = checksign(ms,pkS) in
87    (* Message 3 *)
88    new Na: nonce;
89    out(c, encrypt((Na, xA), pkX));
90    (* Message 6 *)
91    in(c, m: bitstring);
92    let (=Na, NX2: nonce) = decrypt(m, skxA) in
93    event beginBfull(xA, hostX, pkX, pkxA, Na, NX2);
94    (* Message 7 *)
95    out(c, encrypt(nonce_to_bitstring(NX2), pkX));
96    (* OK *)
97    if hostX = B || hostX = A then
98    event endAparam(xA, hostX);
99    event endAfull(xA, hostX, pkX, pkxA, Na, NX2);
100   out(c, sencrypt(secretANa, Na));
101   out(c, sencrypt(secretANb, NX2)).
102
103 (* Role of the responder with identity xB and secret key skxB *)
104 let processResponder(pkS: spkey, skA: skey, skB: skey) =
105   (* The attacker starts the responder by choosing identity xB.
106      We check that xB is honest (i.e. is A or B). *)
107   in(c, xB: host);
108   if xB = A || xB = B then
109   let skxB = if xB = A then skA else skB in
110   let pkxB = pk(skxB) in
```

```
111     (* Real start of the role *)
112     (* Message 3 *)
113     in(c, m: bitstring);
114     let (NY: nonce, hostY: host) = decrypt(m, skxB) in
115     event beginAparam(hostY, xB);
116     (* Message 4: Get the public key certificate for the other host *)
117     out(c, (xB, hostY));
118     (* Message 5 *)
119     in(c,ms: bitstring);
120     let (pkY: pkey,=hostY) = checksign(ms,pkS) in
121     (* Message 6 *)
122     new Nb: nonce;
123     event beginAfull(hostY, xB, pkxB, pkY, NY, Nb);
124     out(c, encrypt((NY, Nb), pkY));
125     (* Message 7 *)
126     in(c, m3: bitstring);
127     if nonce_to_bitstring(Nb) = decrypt(m3, skB) then
128     (* OK *)
129     if hostY = A || hostY = B then
130     event endBparam(hostY, xB);
131     event endBfull(hostY, xB, pkxB, pkY, NY, Nb);
132     out(c, sencrypt(secretBNa, NY));
133     out(c, sencrypt(secretBNb, Nb)).
134
135 (* Server *)
136 let processS(skS: sskey) =
137     in(c,(a: host, b: host));
138     get keys(=b, sb) in
139     out(c,sign((sb,b),skS)).
140
141 (* Key registration *)
142 let processK =
143     in(c, (h: host, k: pkey));
144     if h <> A && h <> B then insert keys(h,k).
145
146 (* Main *)
147 process
148     new skA: skey; let pkA = pk(skA) in out(c, pkA); insert keys(A, pkA);
149     new skB: skey; let pkB = pk(skB) in out(c, pkB); insert keys(B, pkB);
150     new skS: sskey; let pkS = spk(skS) in out(c, pkS);
151     (
152         (* Launch an unbounded number of sessions of the initiator *)
153         (!processInitiator(pkS, skA, skB)) |
154         (* Launch an unbounded number of sessions of the responder *)
155         (!processResponder(pkS, skA, skB)) |
156         (* Launch an unbounded number of sessions of the server *)
157         (!processS(skS)) |
158         (* Key registration process *)
159         (!processK)
160     )
```

The main novelty of this script is that it allows Alice and Bob to play both roles of the initiator and responder. To achieve this, we could simply duplicate the code, but it is possible to have more elegant encodings. Above, we consider processes processInitiator and processResponder that take as argument both skA and skB (since they can be played by Alice and Bob). Looking for instance at the initiator (Lines 71–79), the attacker first starts the initiator by sending the identity xA of the principal playing

the role of the initiator and hostX of its interlocutor. Then, we verify that the initiator is honest, and compute its secret key skxA (skA for A, skB for B) and its corresponding public key pkxA = pk(skxA). We can then run the role as expected. We proceed similarly for the responder.

Other encodings are also possible. For instance, we could define a destructor choosekey by

```
fun choosekey(host, host, host, skey, skey): skey
reduc forall x1: host, x2: host, sk1: skey, sk2: skey;
        choosekey(x1, x1, x2, sk1, sk2) = sk1
otherwise forall x1: host, x2: host, sk1: skey, sk2: skey;
        choosekey(x2, x1, x2, sk1, sk2) = sk2.
```

and let skxA be choosekey(xA, A, B, skA, skB) (if xA = A, it returns skA; if xA = B, it returns skB; otherwise, it fails). The latter encoding is perhaps less intuitive, but it avoids internal code duplication when ProVerif expands tests that appear in terms.

Three other points are worth noting:

- We use secrecy assumptions (Lines 30–33) to speed up the resolution process of ProVerif. These lines inform ProVerif that the attacker cannot have the secret keys skA, skB, skS. This information is checked by ProVerif, so that erroneous proofs cannot be obtained even with secrecy assumptions. (See also Section 6.7.2.) Lines 30–33 can be removed without changing the results, ProVerif will just be slightly slower.

- We set ignoreTypes to false (Lines 1–2). By default, ProVerif ignore all types during analysis. However, for this script, it does not terminate with this default setting. By setting ignoreTypes = false, the semantics of processes is changed to check the types. This setting makes it possible to obtain termination. The known attack against this protocol is detected, but it might happen that some type flaw attacks are undetected, when they appear when the types are not checked in processes. More details on the ignoreTypes setting can be found in Section 6.6.2.

  There are other ways of obtaining termination in this example, in particular by using a different method for relating identities and keys with two function symbols, one that maps the key to the identity, and one that maps the identity to the key. However, this method also has limitations: it does not allow the attacker to create two principals with the same key. More information on this method can be found in Section 6.7.3.

- We use two different levels of authentication: the events that end with "full" serve in proving Lowe's full agreement [Low97], that is, agreement on all parameters of the protocol (here, host names, keys, and nonces). The events that end with "param" prove agreement on the host names only.

As expected, ProVerif is able to prove the authentication of the responder and secrecy for the initiator; whereas authentication of the initiator and secrecy for the responder fail. The reader is invited to modify the protocol according to Lowe's fix and examine the results produced by ProVerif. (A script for the corrected protocol can be found in `examples/pitype/secr-auth/NeedhamSchroederPK-corr.pv`. If you installed by OPAM in the switch ⟨*switch*⟩, it is in `~/.opam/`⟨*switch*⟩`/doc/proverif/examples/pitype/secr-auth/NeedhamSchroederPK-corr.pv`. Note that the fixed protocol can be proved correct by ProVerif even when types are ignored.)

5.4 Variants of these security properties

In this section, we consider several security properties of Lowe's corrected version of the Needham-Schroeder public key protocol.

5.4.1 A variant of mutual authentication

In the previous definitions of authentication that we have considered, we require that internal parameters of the protocol (such as nonces) are the same for the initiator and for the responder. However, in the computational model, one generally uses a session identifier that is publicly computable (such as the

tuple of the messages of the protocol) as argument of events. One can also do that in ProVerif, as in the following example (file `docs/NeedhamSchroederPK-corr-mutual-auth.pv`).

```
(* Queries *)
fun messtermI(host, host): bitstring [data].
fun messtermR(host, host): bitstring [data].

event termI(host, host, bitstring).
event acceptsI(host, host, bitstring).
event acceptsR(host, host, bitstring).
event termR(host, host, bitstring).

query x: host, m: bitstring;
      inj-event(termI(x,B,m)) ==> inj-event(acceptsR(x,B,m)).
query x: host, m: bitstring;
      inj-event(termR(A,x,m)) ==> inj-event(acceptsI(A,x,m)).

(* Role of the initiator with identity xA and secret key skxA *)
let processInitiator(pkS: spkey, skA: skey, skB: skey) =
  (* The attacker starts the initiator by choosing identity xA,
     and its interlocutor xB0.
     We check that xA is honest (i.e. is A or B)
     and get its corresponding key.
  *)
  in(c, (xA: host, hostX: host));
  if xA = A || xA = B then
  let skxA = if xA = A then skA else skB in
  let pkxA = pk(skxA) in
  (* Real start of the role *)
  (* Message 1: Get the public key certificate for the other host *)
  out(c, (xA, hostX));
  (* Message 2 *)
  in(c, ms: bitstring);
  let (pkX: pkey, =hostX) = checksign(ms,pkS) in
  (* Message 3 *)
  new Na: nonce;
  let m3 = encrypt((Na, xA), pkX) in
  out(c, m3);
  (* Message 6 *)
  in(c, m: bitstring);
  let (=Na, NX2: nonce, =hostX) = decrypt(m, skA) in
  let m7 = encrypt(nonce_to_bitstring(NX2), pkX) in
  event termI(xA, hostX, (m3, m));
  event acceptsI(xA, hostX, (m3, m, m7));
  (* Message 7 *)
  out(c, (m7, messtermI(xA, hostX))).

(* Role of the responder with identity xB and secret key skxB *)
let processResponder(pkS: spkey, skA: skey, skB: skey) =
  (* The attacker starts the responder by choosing identity xB.
     We check that xB is honest (i.e. is A or B). *)
  in(c, xB: host);
  if xB = A || xB = B then
  let skxB = if xB = A then skA else skB in
  let pkxB = pk(skxB) in
  (* Real start of the role *)
```

```
54    (* Message 3 *)
55    in(c, m: bitstring);
56    let (NY: nonce, hostY: host) = decrypt(m, skxB) in
57    (* Message 4: Get the public key certificate for the other host *)
58    out(c, (xB, hostY));
59    (* Message 5 *)
60    in(c,ms: bitstring);
61    let (pkY: pkey,=hostY) = checksign(ms,pkS) in
62    (* Message 6 *)
63    new Nb: nonce;
64    let m6 = encrypt((NY, Nb, xB), pkY) in
65    event acceptsR(hostY, xB, (m, m6));
66    out(c, m6);
67    (* Message 7 *)
68    in(c, m3: bitstring);
69    if nonce_to_bitstring(Nb) = decrypt(m3, skB) then
70    event termR(hostY, xB, (m, m6, m3));
71    out(c, messtermR(hostY, xB)).
72
73  (* Server *)
74  let processS(skS: sskey) =
75    in(c,(a: host, b: host));
76    get keys(=b, sb) in
77    out(c,sign((sb,b),skS)).
78
79  (* Key registration *)
80  let processK =
81    in(c, (h: host, k: pkey));
82    if h <> A && h <> B then insert keys(h,k).
83
84  (* Start process *)
85  process
86    new skA: skey; let pkA = pk(skA) in out(c, pkA); insert keys(A, pkA);
87    new skB: skey; let pkB = pk(skB) in out(c, pkB); insert keys(B, pkB);
88    new skS: sskey; let pkS = spk(skS) in out(c, pkS);
89    (
90      (* Launch an unbounded number of sessions of the initiator *)
91      (!processInitiator(pkS, skA, skB)) |
92      (* Launch an unbounded number of sessions of the responder *)
93      (!processResponder(pkS, skA, skB)) |
94      (* Launch an unbounded number of sessions of the server *)
95      (!processS(skS)) |
96      (* Key registration process *)
97      (!processK)
98    )
```

The query

```
10  query x: host, m: bitstring;
11        inj-event(termI(x,B,m)) ==> inj-event(acceptsR(x,B,m)).
```

corresponds to the authentication of the responder B to the initiator x: when the initiator x terminates a session apparently with B (event termI(x,B,m), executed at Line 40, when the initiator terminates, after receiving its last message, message 6), the responder B has accepted with x (event acceptsR(x,B,m), executed at Line 65, when the responder accepts, just before sending message 6). We use a fixed value B for the name of the responder, and not a variable, because if a variable were used, the initiator might run a session with a dishonest participant included in the attacker, and in this case, it is perfectly ok that

the event acceptsR is not executed. Since the initiator is executed with identities A and B, x is either A or B, so the query above proves correct authentication of the responder B to the initiator x when x is A and when it is B. The same property for the responder A holds by symmetry, swapping A and B.

Similarly, the query

```
12 query x: host, m: bitstring;
13        inj-event(termR(A,x,m)) ==> inj-event(acceptsI(A,x,m)).
```

corresponds to the authentication of the initiator A to the responder x: when the responder x terminates a session apparently with A (event termR(A,x,m), executed at Line 70, when the responder terminates, after receiving its last message, message 7), the initiator A has accepted with x (event acceptsI(A,x,m), executed at Line 41, when the initiator accepts, just before sending message 7).

The position of events follows Figure 3.4. The events termR and acceptsI take as arguments the host names of the initiator and the responder, and the tuples of messages exchanged between the initiator and the responder. (Messages sent to or received from the server to obtain the certificates are ignored.) Because the last message is from the initiator to the responder, that message is not known to the responder when it accepts, so that message is omitted from the arguments of the events acceptsR and termI.

5.4.2 Authenticated key exchange

In the computational model, the security of an authenticated key exchange protocol is typically shown by proving, in addition to mutual authentication, that the exchanged key is indistinguishable from a random key. More precisely, in the real-or-random model [AFP06], one allows the attacker to perform several test queries, which either return the real key or a fresh random key, and these two cases must be indistinguishable. When the test query is performed on a session between a honest and a dishonest participant, the returned key is always the real one. When the test query is performed several times on the same session, or on the partner session (that is, the session of the interlocutor that has the same session identifier), it returns the same key (whether real or random). Taking into account partnering in the definition of test queries is rather tricky, so we have developed an alternative characterization that does not require partnering [Bla07].

- We use events similar to those for mutual authentication, except that termR and acceptsI take the exchanged key as an additional argument. We prove the following properties:

```
query x: host, m: bitstring;
      inj-event(termI(x,B,m)) ==> inj-event(acceptsR(x,B,m)).
query x: host, k:nonce, m: bitstring;
      inj-event(termR(A,x,k,m)) ==> inj-event(acceptsI(A,x,k,m)).
query x: host, k:nonce, k':nonce, m: bitstring;
      event(termR(A,x,k,m)) && event(acceptsI(A,x,k',m)) ==> k = k'.
```

- When the initiator or the responder execute a session with a dishonest participant, they output the exchanged key. (This key is also output by the test queries in this case.) We show the secrecy of the keys established by the initiator when it runs sessions with a honest responder, in the sense that these keys are indistinguishable from independent random numbers.

The first two correspondences imply mutual authentication. The real-or-random indistinguishability of the key is obtained by combining the last two correspondences with the secrecy of the initiator's key. Intuitively, the correspondences allow us to show that each responder's key in a session with a honest initiator is in fact also an initiator's key (which we can find by looking for the same session identifier), so showing that the initiator's key cannot be distinguished from independent random numbers is sufficient to show the secrecy of the key.

Outputting the exchanged key in a session with a dishonest interlocutor allows to detect Unknown Key Share (UKS) attacks [DvOW92], in which an initiator A believes he shares a key with a responder B, but B believes he shares that key with a dishonest C. This key is then output to the attacker, so the secrecy of the initiator's key is broken. However, bilateral UKS attacks [CT08], in which A shares a key

with a dishonest C and B shares the same key with a dishonest D, may remain undetected under this definition of key exchange. These attacks can be detected by testing the following correspondence:

```
query x: host, y:host, x':host, y':host, k:nonce, k':nonce,
  m: bitstring, m':bitstring;
  event(termR(x,y,k,m)) && event(acceptsI(x',y',k,m')) ==> x = x' && y = y'.
```

to verify that, if two sessions terminate with the same key, then they are between the same hosts (and we could additionally verify m = m' to make sure that these sessions have the same session identifiers).

The following script aims at verifying this notion of authenticated key exchange, assuming that the exchanged key is Na (file `docs/NeedhamSchroederPK-corr-ake.pv`).

```
1  (* Queries *)
2  free secretA: bitstring [private].
3  query attacker(secretA).
4
5  fun messtermI(host, host): bitstring [data].
6  fun messtermR(host, host): bitstring [data].
7
8  event termI(host, host, bitstring).
9  event acceptsI(host, host, nonce, bitstring).
10 event acceptsR(host, host, bitstring).
11 event termR(host, host, nonce, bitstring).
12
13 query x: host, m: bitstring;
14        inj-event(termI(x,B,m)) ==> inj-event(acceptsR(x,B,m)).
15 query x: host, k:nonce, m: bitstring;
16        inj-event(termR(A,x,k,m)) ==> inj-event(acceptsI(A,x,k,m)).
17
18 query x: host, k:nonce, k':nonce, m: bitstring;
19        event(termR(A,x,k,m)) && event(acceptsI(A,x,k',m)) ==> k = k'.
20
21 (* Query for detecting bilateral UKS attacks *)
22 query x: host, y:host, x':host, y':host, k:nonce, k':nonce,
23   m: bitstring, m':bitstring;
24   event(termR(x,y,k,m)) && event(acceptsI(x',y',k,m')) ==> x = x' && y = y'.
25
26 (* Role of the initiator with identity xA and secret key skxA *)
27 let processInitiator(pkS: spkey, skA: skey, skB: skey) =
28   (* The attacker starts the initiator by choosing identity xA,
29      and its interlocutor xB0.
30      We check that xA is honest (i.e. is A or B)
31      and get its corresponding key.
32   *)
33   in(c, (xA: host, hostX: host));
34   if xA = A || xA = B then
35   let skxA = if xA = A then skA else skB in
36   let pkxA = pk(skxA) in
37   (* Real start of the role *)
38   (* Message 1: Get the public key certificate for the other host *)
39   out(c, (xA, hostX));
40   (* Message 2 *)
41   in(c, ms: bitstring);
42   let (pkX: pkey, =hostX) = checksign(ms,pkS) in
43   (* Message 3 *)
44   new Na: nonce;
45   let m3 = encrypt((Na, xA), pkX) in
```

```
46    out(c, m3);
47    (* Message 6 *)
48    in(c, m: bitstring);
49    let (=Na, NX2: nonce, =hostX) = decrypt(m, skA) in
50    let m7 = encrypt(nonce_to_bitstring(NX2), pkX) in
51    event termI(xA, hostX, (m3, m));
52    event acceptsI(xA, hostX, Na, (m3, m, m7));
53    (* Message 7 *)
54    if hostX = A || hostX = B then
55    (
56       out(c, sencrypt(secretA, Na));
57       out(c, (m7, messtermI(xA, hostX)))
58    )
59    else
60    (
61       out(c, Na);
62       out(c, (m7, messtermI(xA, hostX)))
63    ).
64
65  (* Role of the responder with identity xB and secret key skxB *)
66  let processResponder(pkS: spkey, skA: skey, skB: skey) =
67    (* The attacker starts the responder by choosing identity xB.
68       We check that xB is honest (i.e. is A or B). *)
69    in(c, xB: host);
70    if xB = A || xB = B then
71    let skxB = if xB = A then skA else skB in
72    let pkxB = pk(skxB) in
73    (* Real start of the role *)
74    (* Message 3 *)
75    in(c, m: bitstring);
76    let (NY: nonce, hostY: host) = decrypt(m, skxB) in
77    (* Message 4: Get the public key certificate for the other host *)
78    out(c, (xB, hostY));
79    (* Message 5 *)
80    in(c,ms: bitstring);
81    let (pkY: pkey,=hostY) = checksign(ms,pkS) in
82    (* Message 6 *)
83    new Nb: nonce;
84    let m6 = encrypt((NY, Nb, xB), pkY) in
85    event acceptsR(hostY, xB, (m, m6));
86    out(c, m6);
87    (* Message 7 *)
88    in(c, m3: bitstring);
89    if nonce_to_bitstring(Nb) = decrypt(m3, skB) then
90    event termR(hostY, xB, NY, (m, m6, m3));
91    if hostY = A || hostY = B then
92       out(c, messtermR(hostY, xB))
93    else
94    (
95       out(c, NY);
96       out(c, messtermR(hostY, xB))
97    ).
98
99  (* Server *)
100 let processS(skS: sskey) =
```

```
  in(c,(a: host, b: host));
  get keys(=b, sb) in
  out(c,sign((sb,b),skS)).

(* Key registration *)
let processK =
  in(c, (h: host, k: pkey));
  if h <> A && h <> B then insert keys(h,k).

(* Start process *)
process
  new skA: skey; let pkA = pk(skA) in out(c, pkA); insert keys(A, pkA);
  new skB: skey; let pkB = pk(skB) in out(c, pkB); insert keys(B, pkB);
  new skS: sskey; let pkS = spk(skS) in out(c, pkS);
  (
    (* Launch an unbounded number of sessions of the initiator *)
    (!processInitiator(pkS, skA, skB)) |
    (* Launch an unbounded number of sessions of the responder *)
    (!processResponder(pkS, skA, skB)) |
    (* Launch an unbounded number of sessions of the server *)
    (!processS(skS)) |
    (* Key registration process *)
    (!processK)
  )
```

ProVerif finds a bilateral UKS attack: if C as responder runs a session with A, it gets Na, then D as initiator can use the same nonce Na in a session with responder B, thus obtaining two sessions, between A and C and between D and B, that share the same key Na. (Such an attack appears more generally when the key is determined by a single participant of the protocol.) The other properties are proved by ProVerif.

The above script verifies syntactic secrecy of the initiator's key Na. To be even closer to the computational definition, we can verify its secrecy using the real-or-random secrecy notion (page 58), as in the following script (file `docs/NeedhamSchroederPK-corr-ake-RoR.pv`):

```
(* Termination messages *)
fun messtermI(host, host): bitstring [data].
fun messtermR(host, host): bitstring [data].

set ignoreTypes = false.

(* Role of the initiator with identity xA and secret key skxA *)
let processInitiator(pkS: spkey, skA: skey, skB: skey) =
  (* The attacker starts the initiator by choosing identity xA,
     and its interlocutor xB0.
     We check that xA is honest (i.e. is A or B)
     and get its corresponding key.
  *)
  in(c, (xA: host, hostX: host));
  if xA = A || xA = B then
  let skxA = if xA = A then skA else skB in
  let pkxA = pk(skxA) in
  (* Real start of the role *)
  (* Message 1: Get the public key certificate for the other host *)
  out(c, (xA, hostX));
  (* Message 2 *)
  in(c, ms: bitstring);
```

```
23    let (pkX: pkey, =hostX) = checksign(ms,pkS) in
24    (* Message 3 *)
25    new Na: nonce;
26    let m3 = encrypt((Na, xA), pkX) in
27    out(c, m3);
28    (* Message 6 *)
29    in(c, m: bitstring);
30    let (=Na, NX2: nonce, =hostX) = decrypt(m, skA) in
31    let m7 = encrypt(nonce_to_bitstring(NX2), pkX) in
32    (* Message 7 *)
33    if hostX = A || hostX = B then
34    (
35       new random: nonce;
36       out(c, choice[Na, random]);
37       out(c, (m7, messtermI(xA, hostX)))
38    )
39    else
40    (
41       out(c, Na);
42       out(c, (m7, messtermI(xA, hostX)))
43    ).
44
45 (* Role of the responder with identity xB and secret key skxB *)
46 let processResponder(pkS: spkey, skA: skey, skB: skey) =
47    (* The attacker starts the responder by choosing identity xB.
48       We check that xB is honest (i.e. is A or B). *)
49    in(c, xB: host);
50    if xB = A || xB = B then
51    let skxB = if xB = A then skA else skB in
52    let pkxB = pk(skxB) in
53    (* Real start of the role *)
54    (* Message 3 *)
55    in(c, m: bitstring);
56    let (NY: nonce, hostY: host) = decrypt(m, skxB) in
57    (* Message 4: Get the public key certificate for the other host *)
58    out(c, (xB, hostY));
59    (* Message 5 *)
60    in(c,ms: bitstring);
61    let (pkY: pkey,=hostY) = checksign(ms,pkS) in
62    (* Message 6 *)
63    new Nb: nonce;
64    let m6 = encrypt((NY, Nb, xB), pkY) in
65    out(c, m6);
66    (* Message 7 *)
67    in(c, m3: bitstring);
68    if nonce_to_bitstring(Nb) = decrypt(m3, skB) then
69    if hostY = A || hostY = B then
70       out(c, messtermR(hostY, xB))
71    else
72    (
73       out(c, NY);
74       out(c, messtermR(hostY, xB))
75    ).
76
77 (* Server *)
```

```
78  let processS(skS: sskey) =
79    in(c,(a: host, b: host));
80    get keys(=b, sb) in
81    out(c,sign((sb,b),skS)).
82
83  (* Key registration *)
84  let processK =
85    in(c, (h: host, k: pkey));
86    if h <> A && h <> B then insert keys(h,k).
87
88  (* Start process *)
89  process
90    new skA: skey; let pkA = pk(skA) in out(c, pkA); insert keys(A, pkA);
91    new skB: skey; let pkB = pk(skB) in out(c, pkB); insert keys(B, pkB);
92    new skS: sskey; let pkS = spk(skS) in out(c, pkS);
93    (
94      (* Launch an unbounded number of sessions of the initiator *)
95      (!processInitiator(pkS, skA, skB)) |
96      (* Launch an unbounded number of sessions of the responder *)
97      (!processResponder(pkS, skA, skB)) |
98      (* Launch an unbounded number of sessions of the server *)
99      (!processS(skS)) |
100     (* Key registration process *)
101     (!processK)
102   )
```

Line 36 outputs either the real key Na or a fresh random key, and the goal is to prove that the attacker cannot distinguish these two situations. In order to obtain termination, we require that all code including the attacker be well-typed (Line 5). This prevents in particular the generation of certificates in which the host names are themselves nested signatures of unbounded depth. Unfortunately, ProVerif finds a false attack in which the output key is used to build message 3 (either encrypt((Na, A), pkB) or encrypt((random, A), pkB)), send it to the responder, which replies with message 6 (that is, encrypt((Na, Nb, A), pkA) or encrypt((random, Nb, A), pkA)), which is accepted by the initiator if and only if the key is the real key Na.

A similar verification can be done with other possible keys (for instance, Nb, h(Na), h(Nb), h(Na,Nb) where h is a hash function). We leave these verifications to the reader and just note that the false attack above disappears for the key h(Na) (but we still have to restrict ourselves to a well-typed attacker). In order to obtain this result, a trick is necessary: if random is generated at the end of the protocol, ProVerif represents it internally as a function of the previously received messages, including message 6. This leads to a false attack in which two different values of random (generated after receiving different messages 6) are associated to the same Na. This false attack can be eliminated by moving the generation of random just after the generation of Na.

5.4.3 Full ordering of the messages

We can also show that, if a responder terminates the protocol with a honest initiator, then all messages of the protocol between the initiator and the responder have been exchanged in the right order. (We ignore messages sent to or received from the server.) This is shown in the following script (file `docs/NeedhamSchroederPK-corr-all-messages.pv`).

```
1  (* Queries *)
2  event endB(host, host, pkey, pkey, nonce, nonce).
3  event e3(host, host, pkey, pkey, nonce, nonce).
4  event e2(host, host, pkey, pkey, nonce, nonce).
5  event e1(host, host, pkey, pkey, nonce).
6
```

```
7  query y: host, pkx: pkey, pky: pkey, nx: nonce, ny: nonce;
8        inj-event(endB(A, y, pkx, pky, nx, ny)) ==>
9        (inj-event(e3(A, y, pkx, pky, nx, ny)) ==>
10       (inj-event(e2(A, y, pkx, pky, nx, ny)) ==>
11        inj-event(e1(A, y, pkx, pky, nx)))).
12
13 (* Role of the initiator with identity xA and secret key skxA *)
14 let processInitiator(pkS: spkey, skA: skey, skB: skey) =
15   (* The attacker starts the initiator by choosing identity xA,
16      and its interlocutor xB0.
17      We check that xA is honest (i.e. is A or B)
18      and get its corresponding key.
19   *)
20   in(c, (xA: host, hostX: host));
21   if xA = A || xA = B then
22   let skxA = if xA = A then skA else skB in
23   let pkxA = pk(skxA) in
24   (* Real start of the role *)
25   (* Message 1: Get the public key certificate for the other host *)
26   out(c, (xA, hostX));
27   (* Message 2 *)
28   in(c, ms: bitstring);
29   let (pkX: pkey, =hostX) = checksign(ms,pkS) in
30   (* Message 3 *)
31   new Na: nonce;
32   event e1(xA, hostX, pkxA, pkX, Na);
33   out(c, encrypt((Na, xA), pkX));
34   (* Message 6 *)
35   in(c, m: bitstring);
36   let (=Na, NX2: nonce, =hostX) = decrypt(m, skA) in
37   let m7 = encrypt(nonce_to_bitstring(NX2), pkX) in
38   event e3(xA, hostX, pkxA, pkX, Na, NX2);
39   (* Message 7 *)
40   out(c, m7).
41
42 (* Role of the responder with identity xB and secret key skxB *)
43 let processResponder(pkS: spkey, skA: skey, skB: skey) =
44   (* The attacker starts the responder by choosing identity xB.
45      We check that xB is honest (i.e. is A or B). *)
46   in(c, xB: host);
47   if xB = A || xB = B then
48   let skxB = if xB = A then skA else skB in
49   let pkxB = pk(skxB) in
50   (* Real start of the role *)
51   (* Message 3 *)
52   in(c, m: bitstring);
53   let (NY: nonce, hostY: host) = decrypt(m, skxB) in
54   (* Message 4: Get the public key certificate for the other host *)
55   out(c, (xB, hostY));
56   (* Message 5 *)
57   in(c,ms: bitstring);
58   let (pkY: pkey,=hostY) = checksign(ms,pkS) in
59   (* Message 6 *)
60   new Nb: nonce;
61   event e2(hostY, xB, pkY, pkxB, NY, Nb);
```

```
62    out(c, encrypt((NY, Nb, xB), pkY));
63    (* Message 7 *)
64    in(c, m3: bitstring);
65    if nonce_to_bitstring(Nb) = decrypt(m3, skB) then
66    event endB(hostY, xB, pkY, pkxB, NY, Nb).
67
68  (* Server *)
69  let processS(skS: sskey) =
70    in(c,(a: host, b: host));
71    get keys(=b, sb) in
72    out(c,sign((sb,b),skS)).
73
74  (* Key registration *)
75  let processK =
76    in(c, (h: host, k: pkey));
77    if h <> A && h <> B then insert keys(h,k).
78
79  (* Start process *)
80  process
81    new skA: skey; let pkA = pk(skA) in out(c, pkA); insert keys(A, pkA);
82    new skB: skey; let pkB = pk(skB) in out(c, pkB); insert keys(B, pkB);
83    new skS: sskey; let pkS = spk(skS) in out(c, pkS);
84    (
85      (* Launch an unbounded number of sessions of the initiator *)
86      (!processInitiator(pkS, skA, skB)) |
87      (* Launch an unbounded number of sessions of the responder *)
88      (!processResponder(pkS, skA, skB)) |
89      (* Launch an unbounded number of sessions of the server *)
90      (!processS(skS)) |
91      (* Key registration process *)
92      (!processK)
93    )
```

The event endB (Line 66) means that the responder has completed the protocol, e3 (Line 38) that the initiator received message 6 and sent message 7, e2 (Line 61) that the responder received message 3 and sent message 6, e1 (Line 32) that the initiator sent message 3. These events take as arguments all parameters of the protocol: the host names, their public keys, and the nonces, except e1 which cannot take Nb as argument since it has not been chosen yet when e1 is executed. We prove the correspondence

```
inj-event(endB(A, y, pkx, pky, nx, ny)) ==>
(inj-event(e3(A, y, pkx, pky, nx, ny)) ==>
(inj-event(e2(A, y, pkx, pky, nx, ny)) ==>
 inj-event(e1(A, y, pkx, pky, nx)))).
```

Chapter 6

Advanced reference

This chapter introduces ProVerif's advanced capabilities. We provide the complete grammar in Appendix A.

6.1 Proving correspondence queries by induction

6.1.1 Single query

Consider a correspondence query F ==> F' and a process P. As mentioned in Sections 3.2.2 and 4.3.1, to prove that P satisfies the query F ==> F', ProVerif needs to show that, for all traces of P, if F *was executed in the trace, then* F' *was also executed in the trace before* F. Intuitively, proving the query F ==> F' by induction consists of proving the above property by induction on the length of the traces of P.

To simplify the explanation, let us introduce some informal notations. We consider that a trace of P is a sequence of actions $\mathsf{tr} = a_1 \dots a_n$ representing the actions that have been executed in P similarly to the attack traces (see Section 3.3.2). The length of the trace, denoted $|\mathsf{tr}|$, corresponds to its number of actions, that is, n. Finally, we say that a fact is executed at *step* k, denoted $F, k \vdash \mathsf{tr}$ when F is the action a_k in tr. The induction hypothesis $\mathcal{P}(n)$ can then be expressed as:

for all traces tr of P, if $|\mathsf{tr}| \leq n$, then for all k, if $F, k \vdash \mathsf{tr}$ then $F', k' \vdash \mathsf{tr}$ for some $k' \leq k$.

For ProVerif to prove this property by induction, we only need to prove that $\mathcal{P}(n)$ implies $\mathcal{P}(n+1)$ for all $n \in \mathbb{N}$. (Note that $\mathcal{P}(0)$ is trivially always true.)

By considering a trace $\mathsf{tr} = a_1 \dots a_{n+1}$ and assuming that $\mathcal{P}(n)$ holds, we directly obtain that the sub-trace $\mathsf{tr}' = a_1 \dots a_n$ satisfies $\mathcal{P}(n)$. This yields two interesting properties:

- We can consider that $k = n + 1$, otherwise the result would directly hold thanks to tr'.
- In the solving procedure, when building the derivations of σF, if we can detect that another instance of F, say $\sigma' F$, in the derivation necessarily occurred stricly before σF, then we know by the induction hypothesis $\mathcal{P}(n)$ that $\sigma' F'$ has been executed before $\sigma' F$ and so before σF.

These two properties are the building blocks of the inductive verification of queries in ProVerif: When generating reachable goals, ProVerif builds Horn clauses with instances of F as a conclusion. Upon generating a clause of the form $H \wedge \sigma' F \rightarrow \sigma F$, ProVerif already knows that this clause represents an execution of $\sigma' F$ before an execution of σF. ProVerif uses order constraints to infer that $\sigma' F$ was executed strictly before σF. In this case, the verification procedure will *add* $\sigma' F'$ to the hypotheses of the clause, i.e., it replaces the clause $H \wedge \sigma' F \rightarrow \sigma F$ with the clause $H \wedge \sigma' F' \wedge \sigma' F \rightarrow \sigma F$.

Let us illustrate this concept on the small example, available in `docs/ex_induction.pv`, that is a simplified version of the Yubikey protocol [Yub10].

```
1 free c:channel.
2 free k:bitstring [private].
3 free d_P:channel [private].
```

```
free d_Q:channel [private].

fun senc(nat,bitstring):bitstring.
reduc forall K:bitstring,M:nat; sdec(senc(M,K),K) = M.

event CheckNat(nat).

query i:nat; event(CheckNat(i)) ==> is_nat(i).

let P =
  in(c,x:bitstring);
  in(d_P,(i:nat,j:nat));
  let j':nat = sdec(x,k) in
  event CheckNat(i);
  event CheckNat(j);
  if j' > j
  then out(d_P,(i+1,j'))
  else out(d_P,(i,j)).

let Q =
  in(d_Q,i:nat);
  out(c,senc(i,k));
  out(d_Q,i+1).

process
  out(d_P,(0,0)) | out(d_Q,0) | ! P | ! Q
```

In this protocol, the processes P and Q share a private key k and they both have a memory cell respectively represented by the private channels d_P and d_Q. Every time the process Q increments the value stored in its memory cell, it also outputs the previous value encrypted with the shared key k, i.e. **out**(c,senc(i,k)). On the other hand, the process P stores in its memory cell two values : the number of time it received a *fresh* encryption from Q, represented by i:nat in **in**(d_P,(i:nat, j:nat)) and the last value it received from Q, represented by j:nat.

We aim to prove that the values of the memory cell of P are always natural numbers, which is represented by the query:

```
query i:nat; event(CheckNat(i)) ==> is_nat(i).
```

However, verifying this protocol with `./proverif docs/ex_induction.pv | grep "RES"` produces the following output:

```
RESULT event(CheckNat(i_2)) ==> is_nat(i_2) cannot be proved.
```

If we look more closely at the output, we can observe that ProVerif considers the following reachable goal

```
is_not_nat(i_2 + 1) && j_1 ≥ j_2 + 1 && mess(d_P[],(i_2,j_2)) &&
    mess(d_Q[],j_1) && mess(d_Q[],j'_1) -> end(CheckNat(i_2 + 1))
```

To ensure termination, ProVerif avoids resolving upon facts that would lead to trivial infinite loops. This is the case for the facts representing the memory cells, which are **mess**(d_P[],(i_2,j_2)), **mess**(d_Q[],j_1), and **mess**(d_Q[],j'_1), so resolution stops with the clause above. Since the clause contradicts the query, ProVerif concludes that it cannot prove the query.

By adding the option **induction** after the query as follows

```
query i:nat; event(CheckNat(i)) ==> is_nat(i) [induction].
```

ProVerif would initially generate the following reachable goal:

```
j_1 ≥ j_2 + 1 && begin(CheckNat(j_2)) && begin(CheckNat(i_2)) &&
    mess(d_P[],(i_2,j_2)) && mess(d_Q[],j_1) -> end(CheckNat(i_2 + 1))
```

Furthermore, ProVerif understands that the event CheckNat(i_2) occurs strictly before CheckNat(i_2 + 1). By applying the induction hypothesis on CheckNat(i_2), it adds is_nat(i_2) in the hypotheses of the clause, yielding

```
is_nat(i_2) && j_1 ≥ j_2 + 1 && begin(CheckNat(j_2)) && begin(CheckNat(i_2)) &&
    mess(d_P[],(i_2,j_2)) && mess(d_Q[],j_1) -> end(CheckNat(i_2 + 1))
```

Since this clause does not contradict the query, ProVerif is able to prove the query: Verifying this protocol with `./proverif docs/ex_induction_proof.pv | grep "RES"` produces the output

```
RESULT event(CheckNat(i_2)) ==> is_nat(i_2) is true.
```

Remark. When the setting inductionQueries is set to true, all queries are proved by induction. In such a case, one can use the option [**noInduction**] on one specific query to enforce that it is *not* proved by induction.

6.1.2 Group of queries

Queries may also be stated in the form:

query $x_1 : t_1, \ldots, x_m : t_m$; $q_1; \ldots; q_n$.

where each q_i is a query as defined in Figure 4.3. Furthermore, it is also possible to prove a group of queries by induction. However the output of ProVerif differs from proving a single query by induction. Coming back to our previous example, we would additionally prove that the values stored in the memory cell Q and the value of j' in P are also natural numbers. The input file `docs/ex_induction_group.pv` partially displayed here integrates such queries.

```
9  event CheckNat(nat).
10 event CheckNatQ(nat).
11
12 query i:nat;
13   event(CheckNat(i)) ==> is_nat(i);
14   event(CheckNatQ(i)) ==> is_nat(i);
15   mess(d_Q,i) ==> is_nat(i) [induction].
16
17 let P =
18   in(c,x:bitstring);
19   in(d_P,(i:nat,j:nat));
20   let j':nat = sdec(x,k) in
21   event CheckNat(i);
22   event CheckNat(j);
23   event CheckNatQ(j');
24   if j' > j
25   then out(d_P,(i+1,j'))
26   else out(d_P,(i,j)).
```

Verifying this protocol with `./proverif docs/ex_induction_group.pv | grep "RES"` produces the following output:

```
PARTIAL RESULT event(CheckNat(i_2)) ==> is_nat(i_2) is true if
  the inductive queries can be proved.
PARTIAL RESULT event(CheckNatQ(i_2)) ==> is_nat(i_2) is true if
  the inductive queries can be proved.
PARTIAL RESULT mess(d_Q[],i_2) ==> is_nat(i_2) cannot be proved if
  the inductive queries can be proved.

PARTIAL RESULT event(CheckNat(i_2)) ==> is_nat(i_2) is true if
  the inductive queries can be proved.
```

```
PARTIAL RESULT event(CheckNatQ(i_2)) ==> is_nat(i_2) cannot be proved if
  the inductive queries can be proved.
PARTIAL RESULT mess(d_Q[],i_2) ==> is_nat(i_2) cannot be proved if
  the inductive queries can be proved.

PARTIAL RESULT event(CheckNat(i_2)) ==> is_nat(i_2) is true if
  the inductive queries can be proved.
PARTIAL RESULT event(CheckNatQ(i_2)) ==> is_nat(i_2) cannot be proved if
  the inductive queries can be proved.
PARTIAL RESULT mess(d_Q[],i_2) ==> is_nat(i_2) cannot be proved if
  the inductive queries can be proved.

FINAL RESULT:
RESULT mess(d_Q[],i_2) ==> is_nat(i_2) cannot be proved.
RESULT event(CheckNatQ(i_2)) ==> is_nat(i_2) cannot be proved.
RESULT event(CheckNat(i_2)) ==> is_nat(i_2) is true.
```

The proof of a group of queries by induction is done in multiple steps. In the first step, ProVerif assumes that the inductive hypotheses of all individual queries hold and it tries to prove the group of queries under this assumption. If the verification succeeds, then ProVerif concludes that the group of queries is true. When however ProVerif cannot verify *all* the queries, it will refine the inductive hypotheses to consider. More specifically, it will try to prove the group of queries again, but only under the inductive hypotheses of the individual queries that it was previously able to prove. ProVerif repeats this refinement of inductive queries until it can prove all of them.

In our example, the first three partial results correspond to the first step where ProVerif assumed as inductive hypotheses the three queries. Under this assumption, it was only able to prove two of them, namely **event**(CheckNat(i_2)) ==> is_nat(i_2) and **event**(CheckNatQ(i_2)) ==> is_nat(i_2). The next three partial results therefore correspond to the second step where ProVerif only assumes as inductive hypotheses the queries **event**(CheckNat(i_2)) ==> is_nat(i_2) and **event**(CheckNatQ(i_2)) ==> is_nat(i_2). In this second step, the query **event**(CheckNatQ(i_2)) ==> is_nat(i_2) cannot be proved anymore. Since ProVerif did not prove the two inductive queries, it refines again its inductive hypotheses by considering only **event**(CheckNat(i_2)) ==> is_nat(i_2). Since it is able to prove this query in the third step, ProVerif can conclude that it is true.

Note that the verification summary only displays the final results.

```
Verification summary:

Query event(CheckNat(i_2)) ==> is_nat(i_2) is true.

Query event(CheckNatQ(i_2)) ==> is_nat(i_2) cannot be proved.

Query mess(d_Q[],i_2) ==> is_nat(i_2) cannot be proved.
```

We explain in Section 6.7.2 why ProVerif is not able to prove the query **mess**(d_Q[],i_2) ==> is_nat(i_2) and how one can help ProVerif to prove it.

Remark. By default, for a group of queries, ProVerif does not apply the induction hypothesis during saturation, since some of the queries may not be true. The user may add the option **proveAll** to the group:

query $x_1 : t_1, \ldots, x_m : t_m$; $q_1; \ldots; q_n$ [**induction**, **proveAll**].

in order to tell ProVerif that it should prove all queries; it can then use them as induction hypothesis during saturation. In case some of the queries cannot be proved, all queries of the group are considered as not proved since the proof was attempted with an induction hypothesis that does not hold.

6.2 Axioms, restrictions, and lemmas

ProVerif supports the declaration of lemmas in addition to standard queries with the following syntax:

lemma $x_1 : t_1, \ldots, x_n : t_n$; $cq_1; \ldots; cq_n$.
axiom $x_1 : t_1, \ldots, x_n : t_n$; $cq_1; \ldots; cq_n$.
restriction $x_1 : t_1, \ldots, x_n : t_n$; $cq_1; \ldots; cq_n$.

where $cq_1, \ldots, cq_n$ are reachability or correspondence queries as defined in Figure 4.3 with the following restrictions: If cq_i is the query F_1 && ... && F_m ==> H then

- the variables of H are variables of $F_1, \ldots, F_m$;
- H does not contain any nested correspondence;
- facts of H can only be non-injective events, equalities, inequalities, disequalities;
- $F_1, \ldots, F_m$ can only be non-injective events, attacker predicates, message predicates, or table predicates;
- facts with temporal variables $AF@t$ are not allowed.

These lemmas, restrictions, and axioms will be used internally by ProVerif to remove, simplify or instantiate clauses during the saturation procedure of the main query. Intuitively, a lemma F_1 && ... && F_m ==> H is applied on a clause H' -> C' when there exists a substitution σ such that $F_i\sigma \subseteq H'$ for all $i = 1 \ldots m$; and the resulting clause being H' && $H\sigma$ -> C'.

When restrictions are declared, ProVerif will prove the main queries only on all traces (resp. bitraces) of the input process (resp. biprocess) that satisfy the restrictions.

To preserve soundness, ProVerif proves all the lemmas as if they were standard queries before using them in the saturation procedure. ProVerif will produce an error if it is not able to prove one of the lemmas. Note that if restrictions are also declared, the lemmas are proved only on the traces satisfying the restrictions. However, ProVerif assumes that all axioms are true on the input process and does not attempt to prove them. When axioms are declared, it is important to note that a security proof holds assuming that the axioms also hold. Axioms are typically useful for hand-proved properties that cannot be proved with ProVerif.

Depending on the lemmas, restrictions, and axioms declared, precision and termination of ProVerif can be improved. ProVerif ignores the number of repetitions of actions due to the transformation of processes into Horn clauses. Hence, the following example yields a false attack:

```
new k:key; out(c, senc(senc(s,k),k));
in(c, x:bitstring); out(c, sdec(x,k))
```

where c is a public channel, s is a private free name which should be kept secret, and senc and sdec are symmetric encryption and decryption respectively. ProVerif thinks that one can decrypt senc(senc(s,k),k) by sending it to the input, so that the process replies with senc(s,k), and then sending this message again to the input, so that the process replies with s. However, this is impossible in reality because the input can be executed only once.

However, a generic transformation on processes, presented in [CCT18], using events allows to partially take into account the number of repetitions of actions. Intuitively after each input, an event recording the input message is added.

```
new st:stamp; new k:key; out(c, senc(senc(s,k),k));
in(c, x:bitstring); event UAction(st,x); out(c, sdec(x,k))
```

It was shown in [CCT18] that adding such events preserves the security properties and, moreover, that the following query always holds:

```
forall st:stamp, x:bitstring, y:bitstring;
  event(UAction(st,x)) && event(UAction(st,y)) ==> x = y.
```

Intuitively, the input action **in**(c, x: bitstring) is executed at most once for each value of the stamp st. Hence, if the value of the stamp st is the same, then the value of the input message x must also be the same. Ideally, we would declare this property as a lemma, but ProVerif is unable to prove it. Hence, since that property was shown by hand in [CCT18], we can declare it as an axiom. In the following complete script, ProVerif is thus able to prove the secrecy of s.

```
free c:channel.
free s:bitstring [private].

type key.
type stamp.
fun senc(bitstring,key):bitstring.
reduc forall x:bitstring,y:key; sdec(senc(x,y),y) = x.

event UAction(stamp,bitstring).

axiom st:stamp, x:bitstring, y:bitstring;
  event(UAction(st,x)) && event(UAction(st,y)) ==> x = y.

query attacker(s).

process
  new k:key; out(c, senc(senc(s,k),k));
  in(c, x:bitstring);
  new st[]:stamp;
  event UAction(st,x);
  out(c, sdec(x,k))
```

In fact, this generic transformation has been natively added in ProVerif and can be activated by adding the option [**precise**] after the input. In our example, it would correspond to the following process.

```
new k:key; out(c, senc(senc(s,k),k));
in(c, x:bitstring) [precise]; out(c, sdec(x,k))
```

Similarly, the option [**precise**] can be added in the **get** ... **in** P **else** Q and **let** ... **suchthat** (see Section 6.3) constructs as follows.

get $d(T_1,\ldots,T_n)$ [**precise**] **in** P **else** Q
get $d(T_1,\ldots,T_n)$ **suchthat** M [**precise**] **in** P **else** Q

let $x_1 : t_1,\ldots,x_n : t_n$ **suchthat** $p(M_1,\ldots,M_k)$ [**precise**] **in** P **else** Q

Alternatively, one can use the setting **set** preciseActions = true, which means that all inputs, **get** ... **in** P **else** Q, and **let** ... **suchthat** constructs have the option **precise**. Hence ProVerif is able to prove the secrecy of s in the following script.

```
free c:channel.
free s:bitstring [private].

type key.
fun senc(bitstring,key):bitstring.
reduc forall x:bitstring,y:key; sdec(senc(x,y),y) = x.

set preciseActions = true.

query attacker(s).

process
```

```
13   new k:key; out(c, senc(senc(s,k),k));
14   in(c, x:bitstring);
15   out(c, sdec(x,k))
```

Subterm predicate

We allow in the premise of an axiom or a restriction a special predicate **subterm**(M,N). This predicate holds when M is a subterm of N modulo the equational theory. For example, the following restriction

```
restriction x:bitstring, y:nat; event(A(x)) && subterm(y,x) ==> y <> 0
```

restricts the verification of queries to traces such that if the event A(M) is emitted then 0 is not a subterm of M.

Note that the predicate **subterm** is not allowed in lemmas or queries.

Order of lemmas

As for queries, lemmas can be either grouped inside a single **lemma** declaration or they can be separately declared with multiple **lemma** declarations. While grouping the lemmas may improve the performance of ProVerif, declaring them separately may improve its completeness. Indeed, ProVerif proves the lemmas in the order they are declared in the input file. Moreover, it also uses proven lemmas to help proving new lemmas. For example, by declaring the following lemmas,

lemma $x_1^1 : t_1^1, \ldots, x_n^1 : t_n^1 ;\ \ cq_1 .$
lemma $x_1^2 : t_1^2, \ldots, x_n^2 : t_n^2 ;\ \ cq_2 .$
lemma $x_1^3 : t_1^3, \ldots, x_n^3 : t_n^3 ;\ \ cq_3 .$

ProVerif first tries to prove cq_1 alone then tries to prove cq_2 by using cq_1 in the saturation procedure and finally tries to prove cq_3 by using cq_1 and cq_2 in the saturation procedure.

The order of **axiom** and **restriction** declarations does not matter as they are globally applied to all lemmas and queries.

Options

Adding lemmas in most cases improves the completeness of ProVerif. However, it is less clear how lemmas influence its termination as it heavily depends on the process and declared lemmas. Thus, lemmas can be declared with several options to parameterize how lemmas should be applied during the saturation procedure. The following exclusive options are available: **noneSat**, **discardSat**, **instantiateSat** (default), **fullSat**. The option **noneSat** indicates that the lemma should not be used in the saturation. The saturation behaves as if the lemma was declared as a query. The option **discardSat** enforces that a lemma should only be applied if its application on a Horn clause renders its hypotheses unsatisfiable. The option **instantiateSat** enforces that the lemma should instantiate at least one variable of the Horn clause or render the hypotheses of the clause unsatisfiable. Finally, with the option **fullSat**, the lemma is applied without restriction. These options can also be given for declared axioms.

When defining a lemma with the option **removeEvents**, ProVerif removes from clauses the events that correspond to the premise of this lemma and that do not seem useful anymore because the lemma has already been applied or it will never be applicable using this event. The latter is approximated hence may result in a loss of precision (ProVerif remains sound) but it also speeds up the resolution and may even allow it to terminate. The opposite option **keepEvents** ensures that ProVerif will never remove from clauses any event corresponding to the premise of this lemma. This is the default. The default can be changed using the global setting removeEventsForLemma (see Section 6.6.2), so that all lemmas without explicit option can be considered as being declared with the option **removeEvents**.

In the case of correspondence queries and once the saturation completes, ProVerif will also rely on lemmas, restrictions, and axioms when verifying queries. We therefore have similar options parameterizing how lemmas should be applied during the verification procedure: **noneVerif**, **discardVerif**, **instantiateVerif**, and **fullVerif** (default). Note that, in contrast to the default option for the saturation procedure, ProVerif applies lemmas and axioms without restriction by default during the verification procedure.

Finally, lemmas can be declared with the option **maxSubset**. By default, when ProVerif is unable to prove a lemma or a group of lemmas, it raises an error. With the option **maxSubset**, ProVerif aims to find the maximal subset of provable lemmas in a group and discards the remaining ones. Soundness is guaranteed by the fact that ProVerif only keeps the lemmas in the group that it is able to prove. Note that this option is not allowed for axioms. Moreover, this option is not exclusive with the options **noneSat**, **discardSat**, **instantiateSat**, and **fullSat**. For example, in the following script, ProVerif first tries to find the maximal subset S of lemmas $cq_1, \ldots, cq_n$ that it can prove. Second, it will prove the query **attacker**(s) by only using the lemmas in S when they refute the hypotheses of Horn clauses during the saturation procedure for the query **attacker**(s).

lemma $x_1 : t_1, \ldots, x_n : t_n$; $cq_1; \ldots; cq_n$ [**maxSubset**, **discardSat**].
query attacker(s).

As any query, a lemma can be proved by induction by adding the option **induction**. By default, since a lemma must be proved by ProVerif (otherwise an error is raised), the inductive hypothesis corresponding to the lemma is also applied during the saturation procedure, which may enforce its termination. (For groups of queries, this happens only with the option **proveAll**.) When a group of lemmas is declared with the option **maxSubset**, the inductive hypothesis is not applied during the saturation procedure and is only applied during the verification procedure (similarly to the default situation for queries). Note that the options **noneSat**, **noneVerif**, **discardSat**, ... can also modify how the inductive hypothesis is applied during the saturation and verification procedures.

Remark 1. When the setting inductionLemmas is set to true, all lemmas are proved by induction. In such a case, one can use the option [**noInduction**] on one specific lemma to enforce that it is *not* proved by induction. Moreover, the default applications of lemmas during the saturation and verification procedures can also be modified using global settings (see Section 6.6.2).

Remark 2. When ProVerif fails to prove an equivalence query (or a real or random query) on the initial process, ProVerif tries to generate simplified processes on which to prove the query. Though the simplification preserves equivalence properties, an axiom (resp. restriction) that holds on the initial process does not necessarily hold on the simplified process. Therefore by default, ProVerif does not consider the axioms (resp. restrictions) on these simplified processes. These axioms (resp. restrictions) can however still be considered on simplified processes by declaring the axiom (resp. restriction) with the option **keep**.

Lemmas for equivalence queries

ProVerif also supports the declaration of lemmas for equivalence queries by proving correspondence queries on biprocesses and more specifically on bitraces of biprocesses. Once proved, the lemmas are used during the saturation procedure for the equivalence query. Thus, lemmas can be used to help ProVerif prove a previously unproved equivalence but they can also be used to enforce termination. Intuitively, to prove a lemma on a biprocess, ProVerif generates the same set of Horn clauses as the ones generated for an equivalence proof but removes clauses with bad as conclusion. Indeed, these clauses are only useful to prove equivalence and can be soundly ignored when proving a correspondence query on a biprocess. Since ProVerif does not saturate the same set of Horn clauses, one may hope that ProVerif terminates for the proof of the lemma which would then be used to enforce termination for the equivalence proof. As an example, consider the simplified Yubikey protocol introduced in Section 6.1 modified as follows.

```
free c:channel.
free k:bitstring [private].
free d_P:channel [private].
free d_Q:channel [private].

fun senc(nat,bitstring):bitstring.
reduc forall K:bitstring,M:nat; sdec(senc(M,K),K) = M.

```

```
let P =
  in(c,x:bitstring);
  in(d_P,(i:nat,j:nat));
  let j':nat = sdec(x,k) in
  if j' > j
  then out(d_P,(i+1,choice[j',j'+1]))
  else out(d_P,(i,j)).

let Q =
  in(d_Q,i:nat);
  out(c,senc(i,k));
  out(d_Q,i+1).

process
  out(d_P,(0,0)) | out(d_Q,0) | !P | !Q
```

In this protocol, we compare the case where the process P either records the value encrypted by Q or its value plus one. Executing ProVerif on `docs/ex_lemmas_equiv.pv` yields many termination warnings on increasingly large clauses (a good indicator of the non-termination of the saturation procedure). However, the clauses of almost all warnings contain the fact mess2(d_Q[],j_1,d_Q[],j'_2) suggesting that ProVerif is considering derivations where the value of the memory cell stored by Q differs on the two sides of the equivalence. One can use a lemma to show that these derivations cannot happen and so should be discarded.

The function **choice** can be used in a lemma to specify different terms on the two sides of the bitrace. In our example, the following lemma can be declared (see `docs/ex_lemmas_equiv_proof.pv`)

```
lemma i:nat, i':nat; mess(d_Q,choice[i,i']) ==> i = i'.
```

Executing ProVerif on `docs/ex_lemmas_equiv_proof.pv`, one can see that the lemma is actually encoded as

```
mess2(d_Q[],i_3,d_Q[],i') ==> i_3 = i'
```

and that ProVerif is able to prove the lemma and the equivalence.

Remark 1. In the example `docs/ex_lemmas_equiv_proof.pv`, ProVerif first proves the lemma in the biprocess given as input. However, ProVerif fails to prove the equivalence on this biprocess despite the lemma. Thus it simplifies the biprocess into a new one (named biprocess 1). Even though the lemma was proved on the input biprocess, it does not necessarily imply that the lemma holds on biprocess 1. It was only shown that simplification of biprocesses preserves observational equivalence. Therefore, ProVerif proves the lemma on biprocess 1 again, and finally proves the desired equivalence. In this case, the verification summary just shows the results on the biprocess on which the equivalence was proved, as follows:

```
--------------------------------------------------------------
Verification summary:

Query(ies):
 - Observational equivalence is true.
Associated lemma(s):
 - Lemma mess(d_Q[],choice[i_3,i']) ==> i_3 = i' encoded as
   mess2(d_Q[],i_3,d_Q[],i') ==> i_3 = i' is true in biprocess 1.

--------------------------------------------------------------
```

Remark 2. Lemmas on biprocesses can also be proved by induction by adding the option **induction**.

Remark 3. In fact, to prove a lemma on a biprocess, ProVerif does not remove all clauses with bad as conclusion during the initial generation of Horn clauses. It preserves the clauses corresponding to the attacker power to distinguish messages but removes the ones that focus on the control flow of the biprocess. Keeping these clauses allows ProVerif to activate an optimization during the saturation procedure which improves termination. After completion of the saturation procedure, if bad is shown to be derivable, then ProVerif considers that it cannot prove the lemma. Leaving these clauses with bad as conclusion does not sacrifice soundness since the lemma is rejected when bad is derivable.

Public variables and secrecy

As shown in Section 4.3.1, the syntax of queries q is as follows:

cq **public_vars** $y_1, \ldots, y_m$
secret x **public_vars** $y_1, \ldots, y_m$ [reachability]
secret x **public_vars** $y_1, \ldots, y_m$ [real_or_random]

where the indication [reachability] may be omitted or replaced with [pv_reachability], [real_or_random] may be replaced with [pv_real_or_random], and **public_vars** $y_1, \ldots, y_m$ may be omitted. When present, **public_vars** $y_1, \ldots, y_m$ means that $y_1, \ldots, y_m$ are public, that is, the adversary has read access to them. Queries with public variables are implemented by modifying the considered process to output the contents of these variables on a public channel. Similarly, queries **secret** x **public_vars** $y_1, \ldots, y_m$ [real_or_random] are implemented by modifying the process to express observational equivalence between the case in which the protocol outputs x and the case in which it outputs a fresh random value. (The modified process is then a biprocess.) Different lemmas or axioms may hold for different processes, so for different public variables and for real-or-random secrecy queries. Therefore, the user has to specified to which queries the lemmas and axioms apply. This is done as follows:

- Lemmas and axioms cq_i apply to queries without public variables and that are not real-or-random secrecy queries, that is, correspondence queries with public variables cq, strong secrecy queries, off-line guessing attacks queries and secrecy queries **secret** x [reachability], as well as equivalence queries between two processes. Only in the last case, the lemmas or axioms may contain the function **choice** (but not necessarily).
- Lemmas and axioms cq_i **for** { **public_vars** $y_1, \ldots, y_m$ } apply to queries with public variables $y_1, \ldots, y_m$ and that are not real-or-random secrecy queries, that is, cq **public_vars** $y_1, \ldots, y_m$ and **secret** x **public_vars** $y_1, \ldots, y_m$ [reachability]. These lemmas and axioms must not contain the function **choice**.
- Lemmas and axioms cq_i **for** { **secret** x **public_vars** $y_1, \ldots, y_m$ [real_or_random] } apply only to the query **secret** x **public_vars** $y_1, \ldots, y_m$ [real_or_random], and similarly lemmas and axioms cq_i **for** { **secret** x [real_or_random] } apply only to the query **secret** x [real_or_random]. These lemmas and axioms may contain the function **choice** (but not necessarily).

For example, in the following input file (partially displayed),

```
1  axiom y:bitstring,y':bitstring;
2    event(A(choice[y,y'])) ==> y = y'
3    for { secret s public_vars x [real_or_random] }.
4
5  lemma x:bitstring;
6    event(B(x)) ==> x = a;
7    event(A(x)) ==> x = b.
8
9  lemma x:bitstring; event(A(x)) ==> x = b for { public_vars x }.
10
11 query x:bitstring;
12   event(A(x)) ==> event(B(a)) public_vars x;
13   attacker(x).
14
```

```
15 noninterf d.
16
17 query secret s public_vars x [real_or_random].
```

the axiom will only be used for the proof of **secret** s **public_vars** x [real_or_random]; the first lemma will be used for the proofs of **attacker**(x) and **noninterf** d; the last lemma will be used for the proof of **event**(A(x)) ==> **event**(B(a)) **public_vars** x.

6.3 Predicates

ProVerif supports predicates defined by Horn clauses as a means of performing complex tests or computations. Such predicates are convenient because they can easily be encoded into the internal representation of ProVerif which also uses clauses. Predicates are defined as follows:

pred $p(t_1, \ldots, t_k)$.

declares a predicate p of arity k that takes arguments of types $t_1, \ldots, t_k$. The predicates **attacker**, **mess**, ev, and evinj are reserved for internal use by ProVerif and cannot be declared by the user. The declaration

clauses $C_1; \ldots; C_n$.

declares the clauses $C_1, \ldots, C_n$ which define the meaning of predicates. Clauses are built from facts which can be $p(M_1, \ldots, M_k)$ for some predicate declared by **pred**, $M_1 = M_2$, or $M_1 <> M_2$. The clauses C_i can take the following forms:

- **forall** $x_1 : t_1, \ldots, x_n : t_n$; F

  which means that the fact F holds for all values of the variables $x_1, \ldots, x_n$ of type $t_1, \ldots, t_n$ respectively; F must be of the form $p(M_1, \ldots, M_k)$.

- **forall** $x_1 : t_1, \ldots, x_n : t_n$; F_1 && ... && F_m -> F

  which means that $F_1, \ldots,$ and F_m imply F for all values of the variables $x_1, \ldots, x_n$ of type $t_1, \ldots, t_n$ respectively; F must be of the form $p(M_1, \ldots, M_k)$; $F_1, \ldots, F_m$ can be any fact.

In all clauses, the fact F is considered to hold only if its arguments do not fail and when the arguments of the facts in the hypothesis of the clause do not fail: for facts $p(M_1, \ldots, M_k)$, $M_1, \ldots, M_k$ do not fail, for equalities $M_1 = M_2$ and disequalities $M_1 <> M_2$, M_1 and M_2 do not fail.

Additionally, ProVerif allows the following equivalence declaration in place of a clause

forall $x_1 : t_1, \ldots, x_n : t_n$; F_1 && ... && F_m <-> F

which means that $F_1, \ldots,$ and F_m hold if and only if F holds; $F_1, \ldots, F_m, F$ must be of the form $p(M_1, \ldots, M_k)$. Moreover, σF_i must be of smaller size than σF for all substitutions σ and two facts F of different equivalence declarations must not unify. (ProVerif will check these conditions.) This equivalence declaration can be considered as an abbreviation for the clauses

forall $x_1 : t_1, \ldots, x_n : t_n$; F_1 && ... && F_m -> F
forall $x_1 : t_1, \ldots, x_n : t_n$; F -> F_i $(1 \leq i \leq m)$

but it further enables the replacement of σF with the equivalent facts σF_1 && ... && σF_m in all clauses. This replacement may speed up the resolution process, and generalizes the replacement performed for data constructors.

(The declaration **forall** $x_1 : t_1, \ldots, x_n : t_n$; F_1 && ... && F_m <=> F is equivalent to the previous one. It is kept only for backward compatibility.)

In all these clauses, all variables of $F_1, \ldots, F_m, F$ must be universally quantified by **forall** $x_1 : t_1, \ldots, x_n : t_n$. When $F_1, \ldots, F_m, F$ contain no variables, the part **forall** $x_1 : t_1, \ldots, x_n : t_n$; can be omitted. In **forall** $x_1 : t_1, \ldots, x_n : t_n$, the types $t_1, \ldots, t_n$ can be either just a type identifier, or of the form t **or fail**, which means that the considered variable is allowed to take the special value **fail** in addition to the values of type t.

Finally, the declaration

elimtrue $x_1 : t_1, \ldots, x_n : t_n$; $p(M_1, \ldots, M_k)$.

means that for all values of the variables $x_1, \ldots, x_n$, the fact $p(M_1, \ldots, M_k)$ holds, like the declaration **clauses forall** $x_1 : t_1, \ldots, x_n : t_n$; $p(M_1, \ldots, M_k)$. However, it additionally enables an optimization: in a clause $R = F'$ && H -> C, if F' unifies with F with most general unifier σ_u and all variables of F' modified by σ_u do not occur in the rest of R then the hypothesis F' can be removed: R is transformed into H -> C, by resolving with F. As above, the types t_1, ..., t_n can be either just a type identifier, or of the form t **or fail**.

Predicate evaluation. Predicates can be used in **if** tests. As a trivial example, consider the script:

```
pred p(bitstring,bitstring).

elimtrue x:bitstring, y:bitstring; p(x,y).

event e.
query event(e).

process new m:bitstring; new n:bitstring; if p(m,n) then event e
```

in which ProVerif demonstrates the reachability of event e.

Predicates can also be evaluated using the **let** ... **suchthat** construct:

let $x_1 : t_1, \ldots, x_n : t_n$ **suchthat** $p(M_1, \ldots, M_k)$ **in** P **else** Q

where $M_1, \ldots, M_k$ are terms built over variables $x_1, \ldots, x_n$ of type $t_1, \ldots, t_n$ and other terms. If there exists a binding of $x_1, \ldots, x_n$ such that the fact $p(M_1, \ldots, M_k)$ holds, then P is executed (with the variables $x_1, \ldots, x_n$ bound inside P); if no such binding can be achieved, then Q is executed. As usual, Q may be omitted when it is the null process. When there are several suitable bindings, one possibility is chosen (but ProVerif considers all possibilities when reasoning). Note that the **let** ... **suchthat** construct does not allow an empty set of variables $x_1, \ldots, x_n$; in this case, the construct **if** $p(M_1, \ldots, M_k)$ **then** P **else** Q should be used instead.

The **let** ... **suchthat** construct is allowed in enriched terms (see Section 4.1.4) as well as in processes.

Note that there is an implementability condition on predicates, to make sure that the values of $x_1, \ldots, x_n$ in **let** $x_1 : t_1, \ldots, x_n : t_n$ **suchthat** constructs can be efficiently computed. Essentially, for each predicate invocation, we bind variables in the conclusion of the clauses that define this predicate and whose position corresponds to bound arguments of the predicate invocation. Then, when evaluating hypotheses of clauses from left to right, all variables of predicates must get bound by the corresponding predicate call. The verification of the implementability condition can be disabled by

```
set predicatesImplementable = nocheck.
```

Recursive definitions of predicates are allowed.

Predicates and the **let** ... **suchthat** construct are incompatible with strong secrecy (modeled by **noninterf**) and with **choice**.

Example: Modeling sets with predicates. As an example, we will demonstrate how to model sets with predicates (see file `docs/ex_predicates.pv`).

```
type bset.
fun consset(bitstring, bset): bset [data].
const emptyset: bset [data].
```

Sets are represented by lists: emptyset is the empty list and consset(M,N) concatenates M at the head of the list N.

```
pred mem(bitstring, bset).
clauses
  forall x:bitstring, y:bset; mem(x,consset(x,y));
  forall x:bitstring, y:bset, z:bitstring; mem(x,y) -> mem(x,consset(z,y)).
```

The predicate mem represents set membership. The first clause states that mem(M,N) holds for some terms M, N if N is of the form consset(M,N'), that is, M is at the head of N. The second clause states that mem(M,N) holds if N = consset(M',N') and mem(M,N') holds, that is, if M is in the tail of N. We conclude our example with a look at the following ProVerif script:

```
event e.
event e'.
query event(e).
query event(e').

type bset.
fun consset(bitstring, bset): bset [data].
const emptyset: bset [data].
pred mem(bitstring, bset).
clauses
  forall x:bitstring, y:bset; mem(x,consset(x,y));
  forall x:bitstring, y:bset, z:bitstring; mem(x,y) -> mem(x,consset(z,y)).

process
   new a: bitstring;new b: bitstring; new c: bitstring;
   let x = consset(a,emptyset) in
   let y = consset(b,x) in
   let z = consset(c,y) in (
      if mem(a,z) then
      if mem(b,z) then
      if mem(c,z) then
      event e
   )|(
      let w:bitstring suchthat mem(w,x) in event e'
   )
```

As expected, ProVerif demonstrates reachability of both e and e'. Observe that e' is reachable by binding the name a to the variable w.

Using predicates in queries. User-defined predicates can also be used in queries, so that the grammar of facts F in Figure 4.3 is extended with user-defined facts $p(M_1, \ldots, M_n)$. As an example, the query

```
query x:bitstring; event(e(x)) ==> p(x)
```

holds when, if the event e(x) has been executed, then p(x) holds. (If this property depends on the code of the protocol but not on the definition of p, for instance because the event e(x) can be executed only after a successful test **if** p(x) **then**, a good way to prove this query is to declare the predicate p with option **block** and to omit the clauses that define p, so that ProVerif does not use the definition of p. See below for additional information on the predicate option **block**.)

Predicate options. Predicate declarations may also mention options:

pred $p(t_1, \ldots, t_k)$ $[o_1, \ldots, o_n]$.

The allowed options $o_1, \ldots, o_n$ are:

- **block**: Declares the predicate p as a blocking predicate. Blocking predicates must appear only in hypotheses of clauses. This situation typically happens when the predicate is defined by no clause declaration, but is used in tests or **let** ... **suchthat** constructs in the process (which leads to generating clauses that contain the predicate in hypothesis).

  Instead of trying to prove facts containing these predicates (which is impossible since no clause implies such facts), ProVerif collects hypotheses containing the blocking predicates necessary to

prove the queries. In other words, ProVerif proves properties that hold for *any* definition of the considered blocking predicate.

Blocking predicates are allowed after ==> in lemmas and axioms. Other user-defined predicates are forbidden in lemmas and axioms.

- **memberOptim**: This must be used only when p is defined by

  $p(x, f(x, y))$
  $p(x, y)$ –> $p(x, f(x', y))$

  where f is a data constructor. (Note that it corresponds to the case in which p is the membership predicate and $f(x, y)$ represents the union of element x and set y.)

  memberOptim enables the following optimization: attacker(x) && $p(M_1, x)$ && ... && $p(M_n, x)$ where p is declared **memberOptim** is replaced with attacker(x) && attacker(M_1) && ... && attacker(M_n) when x does not occur elsewhere (just take $x = f(M_1, \ldots f(M_n, x'))$ and notice that attacker(x) if and only if attacker(M_1), ..., attacker(M_n), and attacker(x')), or when the clause has no selected hypothesis. In the last case, this introduces an approximation.

  When x occurs in several **memberOptim** predicates, the transformation may introduce an approximation. (For example, consider p_1 and p_2 defined as above respectively using f_1 and f_2 as data constructors. Then $p_1(M, x)$ && $p_2(M', x)$ is never true: for it to be true, x should be at the same time $f_1(_, _)$ and $f_2(_, _)$.)

6.4 Referring to bound names in queries

Until now, we have considered queries that refer only to free names of the process (declared by **free**), for instance **query attacker**(s) when s is declared by **free** s:t [**private**]. It is in fact also possible to refer to bound names (declared by **new** n:t in the process) in queries. To distinguish them from free names, they are denoted by **new** n in the query. As an example, consider the following input file:

```
1 free c:channel.
2 fun h(bitstring):bitstring.
3
4 free n:bitstring.
5 query attacker(h((n,new n))).
6
7 process new n:bitstring; out(c,n)
```

in which the process constructs and outputs a fresh name. Observe that the free name n is distinct from the bound name n and the query evaluates whether the attacker can construct a hash of the free name paired with the bound name. When an identifier is defined as a free name and the same identifier is used to define a bound name, ProVerif produces a warning. Similarly, a warning is also produced if the same identifier is used by two names or variables within the same scope. For clarity, we strongly discourage this practice and promote the use of distinct identifiers.

The term **new** n in a query designates any name created at the restriction **new** n:t in the process. It is also possible to make it more precise which bound names we want to designate: if the restriction **new** n:t is in the scope of a variable x, we can write **new** $n[x = M]$ to designate any name created by the restriction **new** n:t when the value of x is M. This can be extended to several variables: **new** $n[x_1 = M_1, \ldots, x_n = M_n]$. (This is related to the internal representation of bound names in ProVerif. Essentially, names are represented as functions of the variables which they are in the scope of. For example, the name a in the process **new** a:nonce is not in the scope of any variables and hence the name is modeled without arguments as a[ ]; whereas the name b in the process **in**(c,(x: bitstring ,y: bitstring)); **new** b:nonce is in the scope of variables x, y and hence will be represented by b[x=M,y=N] where the terms M, N are the values of x and y at run time, respectively.) Consider, for example, the process:

```
1 free c:channel.
```

```
free A:bitstring.
event e(bitstring).
query event(e(new a[x=A;y=new B])).

process
  (in(c,(y:bitstring,x:bitstring));new a:bitstring; event e(a))
  | (new B:bitstring;out(c,B))
```

The query **query event**(e(**new** a[x=A;y=**new** B])) tests whether event e can be executed with argument a name created by the restriction **new** a:bitstring when x is A and y is a name created by the restriction **new** B:bitstring. In the example process, such an event can be executed.

Furthermore, in addition to the value of the variables defined above the considered restriction **new**, one can also specify the value of $!i$, which represents the session identifier associated with the i-th replication above the considered **new**, where i is a positive integer. (Replications are numbered from the top of the process: !1 corresponds to the first replication at the top of the syntax tree.) These session identifiers take a different value in each copy of the process created by the replication. It does not make much sense to give a non-variable value to these session identifiers, but they can be useful to designate names created in the same copy or in different copies of the process. Consider the following example:

```
free c:channel.
event e(bitstring,bitstring).
query i:sid; event(e(new A[!1 = i], new B[!1 = i])).

process
  (in(c,(y:bitstring,x:bitstring));event e(x,y))
  | ! (new A: bitstring; new B:bitstring;out(c,(A,B)))
```

The query **event**(e(**new** A[!1 = i], **new** B[!1 = i])) tests if one can execute events e(x,y) where x is a name created at the restriction **new** A: bitstring and y is a name created at **new** B: bitstring in the *same* copy as x (of session identifier i).

It is also possible to use **let** bindings in queries: **let** $x = M$ **in** binds the term M to x inside a query. Such bindings can be used anywhere in a query: they are added to reachability or correspondence queries, hypotheses, and facts in the grammar of correspondence assertions given in Figure 4.3. In such bindings, the term M must be a term without destructor. These bindings are specially useful in the presence of references to bound names. For instance, in the query **query attacker**(h((**new** n,**new** n))), the two occurrences of **new** n may represent different names created at the same restriction **new** n:t in the process. In contrast, in the query **query let** x = **new** n **in attacker**(h((x,x))), x represents any name created at the restriction **new** n:t and (x,x) is a pair containing twice the *same* name. Let bindings **let** $x = M$ **in** therefore allow us to designate several times exactly the same value, even if the term M may designate several possible values due to the use of the **new** n construct.

References to bound names in queries were used, for instance, in [BC08].

6.5 Exploring correspondence assertions

ProVerif allows the user to examine which events must be executed before reaching a state that falsifies the current query. The syntax **putbegin event**:e instructs ProVerif to test which events $e(\ldots)$ are needed in order to falsify the current query. This means that when an event e needs to be executed to trigger another action, a begin fact begin(e(...)) is going to appear in the hypothesis of the corresponding clause. This is useful when the exact events that should appear in a query are unknown. For instance, with the query

query x:bitstring; **putbegin event**:e; **event**($e'(x)$).

ProVerif generates clauses that conclude end($e'(M)$) (meaning that the event e' has been executed), and by manual inspection of the facts begin($e(M')$) that occur in their hypothesis, one can infer the full query:

query $x_1 : t_1, \ldots, x_n : t_n$; **event**($e'(\ldots)$) $\Longrightarrow$ **event**($e(\ldots)$).

As an example, let us consider the process:

```
free c:channel.
fun h(bitstring):bitstring.

event e(bitstring).
event e'(bitstring).

query x:bitstring; putbegin event:e; event (e'(x)).

process
  new s:bitstring;
  (
    event e(s);
    out(c,h(s))
  ) | (
    in(c,=h(s));
    event e'(h(s))
  )
```

ProVerif produces the output:

```
...
-- Query putbegin event:e; not event(e'(x_5))
Completing...
Starting query not event(e'(x_5))
goal reachable: begin(e(s_4[])) -> end(e'(h(s_4[])))
...
```

We can infer that the following correspondence assertion is satisfied by the process:

```
query x:bitstring; event (e'(h(x))) ==> event(e(x)).
```

This technique has been used in the verification of a certified email protocol, which can be found in subdirectory `examples/pitype/certified-mail-AbadiGlewHornePinkas/` (if you installed by OPAM in the switch ⟨*switch*⟩, the directory `~/.opam/`⟨*switch*⟩`/doc/proverif/examples/pitype/certified-mail-AbadiGlewHornePinkas/`).

6.6 ProVerif options

In this section, we discuss the command-line arguments and settings of ProVerif. The default behavior of ProVerif has been optimized for standard use, so these settings are not necessary for basic examples.

6.6.1 Command-line arguments

The syntax for the command-line is

`proverif [options]` ⟨*filename*⟩

where `proverif` is ProVerif's binary, ⟨*filename*⟩ is the input file, and the command-line parameters `[options]` are of the following form:

- `-in [format]`
  Choose the input format (`horn`, `horntype`, `pi`, or `pitype`). When the `-in` option is absent, the input format is chosen according to the file extension, as detailed below. The input format described in this manual is the typed pi calculus, which corresponds to the option `-in pitype`, and is the default when the file extension is `.pv`. We recommend using this format. The other formats are no longer actively developed. Input may also be provided using the untyped pi calculus (option `-in pi`, the default when the file extension is `.pi`), typed Horn clauses (option `-in horntype`, the

default when the file extension is `.horntype`), and untyped Horn clauses (option `-in horn`, the default for all other file extensions). The untyped Horn clauses and the untyped pi calculus input formats are documented in the file `docs/manual-untyped.pdf`.

- `-out [format]`
  Choose the output format, either `solve` (analyze the protocol) or `spass` (stop the analysis before resolution, and output the clauses in the format required for use in the Spass first-order theorem prover, see `http://www.spass-prover.org/`). The default is `solve`. When you select `-out spass`, you must add the option `-o [filename]` to specify the file in which the clauses will be output.

- `-TulaFale [version]`
  For compatibility with the web service analysis tool TulaFale (see the tool download at `http://research.microsoft.com/projects/samoa/`). The version number is the version of TulaFale with which you would like compatibility. Currently, only version 1 is supported.

- `-lib [filename]`
  Specify a particular library file. Library files may contain declarations (including process macros). They are therefore useful for code reuse. Library files must be given the file extension `.pvl`, and this must be omitted from `[filename]`. For example, the library file `crypto.pvl` would be specified as `-lib crypto`. This option is intended for compatibility with CryptoVerif.

- `-color`
  Display a colored output on terminals that support ANSI color codes. (Will result in a garbage output on terminals that do not support these codes.) Unix terminals typically support ANSI color codes. For emacs users, you can run ProVerif in a shell buffer with ANSI color codes as follows:

  - start a shell with `M-x` shell
  - load the `ansi-color` library with `M-x load-library RET ansi-color RET`
  - activate ANSI colors with `M-x ansi-color-for-comint-mode-on`
  - now run ProVerif in the shell buffer

  You can also activate ANSI colors in shell buffers by default by adding the following to your `.emacs`:

  ```
  (autoload 'ansi-color-for-comint-mode-on "ansi-color" nil t)
  (add-hook 'shell-mode-hook 'ansi-color-for-comint-mode-on)
  ```

- `-graph [directory]`
  This option is available only when the command-line option `-html [directory]` is not set. It generates PDF files containing graphs representing traces of attacks that ProVerif had found. These PDF files are stored in the specified directory. That directory must already exist. By default, graphviz is used to create these graphs from the dot files generated by ProVerif. However, the user may specify a command of his own choice to generate graphs with the command line argument `-commandLineGraph`. Two versions of the graphs are available: a standard and a detailed version. The detailed version is built when **set** traceDisplay = long. has been added to the input file.

- `-html [directory]`
  This option is available only when the command-line option `-graph [directory]` is not set. It generates HTML output in the specified directory. That directory must already exist. ProVerif may overwrite files in that directory, so you should create a fresh directory the first time you use this option. You may reuse the same directory for several runs of ProVerif if you do not want to keep the output of previous runs.

  ProVerif includes a CSS file `cssproverif.css` in the main directory of the distribution. You should copy that file to `[directory]`. You may edit it to suit your preferences if you wish.

  After running ProVerif, you should open the file `[directory]/index.html` with your favorite web browser to display the results.

If graphviz is installed and you did not specify a command line with the option `-commandLineGraph`, then drawings of the traces are available by clicking on `graph trace`. Two versions of the drawings are available: a standard and a detailed version. The detailed version is built when **set** traceDisplay = long. has been added to the input file.

- `-commandLineGraph [command line]`
  The option `-graph [directory]` or the option `-html [directory]` must be set. The specified command line is called for each attack trace found by ProVerif. It should contain the string `''%1''` which will be replaced by the name of the file in which ProVerif stores the graphical respresentation of the attack, without its `.dot` extension. For example, if you give the command line option `-commandLineGraph "dot -Tsvg %1.dot -o %1.svg"`, graphviz will generate a SVG file (instead of a PDF file) for each attack found by ProVerif.

- `-help` or `--help`
  Display a short summary of command-line options

6.6.2 Settings

The manner in which ProVerif performs analysis can be modified by the use of parameters defined in the form **set** ⟨name⟩ = ⟨value⟩. The parameters below are supported, where the default value is the first mentioned. ProVerif also accepts no instead of false and yes instead of true.

Attacker configuration settings.

- **set** ignoreTypes = true. (or "**set** ignoreTypes = all.")
  set ignoreTypes = false. (or "**set** ignoreTypes = none." or "**set** ignoreTypes = **attacker**." for backward compatibility)

  Indicates how ProVerif behaves with respect to types. By default (**set** ignoreTypes = true.), ProVerif ignores types; that is, the semantics of processes ignores types: the attacker may build and send ill-typed terms and the processes do not check types. This setting allows ProVerif to detect type flaw attacks. With the setting (**set** ignoreTypes = false.), the protocol respects the type system. In practice, protocols can be implemented to conform to this setting by making sure that the type converter functions and the tuples are correctly implemented: the result of a type converter function must be different from its argument, different from values of the same type obtained without applying the type converter function, and must identify which type converter function was applied, and this information must be checked upon pattern-matching; a tuple must contain the type of its arguments together with their value, and this type information must also be checked upon pattern-matching. Provided there is a single type converter function from one type to another, this can be implemented by adding a tag that represents the type to each term, and checking in processes that the tags are correct. The attacker may change the tag in clear terms (but not under an encryption or a signature, for instance). However, that does not allow it to bypass the type system. (Processes will never inspect inside values whose content does not match the tag.)

  Note that static typing is always enforced; that is, user-defined input files must always be well-typed and ProVerif will report any type errors.

  When types are ignored (**set** ignoreTypes = true.), functions marked **typeConverter** are removed when generating Horn clauses, so that you get exactly the same clauses as if the **typeConverter** function was absent. (In other words, such functions are the identity when types are ignored.)

  When types are taken into account, the state space is smaller, so the verification is faster, but on the other hand fewer attacks are found. Some examples do not terminate with **set** ignoreTypes = true, but terminate with **set** ignoreTypes = false.

- **set attacker** = active.
  set attacker = passive.

  Indicates whether the attacker is active or passive. An active attacker can read messages, compute, and send messages. A passive attacker can read messages and compute but not send messages.

- **set** keyCompromise = none.
  set keyCompromise = approx.
  set keyCompromise = strict.

  By default (**set** keyCompromise = none.), it is assumed that session keys and more generally the session secrets are not a priori compromised. (The session secrets are all the names bound under a replication.) Otherwise, it is assumed that the session secrets of some sessions are compromised, that is, known by the attacker. Then ProVerif determines whether the secrets of other sessions can be obtained by the attacker. In this case, the names that occur in queries always refer to names of non-compromised sessions (the attacker has all names of compromised sessions), and the events that occur before an arrow ==> in a query are executed only in non-compromised sessions. With **set** keyCompromise = approx., the compromised sessions are considered as executing possibly in parallel with non-compromised ones. With **set** keyCompromise = strict., the compromised sessions are finished before the non-compromised ones begin. The chances of finding an attack are greater with **set** keyCompromise = approx.. (It may be a false attack due to the approximations made in the verifier.) Key compromise is incompatible with attack reconstruction; moreover, phases and synchronizations cannot be used with the key compromise parameter enabled, because key compromise introduces a two-phase process.

  Rather than using this setting, we recommend encoding the desired key compromise directly in the process that models the protocol, by outputting the compromised secrets on a public channel.

- **set** privateCommOnPublicFreeNames = true.
  set privateCommOnPublicFreeNames = false.

  By default (**set** privateCommOnPublicFreeNames = true.), ProVerif follows the applied pi calculus semantics, which allows both private communications and communications through the adversary on public channels.

  With the setting **set** privateCommOnPublicFreeNames = false, ProVerif considers that all communications on public free names always go through the adversary, so private communications are forbidden on such channels. This setting sometimes yields a faster analysis, when the queries aim to prove **attacker**(M) or the lemmas or axioms use **attacker**(M) as assumption.

Simplification of processes

- **set** simplifyProcess = true.
  set simplifyProcess = false.
  set simplifyProcess = interactive.

  This setting is useful for proofs of observational equivalences with **choice**. With the setting **set** simplifyProcess = true, in case ProVerif fails to prove the desired equivalence, it tries to simplify the given biprocess and to prove the desired property on the simplified process, which increases its chances of success. With the setting **set** simplifyProcess = false, ProVerif does not compute the simplified biprocesses. With the setting **set** simplifyProcess = interactive, an interactive 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 to select one of the simplified biprocesses for ProVerif to prove the equivalence.

- **set** rejectChoiceTrueFalse = true.
  set rejectChoiceTrueFalse = false.

  With the setting **set** rejectChoiceTrueFalse = true, ProVerif does not try to prove observational equivalence for simplified processes that still contain tests **if choice**[true, false] **then**, because the observational equivalence proof has little chance of succeeding in this case. With the setting **set** rejectChoiceTrueFalse = false, ProVerif still tries to observational equivalence for simplified processes that contain tests **if choice**[true, false] **then**.

- **set** rejectNoSimplif = true.
  set rejectNoSimplif = false.

With the setting **set** rejectNoSimplif = true, ProVerif does not try to prove observational equivalence for simplified processes, when simplification has not managed to merge at least two branches of a test or to decompose a **let** f (...) = f (...) **in**. With the setting **set** rejectNoSimplif = false, ProVerif still tries to observational equivalence for these processes.

Verification of predicate definitions

- **set** predicatesImplementable = check.
  set predicatesImplementable = nocheck.

  Sets whether ProVerif should check that predicate calls are implementable. See Section 6.3 for more details on this check. It is advised to leave the check turned on, as it is by default. Otherwise, the semantics of the processes may not be well-defined.

Patterns with diff or choice

- **set** allowDiffPatterns = false.
  set allowDiffPatterns = true.

  The setting allowDiffPatterns = true enables an extension of ProVerif that allows **diff** or **choice** inside patterns, so that one can write for instance

```
in(c, diff[x:T,y:T]); ...
in(c, diff[x,y]:T); ...
let diff[x,y] = M in ...
```

  or equivalent processes with **choice** instead of **diff**.

  With this extension, the obtained biprocess no longer comes from two independent processes. Indeed, this extension allows extracting each component of a received biterm, and mixing these components together, for instance using the first component in the second biprocess or comparing the first and the second components. This extension is mainly useful for generating possibly infinite families of static equivalences: the biprocess outputs the pairs of messages that should be indistinguishable. An example of application is the verification of frame opacity [HBD19, BDM20].

  This extension is disabled by default.

Induction and lemma settings (see Sections 6.1 and 6.2)

- **set** inductionQueries = false.
  set inductionQueries = true.

  When true, ProVerif proves all the queries by induction.

- **set** inductionLemmas = false.
  set inductionLemmas = true.

  When true, ProVerif proves all the lemmas by induction.

- **set** saturationApplication = instantiate.
  set saturationApplication = full.
  set saturationApplication = none.
  set saturationApplication = discard.

  By default (**set** saturationApplication = instantiate.), lemmas, axioms, and inductive hypotheses are only applied in the saturation procedure when they instantiate at least one variable. With saturationApplication = none, they are never applied during the saturation procedure. With saturationApplication = discard, they are only applied when they imply that the hypotheses of the clause are not satisfiable (hence discarding the clause). Finally, with saturationApplication = full, they are always applied.

- **set** verificationApplication = full.
  set verificationApplication = none.
  set verificationApplication = discard.
  set verificationApplication = instantiate.

  By default (**set** verificationApplication = full.), lemmas, axioms, and inductive hypotheses are always applied during the verification procedure. The different options have the same meaning as the ones for the setting saturationApplication but applied to the verification procedure.

Precision, performance, and termination settings. The performance settings may result in more or fewer false attacks, but they *never* sacrifice soundness. It follows that when ProVerif says that a property is satisfied, then the model really does guarantee that property, regardless of how ProVerif has been configured using the settings presented here.

- **set** preciseActions = false.
  set preciseActions = true.
  set preciseActions = trueWithoutArgsInNames.

  When true, ProVerif increases the precision of the solving procedure by ensuring that it only considers derivations where an input of the process has been uniquely instantiated for each execution of the considered input. Similary for **let** ... **suchthat** constructs and **get** ... **in** constructs. See Section 6.2 for more details. This setting increases precision possibly at the cost of performance and termination.

  When trueWithoutArgsInNames, in addition to the changes above, the fresh names created by **new** have as default arguments only session identifiers. This is usually sufficient thanks to the additional precision brought by preciseActions. (See the setting movenew below for an explanation of these arguments.)

- **set** movenew = false.
  set movenew = true.

  Sets whether ProVerif should try to move restrictions under inputs, to have a more precise analysis (**set** movenew = true.), or leave them where the user has put them (**set** movenew = false.). Internally, ProVerif represents fresh names by functions of the variables bound above the **new**. Adjusting these arguments allows one to change the precision of the analysis: the more arguments are included, the more precise the analysis is, but also the more costly in general. The setting (**set** movenew = true.) yields the most precise analysis. You can fine-tune the precision of the analysis by keeping the default setting and moving **new**s manually in the input process.

- **set** movelet = true.
  set movelet = false.

  When movelet = true, ProVerif moves **let**s downwards in the process as much as possible. By computing variables as late as possible, that can reduce the case distinctions that are made in the generation of clauses on some parts of the process, and thus generate fewer clauses and speed up resolution. When movelet = false, this transformation is not performed.

- **set** maxDepth = none.
  set maxDepth = n.

  Do not limit the depth of terms (none) or limit the depth of terms to n, where n is an integer. A negative value means no limit. When the depth is limited to n, all terms of depth greater than n are replaced with new variables. (Note that this makes clauses more general.) Limiting the depth can be used to enforce termination of the solving process, at the cost of precision. This setting is not recommended: it often causes too much imprecision. Using **nounif** (see Section 6.7.2) is delicate but may be more successful in practice.

- **set** maxHyp = none.
  set maxHyp = n.

Do not limit the number of hypotheses of clauses (none) or limit it to n, where n is an integer. A negative value means no limit. When the number of hypotheses is limited to n, arbitrary hypotheses are removed from clauses, so that only n hypotheses remain. Limiting the number of hypotheses can be used to enforce termination of the solving process at the cost of precision (although in general limiting the depth by the above declaration is enough to obtain termination). This setting is not recommended.

- **set** selFun = TermMaxsize.
  set selFun = Term.
  set selFun = NounifsetMaxsize.
  set selFun = Nounifset.

  Chooses the selection function that governs the resolution process. All selection functions avoid unifying on facts indicated by a **nounif** declaration (see Section 6.7.2). Nounifset does exactly that. Term automatically avoids some other unifications, to help termination, as determined by some heuristics. NounifsetMaxsize and TermMaxsize choose the fact of maximum size when there are several possibilities. This choice sometimes gives impressive speedups.

  When the selection function is set to Nounifset or NounifsetMaxsize; ProVerif will display a warning, and wait for a user response, when ProVerif thinks the solving process will not terminate. This behavior can be controlled by the following additional setting.

  - **set** stopTerm = true.
    set stopTerm = false.

    Display a warning and wait for user answer when ProVerif thinks the solving process will not terminate (true), or go on as if nothing had happened (false). (We reiterate that these settings are only available when the selection function is set to either Nounifset or NounifsetMaxsize.)

- **set** redundancyElim = best.
  set redundancyElim = simple.
  set redundancyElim = no.

  An elimination of redundant clauses has been implemented: when a clause without selected hypotheses is derivable from other clauses without selected hypothesis, it is removed. With redundancyElim = simple, this is applied for newly generated clauses. With redundancyElim = best, this is also applied on the reachable goals before proving a query. With redundancyElim = no, this is never applied.

  Detecting redundant clauses takes time, but redundancy elimination may also speed up the resolution when it eliminates clauses and simplify the final result of ProVerif. The consequences on speed depend on the considered protocol. By default, **set** redundancyElim = best.

- **set** redundantHypElim = beginOnly.
  set redundantHypElim = false.
  set redundantHypElim = true.

  When a clause is of the form H && H' -> C, and there exists σ such that $\sigma H \subseteq H'$ and σ does not change the variables of H' and C, then the clause can be replaced with H' -> C (since there are implications in both directions between these clauses).

  This replacement is done when redundantHypElim is set to true, or when it is set to beginOnly and H contains a begin fact (which is generated when events occur after ==> in a query) or a blocking fact. Indeed, testing this property takes time, and slows down small examples. On the other hand, on big examples, in particular when they contain many events (or blocking facts), this technique can yield huge speedups.

- **set** removeEventsForLemma = false.
  set removeEventsForLemma = true.

  When removeEventsForLemma = true, ProVerif removes event from clauses that are used only for applying lemmas, but do not seem useful anymore because the lemmas have already been applied

or they will never be applicable using this event. It speeds up resolution and may even allow it to terminate. However, this removal is slightly approximate, so it may in some rare cases prevent a useful application of a lemma. (ProVerif remains sound in this case, but may lose precision.) This option applies only to lemmas that are not declared with an explicit **removeEvents** or **keepEvents** option.

- **set** eqInNames = false.
  set eqInNames = true.

  This setting will probably not be used by most users. It influences the arguments of the functions that represent fresh names internally in ProVerif. When eqInNames = false, these arguments consist of variables defined by inputs, indices associated to replications, and terms that contain destructors defined by several rewrite rules, but do not contain other computed terms since their value is fixed knowing the arguments already included. When eqInNames = true, these arguments additionally include terms that contain constructors associated with several rewrite rules due to the equational theory. Because of these several rewrite rules, these terms may reduce to several syntactically different terms, which are all equal modulo the equational theory. In some rare examples, eqInNames = true speeds up the analysis because equality of the fresh names then implies that these terms are syntactically equal, so fewer clauses are considered. However, for technical reasons, eqInNames = true is incompatible with attack reconstruction.

- **set** preciseLetExpand = true.
  set preciseLetExpand = false.

  This setting modifies the expansion of terms **let** ... = ... **in** ... **else** ... . By default (with the setting **set** preciseLetExpand = true.), they are expanded into processes **let** that define variables but never fail, and the test that decides whether the **in** branch or **else** branch is taken is encoded as a term. This expansion is more precise when proving observational equivalence with **choice**, but leads to a slower generation of the clauses for some examples. With the setting **set** preciseLetExpand = false., terms **let** ... = ... **in** ... **else** ... are transformed into processes **let** that directly determine which branch is taken.

- **set** expandSimplifyIfCst = true.
  set expandSimplifyIfCst = false.

  This setting modifies the expansion of terms to into processes. With expandSimplifyIfCst = true, if a process **if** M **then** P **else** Q occurs during this expansion and M is true, then this process is transformed into P. If this process occurs and M is false, then this process is transformed into Q. This transformation is useful because the expansion of terms into processes may introduce such tests with constant conditions. However, the transformation will be performed even if the constant was already there in the initial process, which may cut part of the process, and for instance remove restrictions that occur in the initial process and are needed for some queries or secrecy assumptions.

  With the setting **set** expandSimplifyIfCst = false., this transformation is not performed.

- **set** nounifIgnoreAFewTimes = none.
  set nounifIgnoreAFewTimes = auto.
  set nounifIgnoreAFewTimes = all.

  This setting controls the default behavior of **nounif** declarations with respect to the **ignoreAFewTimes** option (see Section 6.7.2). When nounifIgnoreAFewTimes = none, the **nounif** declarations do not have the **ignoreAFewTimes** option unless it is explicitly mentioned. When nounifIgnoreAFewTimes = auto, the **nounif** declarations automatically guessed by ProVerif during resolution have the **ignoreAFewTimes** option. When nounifIgnoreAFewTimes = all, all **nounif** declarations have the **ignoreAFewTimes** option.

- **set** nounifIgnoreNtimes = n. (default: $n = 1$)

  This option determines how many times **nounif** declarations with option **ignoreAFewTimes** are ignored. By default, they are ignored once.

- **set** symbOrder = "$f_1 > \cdots > f_n$".

  ProVerif uses a lexicographic path ordering in order to prove termination of convergent equational theories. By default, it uses a heuristic to build the ordering of function symbols underlying this lexicographic path ordering. This setting allows the user to set this ordering of function symbols.

- **set** featureFuns = true.
  set featureFuns = false.

  set featureNames = false.
  set featureNames = true.

  set featurePredicates = true.
  set featurePredicates = false.

  set featureEvents = true.
  set featureEvents = false.

  set featureTables = true.
  set featureTables = false.

  set featureDepth = false.
  set featureDepth = true.

  set featureWidth = false.
  set featureWidth = true.

  ProVerif uses an indexing mechanism based on features of clauses [Sch13] to quickly eliminate clauses for which the subsumption test is certainly false. These settings determine which features are used in this indexing mechanism: ProVerif uses as features the number of hypotheses of the clause, and the number of occurrences of each predicate (when featurePredicates = true), event (when featureEvents = true), and table (when featureTables = true) symbol in the hypothesis and in the conclusion of the clause. It also uses, for each function (when featureFuns = true) and name (when featureNames = true) symbol, either its number of occurrences and its maximum depth in the hypothesis and in the conclusion of the clause (when featureDepth = false) or the number of its occurrences at each depth in the hypothesis and in the conclusion of the clause (when featureDepth = true), as well as its number of occurrences at each width in the hypothesis and in the conclusion of the clause (when featureWidth = true; a symbol f occurs at width w in a fact F when it is at the root of the w-th argument of the symbol immediately above f, i.e., $F = C[g(M_1, \ldots, M_{w-1}, f(\ldots), M_{w+1}, \ldots, M_n)]$). Finally, ProVerif also uses the total number of occurrences in the hypothesis and in the conclusion of the clause of any function or name symbol not used in the previous features.

Attack reconstruction settings.

- **set** simplifyDerivation = true.
  set simplifyDerivation = false.

  Should the derivation be simplified by removing duplicate proofs of the same **attacker** facts?

- **set** abbreviateDerivation = true.
  set abbreviateDerivation = false.

  When abbreviateDerivation = true, ProVerif defines symbols to abbreviate terms that represent names $a[\ldots]$ before displaying the derivation, and uses these abbreviations in the derivation. These abbreviations generally make the derivation easier to read by reducing the size of terms.

- **set** explainDerivation = true.
  set explainDerivation = false.

  When explainDerivation = true, ProVerif explains in English each step of the derivation (returned in case of failure of a proof). This explanation refers to program points in the given process. When explainDerivation = false, it displays the derivation by referring to the clauses generated initially.

- **set** reconstructTrace = n. (default $n = 4$)
  set reconstructTrace = true.
  set reconstructTrace = false.

  With reconstructTrace = true, when a query cannot be proved, the tool tries to build a pi calculus execution trace that is a counter-example to the query [AB05c]. With reconstructTrace = false, the tool does not try to reconstruct a trace. With reconstructTrace = n, it tries to reconstruct a trace at most n times for each query.

  Trace reconstruction is currently incompatible with key compromise (that is, when keyCompromise is set to either approx or strict).

  Moreover, for **noninterf** and **choice**, it reconstructs a trace, but this trace may not always prove that the property is wrong: for **noninterf**, it reconstructs a trace until a program point at which the process behaves differently depending on the value of the secret (takes a different branch of a test, for instance), but this different behavior is not always observable by the attacker; similarly, for **choice**, it reconstructs a trace until a program point at which the process using the first argument of **choice** behaves differently from the process using the second argument of **choice**.

- **set** unifyDerivation = true.
  set unifyDerivation = false.

  When set to true, activates a heuristic that increases the chances of finding a trace that corresponds to a derivation. This heuristic unifies messages received by the same input (same occurrence and same session identifiers) in the derivation. Indeed, these messages must be equal if the derivation corresponds to a trace.

- **set** reconstructDerivation = true.
  set reconstructDerivation = false.

  When a fact is derivable, should we reconstruct the corresponding derivation? (This setting has been introduced because in some extreme cases reconstructing a derivation can consume a lot of memory.)

- **set** displayDerivation = true.
  set displayDerivation = false.

  Should the derivation be displayed? Disabling derivation display is useful for very big derivations.

- **set** traceBacktracking = true.
  set traceBacktracking = false.

  Allow or disable backtracking when reconstructing traces. In most cases, when traces can be found, they are found without backtracking. Disabling backtracking makes it possible to display the trace during its computation, and to forget previous states of the trace. This reduces memory consumption, which can be necessary for reconstructing very big traces.

Swapping settings.

- **set** interactiveSwapping = false.
  set interactiveSwapping = true.

  By default, in order to prove observational equivalence in the presence of synchronization (see Section 4.3.2), ProVerif tries all swapping strategies. With the setting interactiveSwapping = true, it asks the user which swapping strategy to use.

- **set** swapping = "swapping stragegy".

  This settings determines which swapping strategy to usein order to prove observational equivalence in the presence of synchronization. See Section 4.3.2 for more details, in particular the syntax of swapping strategies.

Display settings.

- **set** traceDisplay = short.
  set traceDisplay = long.
  set traceDisplay = none.

  Choose the format in which the trace is displayed after trace reconstruction. By default (traceDisplay = short.), outputs the labels of a labeled reduction. With **set** traceDisplay = long., outputs the current state before each input and before and after each I/O reduction, as well as the list of all executed reductions. With **set** traceDisplay = none., the trace is not displayed.

- **set** verboseClauses = none.
  set verboseClauses = explained.
  set verboseClauses = short.

  When verboseClauses = none, ProVerif does not display the clauses it generates. When verboseClauses = short, it displays them. When verboseClauses = explained, it adds an English sentence after each clause it generates to explain where this clause comes from.

- **set** verboseLemmas = false.
  set verboseLemmas = true.

  When verboseLemmas = true, ProVerif displays the lemmas, axioms and inductive hypotheses that are used during the saturation and/or the verification procedures (see Sections 6.1 and 6.2).

- **set** abbreviateClauses = true.
  set abbreviateClauses = false.

  When abbreviateClauses = true, ProVerif defines symbols to abbreviate terms that represent names $a[\ldots]$ and uses these abbreviations in the display of clauses. These abbreviations generally make the clauses easier to read by reducing the size of terms.

- **set** removeUselessClausesBeforeDisplay = true.
  set removeUselessClausesBeforeDisplay = false.

  When removeUselessClausesBeforeDisplay = true, ProVerif removes subsumed clauses and tautologies from the initial clauses before displaying them, to avoid showing many useless clauses. When removeUselessClausesBeforeDisplay = false, all generated clauses are displayed.

- **set** verboseEq = true.
  set verboseEq = false.

  Display information on handling of equational theories when true.

- **set** verboseTerm = true.
  set verboseTerm = false.

  Display information on termination when true (changes in the selection function to improve termination; termination warnings).

- **set** verboseRules = false.
  set verboseRules = true.

  Display the number of clauses every 200 clause created during the solving process (false) or display each clause created during the solving process (true).

- **set** verboseBase = false.
  set verboseBase = true.

  When true, display the current set of clauses at each resolution step during the solving process.

- **set** verboseRedundant = false.
  set verboseRedundant = true.

  Display eliminated redundant clauses when true.

- **set** verboseCompleted = false.
  set verboseCompleted = true.

  Display completed set of clauses after saturation when true.

6.7 Theory and tricks

In this section, we discuss tricks to get the most from ProVerif for advanced users. These tricks may improve performance and aid termination. We also propose alternative ways to encode protocols and pi calculus encodings for some standard features. We also detail sources of incompleteness of ProVerif, for a better understanding of when and why false attacks happen.

User tricks. You are invited to submit your own ProVerif tricks, which we may include in future revisions of this manual.

6.7.1 The resolution strategy of ProVerif

ProVerif represents protocols internally by Horn clauses, and the resolution algorithm [Bla11] combines clauses: from two clauses R and R', it generates a clause $R \circ_{F_0} R'$ defined as follows

$$\frac{R = H \text{ -> } C \qquad R' = F_0 \text{ \&\& } H' \text{ -> } C'}{R \circ_{F_0} R' = \sigma H \text{ \&\& } \sigma H' \text{ -> } \sigma C'}$$

where σ is the most general unifier of C and F_0, C is selected in R, and F_0 is selected in R'. The selected literal of each clause is determined by a selection function, which can be chosen by **set** selFun = *name*., where *name* is the name of the selection function, Nounifset, NounifsetMaxsize, Term, or TermMaxsize. The selection functions work as follows:

- Hypotheses of the form $p(\ldots)$ when p is declared with option **block** and internal predicates begin and testunif are unselectable. (The predicate testunif is handled by a specific internal treatment. The predicates with option **block** and the predicate begin have no clauses that conclude them; the goal is to produce a result valid for any definition of these predicates, so they must not be selected.)

  The conclusion bad is also unselectable. (The goal is to determine whether bad is derivable, so we should select a hypothesis if there is some, to determine whether the hypothesis is derivable.)

  Facts $p(x)$ when p is an internal predicate **attacker** or comp are also unselectable. (Due to data-decomposition clauses, selecting such facts would lead to non-termination.)

  Unselectable hypotheses are never selected. An unselectable conclusion is selected only when all hypotheses are unselectable (or there is no hypothesis).

- If there is a selectable literal, the selection function selects the literal of maximum weight among the selectable literals. In case several literals have the maximum weight, the conclusion is selected in priority if it has the maximum weight, then the first hypothesis with maximum weight is selected. The weight of each literal is determined as follows:

  - If the selection function is Term or TermMaxsize (the default), and a hypothesis is a *looping instance* of the conclusion, then the conclusion has weight -7000. (A fact F is a *looping instance* of a fact F' when there is a substitution σ such that $F = \sigma F'$ and σ maps some variable x to a term that contains x and is not a variable. In this case, repeated instantiations of F' by σ generate an infinite number of distinct facts $\sigma^n F'$.)

    The goal has weight -3000. (The goal is the fact for which we want to determine whether it is derivable or not. It appears as a conclusion in the second stage of ProVerif's resolution algorithm.)

    If the conclusion is a fact F whose weight has been manually set by a declaration **nounif** ... [**conclusion**] (see Section 6.7.2), then the conclusion has the weight in question.

    In all other cases, the conclusion has weight -1.

  - If the selection function is Term or TermMaxsize (the default), and the conclusion is a looping instance of a hypothesis, then this hypothesis has weight -7000.

    If the hypothesis is a fact F whose weight has been set by a declaration **nounif** (see Section 6.7.2) or by a previous selection step (see below), then the hypothesis has the weight in question.

    All other hypotheses have as weight their size with the selection functions TermMaxsize (the default) and NounifsetMaxsize. They have weight 0 with the selection functions Term and Nounifset.

- If the selection function is Term or TermMaxsize (the default) and the conclusion is selected in a clause, then for each hypothesis F of that clause such that the conclusion C is a looping instance of F ($C = \sigma F$), the weight of hypotheses $\sigma' F$, where σ and σ' have disjoint supports, is set to -5000 for the rest of the resolution. (σ and σ' have disjoint supports means that, if σx is not a variable, then $\sigma' x$ must be a variable.)

The selection functions Term and TermMaxsize try to favor termination by auto-detecting loops and tuning the selection function to avoid them. For instance, suppose that the conclusion is a looping instance of a hypothesis, so the clause is of the form H && F -> σF.

- Assume that F is selected in this clause, and there is a clause H' -> F', where F' unifies with F, and the conclusion is selected in H' -> F'. Let σ' be the most general unifier of F and F'. So the algorithm generates:

  $\sigma' H'$ && $\sigma' H$ -> $\sigma'\sigma F$
  ...
  $\sigma' H'$ && $\sigma' H$ && $\sigma'\sigma H$ &&... && $\sigma'\sigma^{n-1} H$ -> $\sigma'\sigma^n F$

  assuming that the conclusion is selected in all these clauses, and that no clause is removed because it is subsumed by another clause. So the algorithm would not terminate. Therefore, in order to avoid this situation, we should avoid selecting F in the clause H && F -> σF. That is why we give F weight -7000 in this case. A symmetric situation happens when a hypothesis is a looping instance of the conclusion, so we give weight -7000 to the conclusion in this case.

- Assume that the conclusion is selected in the clause H && F -> σF, and there is a clause H' && $\sigma' F$ -> C (up to renaming of variables), where σ' commutes with σ (in particular, when σ and σ' have disjoint supports), and that $\sigma' F$ is selected in this clause. So the algorithm generates:

  $\sigma' H$ && $\sigma H'$ && $\sigma' F$ -> σC
  ...
  $\sigma' H$ && $\sigma'\sigma H$ && ... && $\sigma'\sigma^{n-1} H$ && $\sigma^n H'$ && $\sigma' F$ -> $\sigma^n C$

  assuming that $\sigma' F$ is selected in all these clauses, and that no clause is removed because it is subsumed by another clause. So the algorithm would not terminate. Therefore, in order to avoid this situation, if the conclusion is selected in the clause H && F -> σF, we should avoid selecting facts of the form $\sigma' F$, where σ' and σ have disjoint supports, in other clauses. That is why we automatically set the weight to -5000 for these facts.

Obviously, these heuristics do not avoid all loops. One can use manual **nounif** declarations to tune the selection function further, as explained in Section 6.7.2.

The selection functions TermMaxsize and NounifsetMaxsize preferably select large facts. This can yield important speed-ups for some examples.

6.7.2 Performance and termination

Secrecy assumptions

Secrecy assumptions may be added to ProVerif scripts in the form:

not $x_1 : t_1, \ldots, x_n : t_n$; F .

which states that F cannot be derived, where F can be a fact **attacker**(M), **attacker**(M) **phase** n, **mess**(N, M), **mess**(N, M) **phase** n, **table**($d(M_1, \ldots, M_n)$), **table**($d(M_1, \ldots, M_n)$) **phase** n as defined in Figure 4.3 or a user-defined predicate $p(M_1, \ldots, M_k)$ (see Section 6.3). When F contains variables, the secrecy assumption **not** $x_1 : t_1, \ldots, x_n : t_n$; F. means that no instance of F is derivable.

ProVerif can then optimize its internal clauses by removing clauses that contain F in hypotheses, thus simplifying the clause set and resulting in a performance advantage. The use of secrecy assumptions preserves soundness because ProVerif also checks that F cannot be derived; if it can be derived, ProVerif fails with an error message. Secrecy assumptions can be extended using the binding **let** $x = M$ **in** and bound names designated by **new** $a[\ldots]$ as discussed in Section 6.4; these two constructs are allowed as part of F.

The name "secrecy assumptions" comes from the particular case

```
not attacker(M).
```

which states that **attacker**(M) cannot be derived, that is, M is secret.

Secrecy assumptions may also be added when proving equivalence between two processes P and Q. For example, in the declaration

```
free k:bitstring [private].
not attacker(k).
```

the assumption **not attacker**(k) indicates that k cannot be deduced by the attacker in P and Q at the same time. Secrecy assumptions can also differ between P and Q using **choice**. The declaration **not attacker**(**choice**[k1,k2]) indicates that the attacker cannot deduce k1 in P and k2 in Q at the same time. Note that if the attacker can deduce k1 in P but is not able to deduce k2 in Q then the secrecy assumption is satisfied. As such, it is possible to declare a secrecy assumption only for P as follows.

```
not x:bitstring; attacker(choice[k,x]).
```

This secrecy assumption intuitively only indicates that the attacker cannot deduce k in P but does not say anything about Q.

Grouping queries

As mentioned in Section 6.1, queries may also be stated in the form:

query $x_1 : t_1, \ldots, x_m : t_m$; $q_1; \ldots; q_n$.

where each q_i is a query as defined in Figure 4.3, or a **putbegin** declaration (see Section 6.5). A single **query** declaration containing $q_1; \ldots; q_n$ is evaluated by building one set of clauses and performing resolution on it, whilst independent query declarations

query $x_1 : t_1, \ldots, x_m : t_m$; q_1 .
...
query $x_1 : t_1, \ldots, x_m : t_m$; q_n .

are evaluated by rebuilding a new set of clauses from scratch for each q_i. So the way queries are grouped influences the sharing of work between different queries, and therefore performance. For optimization, one should group queries that involve the same events; but separate queries that involve different events, because the more events appear in the query, the more complex the generated clauses are, which can slow down ProVerif considerably, especially on complex examples. If one does not want to optimize, one can simply put a single query in each **query** declaration.

Tuning the resolution strategy.

The resolution strategy can be tuned using declarations:

nounif $x_1 : t_1, \ldots, x_k : t_k$; F/w $[o_1, \ldots, o_n]$.

The fact F can be **attacker**(M), **attacker**(M) **phase** n, **mess**(N, M), **mess**(N, M) **phase** n, **table**($d(M_1, \ldots, M_m)$), **table**($d(M_1, \ldots, M_m)$) **phase** n as defined in Figure 4.3 or a user-defined predicate $p(M_1, \ldots, M_m)$ (see Section 6.3), and F can also include the construct **new** a[...] to designate

bound names and let bindings **let** $x = M$ **in** (see Section 6.4). The fact F may contain two kinds of variables: ordinary variables match only variables, while star variables, of the form $*x$ where x is a variable name, match any term.

The indications $x_1 : t_1, \ldots, x_k : t_k$ specify the types of the variables $x_1, \ldots, x_k$ that occur in F.

The declaration **nounif** F prevents ProVerif from resolving upon facts that match F. The integer w indicates how much ProVerif should avoid resolution upon facts that match F: the greater w, the more such resolutions will be avoided. Formally, the **nounif** declaration gives weight $-w$ to facts that match F in the resolution algorithm explained in Section 6.7.1. By default, only the weight of hypotheses that match F is modified; the weight of conclusions that match is left unchanged. The options $o_1, \ldots, o_n$ may modify that as detailed below. The minimum weight that can be set by **nounif** is -9999. If $-w \leq -10000$, the weight will be set to -9999. The integer w can be omitted, be removing $/w$ from the declaration. When w is not mentioned, w is implicitly set to 6000, so facts that match F have weight -6000. This weight is such that, by default, manual **nounif** declarations have priority over automatic weight assignments (weight -5000), but have lower priority than situations in which the conclusion is a looping instance of a hypothesis or conversely (weight -7000). One can adjust the weight manually to obtain different priority levels.

The options $o_1, \ldots, o_n$ specify further how the **nounif** declaration applies. The allowed options are:

- **hypothesis**, **conclusion**: When the option **conclusion** is *not* mentioned (e.g., **nounif** $x_1 : t_1, \ldots, x_k : t_k$; F/w [**hypothesis**]. or **nounif** $x_1 : t_1, \ldots, x_k : t_k$; F/w.), the declaration modifies the weight of hypotheses matching F and leaves the weight of conclusions matching F unchanged.

  When the option **conclusion** alone is mentioned (**nounif** $x_1 : t_1, \ldots, x_k : t_k$; F/w [**conclusion**].), the declaration modifies the weight of conclusions matching F and leaves the weight of hypotheses matching F unchanged.

  When the option **conclusion** and another option are mentioned (e.g., **nounif** $x_1 : t_1, \ldots, x_k : t_k$; F/w [**hypothesis**,**conclusion**].), the declaration modifies the weight of both hypotheses and conclusions matching F.

  For example, the declarations

  nounif $x_1 : t_1, \ldots, x_k : t_k$; F/w_1 .
  nounif $x_1 : t_1, \ldots, x_k : t_k$; F/w_2 [**conclusion**] .

  indicate that the weight of conclusions matching F is $-w_2$ whereas the weight of hypotheses matching F is $-w_1$.

- **ignoreAFewTimes**: The **nounif** declarations help the saturation procedure to terminate but they may lower the precision of ProVerif by preventing resolution steps. In the second stage of the resolution algorithm, i.e. after saturation has completed and when ProVerif determines whether the goal is derivable or not, we allow a hypothesis F in a clause $F \wedge H \rightarrow C$ matching a **nounif** declaration with option **ignoreAFewTimes** to be selected instead of selecting the conclusion C. To prevent the non-termination issue, such selection, which ignores the **nounif** declaration, can only happen a limited number of times, determined by the setting **set** nounifIgnoreNtimes $= n$. (By default, it happens only once.) When we resolve $F \wedge H \rightarrow C$ with $H' \rightarrow C'$ upon F ignoring a **nounif** declaration and that yields a clause $\sigma H' \wedge \sigma H \rightarrow \sigma C$, we consider that the **nounif** have already been ignored once for all facts in $\sigma H'$, so if **nounif** declarations could be ignored n times for F, then they can be ignored $n-1$ times for facts in $\sigma H'$. With the default setting, the **nounif** declarations can no longer be ignored for facts in $\sigma H'$.

  For example, consider a clause among the saturated clauses such that the conclusion is a looping instance of an hypothesis, so the clause is of the form $H \wedge F \rightarrow \sigma F$. Suppose now that during the second step of ProVerif's algorithm, a clause $F \rightarrow C$ needs to be resolved. Since σF is a looping instance of F, ProVerif would have automatically associated to F the weight -5000. In such a case, the conclusion C would be selected and the clause $F \rightarrow C$ would be considered as resolved. By declaring

  nounif $x_1 : t_1, \ldots, x_k : t_k$; F [**ignoreAFewTimes**] .

the hypothesis F in $F \to C$ is selected and allows a resolution step on F. Among possibly others, this will generate the clause $H \wedge F \to \sigma C$. However on this new clause, F will not be selected since it was already selected in $F \to C$ and this clause was used to generate $H \wedge F \to \sigma C$.

The option **ignoreAFewTimes** is best used when proving a query by induction as it may allow ProVerif to apply additional inductive hypotheses. Let us come back to the simplified Yubikey protocol `docs/ex_induction_group.pv` introduced in Section 6.1.

```
1  free c:channel.
2  free k:bitstring [private].
3  free d_P:channel [private].
4  free d_Q:channel [private].
5
6  fun senc(nat,bitstring):bitstring.
7  reduc forall K:bitstring,M:nat; sdec(senc(M,K),K) = M.
8
9  event CheckNat(nat).
10 event CheckNatQ(nat).
11
12 query i:nat;
13   event(CheckNat(i)) ==> is_nat(i);
14   event(CheckNatQ(i)) ==> is_nat(i);
15   mess(d_Q,i) ==> is_nat(i) [induction].
16
17 let P =
18   in(c,x:bitstring);
19   in(d_P,(i:nat,j:nat));
20   let j':nat = sdec(x,k) in
21   event CheckNat(i);
22   event CheckNat(j);
23   event CheckNatQ(j');
24   if j' > j
25   then out(d_P,(i+1,j'))
26   else out(d_P,(i,j)).
27
28 let Q =
29   in(d_Q,i:nat);
30   out(c,senc(i,k));
31   out(d_Q,i+1).
32
33 process
34   out(d_P,(0,0)) | out(d_Q,0) | ! P | ! Q
```

ProVerif is not able to prove the query **mess**(d_Q,i) ==> is_nat(i). By looking at the output of ProVerif on the execution of `docs/ex_induction_group.pv`, one can notice the following:

- ProVerif automatically assigns weight −5000 to **mess**(d_Q[],i_2), which is displayed **nounif** **mess**(d_Q[],i_2)/−5000, because the process Q generates the Horn clause **mess**(d_Q,i) −> **mess**(d_Q,i+1) where the conclusion is a looping instance of the hypothesis.
- ProVerif generates the reachable goal is_not_nat(i_2) && **mess**(d_Q[],i_2) −> **mess**(d_Q[],i_2). Because of the **nounif** declaration, ProVerif does not try to solve the hypothesis **mess**(d_Q[],i_2) which prevents it from proving the query.

To help ProVerif, one can add in the input file the **nounif** declaration on **mess**(d_Q[],i_2) with the option **ignoreAFewTimes** to allow ProVerif to resolve once upon the fact **mess**(d_Q[],i_2) during the verification procedure. In the file `docs/ex_induction_group_proof.pv`, we added the line

```
12 nounif i:nat; mess(d_Q,i) [ignoreAFewTimes].
```

By looking at the output of the execution of `docs/ex_induction_group_proof.pv`, one can notice that ProVerif is able to prove all queries and in particular, it generates the following reachable goals for the query **mess**(d_Q,i) ==> is_nat(i):

```
goal reachable: is_nat(i_2) && mess(d_Q[],i_2) -> mess(d_Q[],i_2 + 1)
goal reachable: mess(d_Q[],0)
```

- **inductionOn**=i: i must be a variable of type nat that has been declared in the environment of the **nounif** declaration (i.e., i is one of the variables $x_1, \ldots, x_k$), i must occur in F, but $*i$ must not occur in F.

  During the saturation procedure, when a clause is of the form $F\sigma_1 \wedge F\sigma_2 \wedge H \rightarrow C$ where H implies $i\sigma_1 \geq i\sigma_2$, the hypotheses containing the variable $i\sigma_2$ will be removed from the clause.

  Due to the presence of natural numbers, the saturation procedure may not terminate in some cases because it generates infinitely many clauses of the following form $H \wedge F(j_1) \wedge j_1 < j_2 \wedge F(j_2) \wedge j_2 < j_3 \wedge F(j_3) \wedge \ldots \rightarrow C$ that may not be removed by subsumption. This usually occurs when the facts $F(j_1), F(j_2), \ldots$ are unselectable (e.g. events) or match a **nounif**. In our example, adding the option **inductionOn**=i may ensure termination as it will simplify the clauses into $H \wedge F(j) \rightarrow C$.

  Note that similarly to the setting **set** maxHyp = n., this option can be used to ensure termination at the cost of precision. However, this option is best used when proving a query by induction, specifically when F is part of the goal of the query and when $i\sigma_1 \geq i\sigma_2$ implies that $F\sigma_1$ occurs after $F\sigma_2$. In such a case, the application of the inductive hypothesis on $F\sigma_1$ may be precise enough to prove the query by induction.

  When F contains multiple natural variables, one can add the option **inductionOn**=$\{i_1, \ldots, i_n\}$. In such a case, the hypotheses containing the variables $i_1\sigma_2, \ldots, i_n\sigma_2$ will be removed from the clause when H implies $\bigwedge_{k=1}^{n} i_k\sigma_1 \geq i_k\sigma_2$.

In order to determine the desired **nounif** declarations, one typically uses **set** verboseRules = true. to display the clauses generated by ProVerif. One can then observe the loops that occur, and one can try to avoid them by using a **nounif** declaration that prevents the selection of the literal that causes the loop.

Tagged protocols

A typical cause of non-termination of ProVerif is the existence of loops inside protocols. Consider for instance a protocol with the following messages:

```
B → A: senc(Nb, k)
A → B: senc(f(Nb), k)
```

(This example is inspired from the Needham-Schroeder shared-key protocol.) Suppose that A does not know the value of Nb (nonce generated by B). In this case, in A's role, Nb is a variable. Then, the attacker can send the second message to A as if it were the first one, and obtain as reply senc(f(f(Nb), k), which can itself be sent as if it were the first message, and so on, yielding to a loop that generates senc(f^n(Nb), k) for any integer n.

A way to avoid such loops is to add *tags*. A tag is a distinct constant for each application of a cryptographic primitive (encryption, signatures, ...) in the protocol. Instead of applying the primitive just to the initial message, one applies it to a pair containing a tag and the message. For instance, after adding tags, the previous example becomes:

```
B → A: senc((c0, Nb), k)
A → B: senc((c1, f(Nb)), k)
```

After adding tags, the second message cannot be mixed with the first one because of the different tags c0 and c1, so the previous loop is avoided. More generally, one can show that ProVerif always terminates for tagged protocols (modulo some restrictions on the primitives in use and on the properties that are proved) [BP05], [Bla09, Section 8.1]. Adding tags is a good design practice [AN96]: the tags facilitate the parsing of messages, and they also prevent type-flaw attacks (in which messages of different types

are mixed) [HLS00]. Tags are used in some practical protocols such as SSH. However, if one verifies a protocol with tags, one should implement the protocol with these tags: the security of the tagged protocol does not imply the security of the untagged version.

Position and arguments of new

Internally, fresh names created by **new** are represented as functions of the inputs located above that **new**. So, by moving **new** upwards or downwards, one can influence the internal representation of the names and tune the performance and precision of the analysis. Typically, the more the **new** are moved downwards in the process, the more precise and the more costly the analysis is. (There are exceptions to this general rule, see for example the end of Section 5.4.2.)

The setting **set** movenew = true. allows one to move **new** automatically downwards, potentially yielding a more precise analysis. By default, the **new** are left where they are, so the user can manually tune the precision of the analysis. Furthermore, it is possible to indicate explicitly at each replication which variables should be included as arguments in the internal representation of the corresponding fresh name: inside a process

$\mathbf{new}\ a\,[\,x_1,\ldots,x_n\,]:t$

means that the internal representation of names created by that restriction is going to include $x_1,\ldots,x_n$ as arguments. In any case, the internal representation of names always includes session identifiers (necessary for soundness) and variables needed to answer queries. These annotations are ignored in the case of observational equivalence proof between two processes (keyword **equivalence**) or when the biprocess is simplified before an observational equivalence proof. (Otherwise, the transformations of the processes might be prevented by these annotations.)

In general, we advise generating the fresh names by **new** when they are needed. Generating all fresh names at the beginning of the protocol is a bad idea: the names will essentially have no arguments, so ProVerif will merge all of them and the analysis will be so imprecise that it will not be able to prove anything. On the other hand, if the **new** take too many arguments, the analysis can become very costly or even not terminate. By the setting **set** verboseRules = true., one can observe the clauses generated by ProVerif; if these clauses contain names with very large arguments that grow more and more, moving **new** upwards or giving an explicit list of arguments to remove some arguments can improve the speed of ProVerif or make it terminate. The size of the arguments of names associated with random coins is the reason of the cost of the analysis in the presence of probabilistic encryption (see Section 4.2.3). When one uses function macros to represent encryption, one cannot easily move the **new** upwards. If needed, we advise manually expanding the encryption macro and moving the **new** that comes from this macro upwards or giving it explicit arguments.

Additional arguments of events

In order to prove injective correspondences such as

$$\mathbf{query}\ \ x_1:t_1,\ldots,x_n:t_n\,;\ \ \mathbf{inj\text{-}event}\,(e(M_1,\ldots,M_j))\ \Longrightarrow\ \mathbf{inj\text{-}event}\,(e'(N_1,\ldots,N_k))\,.$$

ProVerif adds a name with arguments to the injective event e' that occur after the arrow. Injectivity is proved when the session identifier of the event e occurs in those arguments. By default, ProVerif puts as many arguments as possible in that name. In some examples, this may lead to a loop or to a slow resolution. So ProVerif allows the user to specify which arguments should be given to that name, by adding the desired arguments between brackets in the process:

$$\mathbf{event}\,(e'(N'_1,\ldots,N'_k)\,)\,[\,x_1,\ldots,x_l\,]\,;\ \ P\,.$$

puts variables $x_1,\ldots,x_l$ as arguments in the name added to event e'. When no argument is mentioned:

$$\mathbf{event}\,(e'(N'_1,\ldots,N'_k)\,)\,[\,]\,;\ \ P\,.$$

ProVerif uses the arguments of the event instead, here $N'_1,\ldots,N'_k$. Typically, the arguments should include a fresh name (e.g., a nonce) created by the process that contains event e, and received by the process that contains event e', before executing e'.

6.7.3 Alternative encodings of protocols

Key distribution

In Section 4.1.5, we introduced tables and demonstrated their application for key distribution with respect to the Needham-Schroeder public key protocol (Sections 5.2 and 5.3). There are three further noteworthy key distribution methods which we will now discuss.

1. *Key distribution by scope.* The first alternative key distribution mechanism simply relies on variable scope and was used in our exemplar handshake protocol and in Section 5.1 without discussion. In this formalism, we simply ensure that the required keys are within the scope of the desired processes. The main limitation of this encoding is that it does not allow one to establish a correspondence between host names and keys for an unbounded number of hosts.

2. *Key distribution over private channels.* In an equivalent manner to tables, keys may be distributed over private channels.

   - Instead of declaring a table d, we declare a private channel by **free** cd: channel [**private**].
   - Instead of inserting an element, say (h,k), in table d, we output an unbounded number of copies of that element on channel cd by !**out**(cd, (h,k)). (The rest of the process should be in parallel with that output so that it does not get replicated as well.)
   - Instead of getting an element, say by **get**(d, (=h,k)) to get the key k for host h, we read on the private channel cd by **in**(cd, (=h,k:key)).

   With this encoding, all keys inserted in the table become available (in an unbounded number of copies) on the private channel cd.

   We present this encoding as an example of what can be done using private channels. It does not have advantages with respect to using the specific ProVerif constructs for inserting and getting elements of tables.

3. *Key distribution by constructors and destructors.* Finally, as we alluded in Section 3.1.1, private constructors can be used to model the server's key table. In this case, we make use of the following constructors and associated destructors:

```
type host.
type skey.
type pkey.

fun pk(skey): pkey.
fun fhost(pkey): host.
reduc x:pkey; getkey(fhost(x)) = x [private].
```

   The constructor fhost generates a host name from a public key, while the destructor getkey returns the public key from the host name. The constructor fhost is public so that the attacker can create host names for the keys it creates. The destructor getkey is private; this is not essential for public keys, but when this technique is used with long-term secret keys rather than public keys, it is important that getkey be private so that the attacker cannot obtain all secret keys from the (public) host names.

   This technique allows one to model an unbounded number of participants, each with a distinct key. This is however not necessary for most examples: one honest participant for each role is sufficient, the other participants can be considered dishonest and included in the attacker. An advantage of this technique is that it sometimes makes it possible for ProVerif to terminate while it does not terminate with the table of host names and keys used in previous chapters (because host names and keys that are complex terms may be registered by the attacker). For instance, in the file `examples/pitype/choice/NeedhamSchroederPK-corr1.pv` (if you installed by OPAM in the switch ⟨*switch*⟩, the file `~/.opam/`⟨*switch*⟩`/doc/proverif/examples/pitype/choice/NeedhamSchroederPK-corr1.pv`), we had to perform key registration in an earlier phase

than the protocol in order to obtain termination. Using the fhost/getkey encoding, we can obtain termination with a single phase (see `examples/pitype/choice/NeedhamSchroederPK-corr1-host-getkey.pv`). However, this encoding also has limitations: for instance, it does not allow the attacker to register several host names with the same key, which is sometimes possible in reality, so this can lead to missing some attacks.

Bound and private names

The following three constructs are essentially equivalent: a free name declared by **free** n:t, a constant declared by **const** n:t, and a bound name created by **new** n:t not under any replication in the process. They all declare a constant. However, in queries, bound names must be referred to by **new** n rather than n (see Section 6.4). Moreover, from a semantic point of view, it is much easier to define the meaning of a free name or a constant in a query than a reference to a bound name. (The bound name can be renamed, and the query is not in the scope of that name.) For this reason, we recommend using free names or constants rather than bound names in queries when possible.

6.7.4 Applied pi calculus encodings

The applied pi calculus is a powerful language that can encode many features (including arithmetic!), using private channels and function symbols. ProVerif cannot handle all of these encodings: it may not terminate if the encoding is too complex. It can still take advantage of the power of the applied pi calculus in order to encode non-trivial features. This section presents a few examples.

Asymmetric channels

Up to now, we have considered only public channels (on which the attacker can read and write) and private channels (on which the attacker can neither read nor write). It is also possible to encode asymmetric channels (on which the attacker can either read or write, but not both).

- A channel cwrite on which the attacker can write but not read can be encoded as follows: declare cwrite as a private channel by **free** cwrite:channel [**private**]. and add in your process !**in**(c, x:t); **out**(cwrite, x) where c is a public channel. This allows the attacker to send any value of type t on channel cwrite, and can be done for several types if desired. When types are ignored (the default), it in fact allows the attacker to send any value of any type on channel cwrite.
- A channel cread on which the attacker can read but not write can be encoded as follows: declare cread as a private channel by **free** cread:channel [**private**]. and add in your process !**in**(cread, x:t); **out**(c, x) where c is a public channel. This allows the attacker to obtain any value of type t sent on channel cread, and can be done for several types if desired. As above, when types are ignored, it in fact allows the attacker to obtain any value sent on channel cread.

Memory cell

One can encode a memory cell in which one can read and write. We declare three private channels: one for the cell itself, one for reading and one for writing in the cell.

```
free cell, cread, cwrite: channel [private].
```

and include the following process

```
out(cell, init) |
(!in(cell, x:t); in(cwrite, y:t); out(cell, y)) |
(!in(cell, x:t); out(cread, x); out(cell, x))
```

where t is the type of the content of the cell, and *init* is its initial value. The current value of the cell is the one available as an output on channel cell. We can then write in the cell by outputting on channel cwrite and read from the cell by reading on channel cread.

We can give the attacker the capability to read and/or write the cell by defining cread as a channel on which the attacker can read and/or cwrite as a channel on which the attacker can write, using the asymmetric channels presented above.

It is important for the soundness of this encoding that one never reads on cwrite or writes on cread, except in the code of the cell itself.

Due to the abstractions performed by ProVerif, such a cell is treated in an approximate way: all values written in the cell are considered as a set, and when one reads the cell, ProVerif just guarantees that the obtained value is one of the written values (not necessarily the last one, and not necessarily one written before the read).

Interface for creating principals

Instead of creating two protocol participants A and B, it is also possible to define an interface so that the attacker can create as many protocol participants as he wants with the parameters of its choice, by sending appropriate messages on some channels.

In some sense, the interface provided in the model of Section 5.3 constitutes a limited example of this technique: the attacker can start an initiator that has identity h_I and that talks to responder h_R by sending the message (h_I, h_R) to the first input of processInitiator and it can start a responder that has identity h_R by sending that identity to the first input of processResponder.

A more complex interface can be defined for more complex protocols. Such an interface has been defined for the JFK protocol, for instance. We refer the reader to [ABF07] (in particular Appendix B.3) and to the files in `examples/pitype/jfk` (if you installed by OPAM in the switch ⟨*switch*⟩, in `~/.opam/`⟨*switch*⟩`/doc/proverif/examples/pitype/jfk`) for more information.

6.7.5 Sources of incompleteness

In order to prove protocols, ProVerif translates them internally into Horn clauses. This translation performs safe abstractions that sometimes result in false counterexamples. We detail the main abstractions in this section. We stress that these abstractions preserve soundness: if ProVerif claims that a property is true or false, then this claim is correct. The abstractions only have as a consequence that ProVerif sometimes says that a property "cannot be proved", which is a "don't know" answer.

Repetition of actions. The Horn clauses can be applied any number of times, so the translation ignores the number of repetitions of actions. For instance, ProVerif finds a false attack in the following process, already mentioned in Section 6.2:

```
new k:key; out(c, senc(senc(s,k),k));
in(c, x:bitstring); out(c, sdec(x,k))
```

It thinks that one can decrypt senc(senc(s,k),k) by sending it to the input, so that the process replies with senc(s,k), and then sending this message again to the input, so that the process replies with s. However, this is impossible in reality because the input can be executed only once. The previous process has the same translation into Horn clauses as the process

```
new k:key; out(c, senc(senc(s,k),k));
!in(c, x:bitstring); out(c, sdec(x,k))
```

with an additional replication, and the latter process is subject to the attack outlined above.

This approximation is the main approximation made by ProVerif. In fact, for secrecy (and probably also for basic non-injective correspondences), when all channels are public and the fresh names are generated by **new** as late as possible, this is the only approximation [Bla05]. The option [**precise**], introduced in Section 6.2, allows the user to eliminate many false attacks coming from this approximation.

Position of new. The position of **new** in the process influences the internal representation of fresh names in ProVerif: fresh names created by **new** are represented as functions of the inputs located above that **new**. So the more the **new** are moved downwards in the process, the more arguments they have, and in general the more precise and the more costly the analysis is. (See also Section 6.7.2 for additional discussion of this point.)

Private channels. Private channels are a powerful tool for encoding many features in the pi calculus. However, because of their power and complexity, they also lead to additional approximations in ProVerif. In particular, when c is a private channel, the process P that follows **out**(c, M); P can be executed only when some input listens on channel c; ProVerif does not take that into account and considers that P can always be executed.

Moreover, ProVerif just computes a set of messages sent on a private channel, and considers that any input on that private channel can receive any of these messages (independently of the order in which they are sent). This point can be considered as a particular case of the general approximation that repetitions of actions are ignored: if a message has been sent on a private channel at some point, it may be sent again later. Ignoring the number of repetitions of actions then tends to become more important in the presence of private channels than with public channels only.

Let us consider for instance the process

```
new c:channel; (out(c,M) | in(c,x:t); in(c,y:t); P)
```

The process P cannot be executed, because a single message is sent on channel c, but two inputs must be performed on that channel before being able to execute P. ProVerif cannot take that into account because it ignores the number of repetitions of actions: the process above has the same translation into Horn clauses as the variant with replication

```
new c:channel; ((!out(c,M)) | in(c,x:t); in(c,y:t); P)
```

which can execute P.

Similarly, the process

```
new c:channel; (out(c,s) | in(c,x:t); out(d,c))
```

preserves the secrecy of s because the attacker gets the channel c too late to be able to obtain s. However, ProVerif cannot prove this property because the translation treats it like the following variant

```
new c:channel; ((!out(c,s)) | in(c,x:t); out(d,c))
```

with an additional replication, which does not preserve the secrecy of s.

Observational equivalence. In addition to the previous approximations, ProVerif makes further approximations in order to prove observational equivalence. In order to show that P and Q are observationally equivalent, it proves that, at each step, P and Q reduce in the same way: the same branch of a test or destructor application is taken, communications happen in both processes or in neither of them. This property is sufficient for proving observational equivalence, but it is not necessary. For instance, in a test

```
if M = N then R1 else R2
```

if the **then** branch is taken in P and the **else** branch is taken in Q, then ProVerif cannot prove observational equivalence. However, P and Q may still be observationally equivalent if the attacker cannot distinguish what R_1 does from what R_2 does.

Along similar lines, the biprocess

```
P = out(c,choice[m,n]) | out(c,choice[n,m])
```

satisfies observational equivalence but ProVerif cannot show this: the first component of the parallel composition outputs either m or n, and the attacker has these two names, so ProVerif cannot prove observational equivalence because it thinks that the attacker can distinguish these two situations. In fact, the difference in the first output is compensated by the second output, so that observational equivalence holds. In this simple example, it is easy to prove observational equivalence by rewriting the process into the structurally equivalent process **out**(c,**choice**[m,m]) | **out**(c,**choice**[n,n]) for which ProVerif can obviously prove observational equivalence. It becomes more difficult when a configuration similar to the one above happens in the middle of the execution of the process. Ben Smyth *et al.* are working on an extension of ProVerif to tackle such cases [DRS08].

Limitations of attack reconstruction. Some limitations also come from attack reconstruction. There is no attack reconstruction against nested correspondences. (ProVerif reconstructs attacks only when the basic correspondence at the root of the nested correspondence fails.) The reconstruction of attacks against injective correspondences is based on heuristics that sometimes fail. For observational equivalences, ProVerif can reconstruct a trace that reaches the first point at which the two processes start reducing differently. However, such a trace does not guarantee that observational equivalence is wrong; for this reason, ProVerif never says that an observational equivalence is false.

6.7.6 Misleading syntactic constructs

- In the following ProVerif code

```
if ... then
   let x = ... in
   ...
else
   ...
```

  the **else** branch refers to **let** construct, not to the **if**. The constructs **if**, **let**, and **get** can all have **else** branches, and **else** always refers to the latest one. This is true even if the **else** branch of **let** can never be executed because the **let** always succeeds. Hence, the code above is correctly indented as follows:

```
if ... then
let x = ... in
   ...
else
   ...
```

  and if the **else** branch refers to the **if**, parentheses must be used:

```
if ... then
(
   let x = ... in
   ...
)
else
   ...
```

- When *tc* is a **typeConverter** function and types are ignored, the construct

```
let tc(x) = M in ... else ...
```

  is equivalent to

```
let x = M in ... else ...
```

  Hence, its **else** branch will be executed only if the evaluation of M fails. When M never fails, this is clearly not what was intended.

- In patterns, identifiers without argument are always variables bound by the pattern. For instance, consider

```
const c: bitstring.

let (c, x) = M in ...
```

  Even if c is defined before, c is redefined by the pattern-matching, and the pattern (c, x) matches any pair. ProVerif displays a warning saying that c is rebound. If you want to refer to the constant c in the pattern, please write:

```
const c: bitstring.

let (=c, x) = M in ...
```

The pattern (=c, x) matches pairs whose first component is equal to c. If you want to refer to a data function without argument, the following syntax is also possible:

```
const c: bitstring [data].

let (c(), x) = M in ...
```

- The construct **if** M **then** P **else** Q does not catch failure inside the term M, that is, it executes nothing when the evaluation of M fails. Its **else** branch is executed only when the evaluation of M succeeds and its result is different from true.

  In contrast, the construct **let** $T = M$ **in** P **else** Q catches failure inside T and M. That is, its **else** branch is executed when the evaluation of T or M fails, or when these evaluations succeed and the result of M does not match T.

6.8 Compatibility with CryptoVerif

For a large subset of the ProVerif and CryptoVerif languages, you can run the same input file both in ProVerif and in CryptoVerif. (CryptoVerif is a computationally-sound protocol verifier that can be downloaded from `http://cryptoverif.inria.fr`.) ProVerif proves protocols in the formal model and can reconstruct attacks, while CryptoVerif proves protocols in the computational model. CryptoVerif proofs are more satisfactory, because they rely on a less abstract model, but CryptoVerif is more difficult to use and less widely applicable than ProVerif, and it cannot reconstruct attacks, so these two tools are complementary.

ProVerif includes the following extensions to allow that. ProVerif allows to use macros for defining the security assumptions on primitives. One can define a macro $name(i_1, \ldots, i_n)$ by

def $name(i_1, \ldots, i_n)$ {
  declarations
}

Then **expand** $name(a_1, \ldots, a_n)$. expands to the declarations inside **def** with $a_1, \ldots a_n$ substituted for $i_1, \ldots, i_n$. As an example, we can define block ciphers by

```
def SPRP_cipher(keyseed, key, blocksize, kgen, enc, dec, Penc) {

fun enc(blocksize, key): blocksize.
fun kgen(keyseed):key.
fun dec(blocksize, key): blocksize.

equation forall m:blocksize, r:keyseed;
  dec(enc(m, kgen(r)), kgen(r)) = m.
equation forall m:blocksize, r:keyseed;
  enc(dec(m, kgen(r)), kgen(r)) = m.

}
```

SPRP stands for Super Pseudo-Random Permutation, a standard computational assumption on block ciphers; here, the ProVerif model tries to be close to this assumption, even if it is probably stronger. Penc is the probability of breaking this assumption; it makes sense only for CryptoVerif, but the goal to use the same macros with different definitions in ProVerif and in CryptoVerif.

We can then declare a block cipher by

```
expand SPRP_cipher(keyseed, key, blocksize, kgen, enc, dec, Penc).
```

without repeating the whole definition.

The definitions of macros are typically stored in a library. Such a library can be specified by the command-line option `-lib`. The file `cryptoverif.pvl` (at the root of the ProVerif distribution or, if you installed by OPAM in the switch ⟨*switch*⟩, in `~/.opam/`⟨*switch*⟩`/share/proverif/cryptoverif.pvl`) is an example of such a library. It can be included by calling

```
proverif -lib cryptoverif ⟨filename⟩.pv
```

The definitions of cryptographic primitives need to be included in such a library, because they are typically very different between ProVerif and CryptoVerif. We can then use a different library for each tool.

ProVerif also supports but ignores the CryptoVerif declarations **param**, **proba**, **proof**, **implementation**, as well as the implementation annotations. It supports options after a type declaration, as in **type** t [*option*]. These options are ignored. It supports **channel** $c_1, \ldots, c_n$. as a synonym of **free** $c_1, \ldots, c_n$: channel. (Only the former is supported by CryptoVerif.) It supports **yield** as a synonym of 0. It allows ! i <= n instead of just !. (CryptoVerif uses the former.) It also allows the following constructs:

New syntax	Original equivalent syntax
foreach i <= N **do** P	! i <= n P
x <−R T; P	**new** x:T; P
x[:T] <− M; P	**let** x[:T] = M **in** P
x <−R T; N	**new** x:T; N
x[:T] <− M; N	**let** x[:T] = M **in** N

ProVerif accepts

equation [**forall** $x_1 : T_1, \ldots, x_n : T_n$;] M .

as a synonym for

equation [**forall** $x_1 : T_1, \ldots, x_n : T_n$;] M = true .

ProVerif also allows disequations

equation [**forall** $x_1 : T_1, \ldots, x_n : T_n$;] M_1 <> M_2 .

It just checks that they are valid (the terms M_1 and M_2 do not unify modulo the equational theory) and otherwise ignores them. Examples of protocols written with CryptoVerif compatibility in mind can be found in subdirectory `examples/cryptoverif/`. You can run them by

```
./proverif -lib cryptoverif examples/cryptoverif/⟨filename⟩.pcv
```

from the main directory of the ProVerif distribution. If you installed ProVerif by OPAM in the switch ⟨*switch*⟩, these files are in `~/.opam/`⟨*switch*⟩`/doc/proverif/examples/cryptoverif` and you can run them by

```
proverif -lib ~/.opam/⟨switch⟩/share/proverif/cryptoverif ⟨filename⟩.pcv
```

from the directory `~/.opam/`⟨*switch*⟩`/doc/proverif/examples/cryptoverif/`.

There are still features supported only by one of the two tools. CryptoVerif does not support the definition of free names. You can define free public channels by **channel** $c_1, \ldots, c_n$, and you can define public constants by **const** c:T. CryptoVerif does not support private constants or functions. All constants and functions are public. CryptoVerif does not support private channels (**free** c: channel [**private**]. and channels bound by **new**). All channels must be public and free. It is recommended to use a distinct channel for each input and output in CryptoVerif, because when there are several possible input on the same channel, the one that receives the message is chosen at random. CryptoVerif does not support phases (**phase**) nor synchronization (**sync**).

CryptoVerif does not support destructors. It supports equations, though they have a different meaning: in ProVerif, you need to give all equations that hold, all other terms are considered as different; in CryptoVerif, you give some equations that hold, there may be other true equalities. CryptoVerif does not support **fail**. All terms always succeed in CryptoVerif. Conversely, ProVerif does not support

equationbuiltin: it does not support associativity; you can define commutativity by giving the explicit equation. ProVerif does not support defining primitives with indistinguishability axioms as it is done in CryptoVerif and it does not support **collision** statements. You can use different definitions of primitives in ProVerif and CryptoVerif by using a different library or using **ifdef**. You should still make sure that all functions always succeed, possibly returning a special value instead of failing.

CryptoVerif supports the correspondence, secrecy, and equivalence queries. However, it does not support queries using the predicates attacker, mess, or table. CryptoVerif does not support nested correspondences, nor **noninterf**, **weaksecret**, **choice**. CryptoVerif does not support predicates and **let** ... **suchthat**. It does not support refering to bound names in queries. It does not support **putbegin**, secrecy assumptions (**not** F.), **nounif**, **axiom**, nor **lemma**. You can put queries and proof indications specific to ProVerif inside **ifdef**s.

The processes in CryptoVerif must alternate inputs and outputs: an input must be followed by computations and an output; an output must be followed by replications, parallel compositions, and inputs. The main process must start with an input, a replication or a parallel composition. This constraint allows the adversary to schedule the execution of the processes by sending a message to the appropriate input. You can always add inputs or outputs of empty messages to satisfy this constraint.

ProVerif does not support the **find** construct of CryptoVerif. **find** can be encoded using tables with **insert** and **get** instead.

The `emacs` mode included in the CryptoVerif distribution includes a mode for `.pcv` files, which is designed for compatibility with both tools. Common keywords and builtin identifiers are displayed normally, while keywords and builtin identifiers supported by only one of the two tools are highlighted in red. The recommended usage is to use

ifdef(‘ProVerif’,‘*ProVerif specific code*’)
ifdef(‘CryptoVerif’,‘*CryptoVerif specific code*’)

inside your `.pcv` files. When ProVerif is called with a `.pcv` file as argument, it automatically preprocesses it with m4, as if you ran

`m4 -DProVerif` ⟨*filename*⟩`.pcv > `⟨*filename*⟩`.pv`

before analyzing it. Similarly, when CryptoVerif is called with a `.pcv` file as argument, it automatically preprocesses it with m4, as if you ran

`m4 -DCryptoVerif` ⟨*filename*⟩`.pcv > `⟨*filename*⟩`.cv`

6.9 Additional programs

6.9.1 `test`

Usage:

`test` ⟨mode⟩ ⟨test_set⟩

where ⟨mode⟩ can be:

- `test`: test the mentioned scripts
- `test_add`: test the mentioned scripts and add the expected result in the script when it is missing
- `add`: add the expected result in the script when it is missing, do not test scripts that already have an expected result
- `update`: test the mentioned scripts and update the expected result in the script

and ⟨test_set⟩ can be:

- `typed` runs ProVerif on examples for the typed front-ends
- `untyped` runs ProVerif on examples for the untyped front-ends

- `arinc` runs ProVerif on the ARINC823 protocol
- `dir` ⟨prefix⟩ ⟨list_of_directories⟩ analyzes the mentioned directories using ProVerif, using ⟨prefix⟩ as prefix for the output files.

⟨test_set⟩ can be omitted when it is `typed`, and ⟨mode⟩ ⟨test_set⟩ can both be omitted when they are `test typed`.

The script `test` is a bash shell script, so you must have bash installed. On Windows, the best is to install Cygwin and run `test` from a Cygwin terminal.

The script `test` must be run in the CryptoVerif main directory; the programs `analyze` and `proverif` must be present in that directory.

The script `test` first runs the script `prepare` in each directory when it is present. That allows for instance to generate the ProVerif scripts to run. Then it runs the program `analyze` described below.

6.9.2 analyze

The program `analyze` is mainly meant to be called from `test`, but it can also be called directly.

Usage:

`analyze` ⟨prog⟩ ⟨mode⟩ ⟨tmp_directory⟩ ⟨prefix_for_output_files⟩ `dirs` ⟨directories⟩

`analyze` ⟨prog⟩ ⟨mode⟩ ⟨tmp_directory⟩ ⟨prefix_for_output_files⟩ `file` ⟨directory⟩ ⟨filename⟩

where ⟨prog⟩ is either `CV` for CryptoVerif or `PV` for ProVerif and ⟨mode⟩ is as for the `test` program above.

This program analyzes a series of scripts using the program specified by ⟨prog⟩.

- In the first command line, it analyzes scripts in the mentioned directories and in their subdirectories. The files whose name contains `.m4.` or `.out.` are excluded. (The first ones are supposed to be files to preprocess by `m4` before actually analyzing them; the second ones are supposed to be output files.) When the program is CryptoVerif, the files whose name ends with `.cv`, `.ocv`, or `.pcv` are analyzed. When the program is ProVerif, the files whose name ends with `.pcv`, `.pv`, `.pi`, `.horn`, or `.horntype` are analyzed.
- In the second command line, the specified file in the specified directory is analyzed, provided it has one of the extensions above. (The directory and the file are mentioned separately because the directory may be used to locate the library `mylib.*`, see below.)

The executable for CryptoVerif is searched in the current directory, in `$HOME/CryptoVerif`, and in the `PATH`. The executable for ProVerif is searched in the current directory, in `$HOME/proverif/proverif`, and in the `PATH`.

When `mylib.cvl` is present in a directory, its files with extension `.cv` or `.pcv` are analyzed using that library of primitives for CryptoVerif. Otherwise, the default library is used.

When `mylib.ocvl` is present in a directory, its files with extension `.ocv` are analyzed using that library of primitives for CryptoVerif. Otherwise, the default library is used.

When `mylib.pvl` is present in a directory, its files with extension `.pcv` or `.pv` are analyzed using that library of primitives for ProVerif. Otherwise, the library `cryptoverif.pvl` is used for `.pcv` files and no library for `.pv` files. The file `cryptoverif.pvl` is searched in the current directory, `$HOME/CryptoVerif` and `$HOME/proverif/proverif`. If it is not found and `mylib.pvl` is not present in the directory, .pcv files are not analyzed using ProVerif.

The result of running each script is compared to the expected result. The expected result is found in the script itself in a comment that starts with `EXPECTED` for CryptoVerif and `EXPECTPV` for ProVerif, and ends with `END`. (The entire lines that contain `EXPECTED`, resp. `EXPECTPV` and `END` do not belong to the expected result.) For CryptoVerif, the expected result consists of the line `RESULT Could not prove ...` or `All queries proved` in the output of CryptoVerif. For ProVerif, it consists of the lines that start with `RESULT` in the output of ProVerif. It also includes a runtime of the script or an error message `xtime: ...` if the execution terminates with an error.

In the modes `update` (resp. `test_add` or `add`), the expected result is updated (resp. added if it is absent or empty). To deal with generated files, the `EXPECTED`, resp. `EXPECTPV` line may contain the indications

`FILENAME:` *name of the file* `TAG:` *distinct tag*

In this case, the expected result is not updated in the script itself, but in the file whose name is mentioned after `FILENAME:`, and inside this file after an exact copy of the line that contains `EXPECTED`, resp. `EXPECTPV`. (This line is unique thanks to the tag.) The idea is that this file is the file from which the script was generated. Hence regenerating the script from this file with an updated expected result will update the expected result in the script.

6.9.3 `addexpectedtags`

Usage:

`addexpectedtags` ⟨directories⟩

For each mentioned directory, for each file in that directory or its subdirectories that contains `.m4.` in its name and ends with `.cv`, `.ocv`, `.pcv`, `.pv`, `.pi`, `.horntype`, `.horn`, this program adds at the end of each line that contains `EXPECTED` or `EXPECTPV` the indications

`FILENAME:` *name of the file* `TAG:` *distinct integer*

These files are supposed to be initial models used to generate CryptoVerif or ProVerif scripts by the `m4` preprocessor. The additional indications will propagate to the generated scripts, and will allow the `analyze` program above to find from which `m4` file the script was generated (indicated after `FILENAME:`) and inside this `m4` file, which expected result indication ended up in the considered script (identified by the integer after `TAG:`). It can then update the expected results in the mode `update`, `add`, or `test_add` (the last two when the expected result was initially empty).

Chapter 7

Outlook

The ProVerif software tool is the result of more than a decade of theoretical research. This manual explained how to use ProVerif in practice. More information on the theory behind ProVerif can be found in research papers:

- For a general survey, please see [Bla16].
- For the verification of secrecy as reachability, we recommend [Bla11, AB05a].
- For the verification of correspondences, we recommend [Bla09].
- For the verification of strong secrecy, see [Bla04]; for observational equivalence, guessing attacks, and the treatment of equations, see [BAF08]. See [BS18] for the extension to synchronization.
- For the reconstruction of attacks, see [AB05c].
- For the termination result on tagged protocols, see [BP05].
- Case studies can be found in [AB05b, ABF07, BC08, KBB17, BBK17, Bla17].

ProVerif is a powerful tool for verifying protocols in formal model. It works for an unbounded number of sessions and an unbounded message space. It supports many cryptographic primitives defined by rewrite rules or equations. It can prove various security properties: reachability, correspondences, and observational equivalences. These properties are particularly interesting to the security domain because they allow analysis of secrecy, authentication, and privacy properties. It can also reconstruct attacks when the desired properties do not hold.

However, ProVerif performs abstractions, so there are situations in which the property holds and cannot be proved by ProVerif. Moreover, proofs of security properties in ProVerif abstract away from details of the cryptography, and therefore may not in general be sound with respect to the computational model of cryptography. The CryptoVerif tool (`http://cryptoverif.inria.fr`), an automatic prover for security properties in the computational security model, aims to address this problem.

Appendix A

Language reference

In this appendix, we provide a reference for the typed pi calculus input language of ProVerif. We adopt the following conventions. X^* means any number of repetitions of X; and $[X]$ means X or nothing. seq⟨X⟩ is a sequence of X, that is, seq⟨X⟩ = [(⟨X⟩`,`)*⟨X⟩] = ⟨X⟩`,` ... `,`⟨X⟩. (The sequence can be empty, it can be one element, or it can be several elements separated by commas.) seq$^+$⟨X⟩ is a non-empty sequence of X: seq$^+$⟨X⟩ = (⟨X⟩`,`)*⟨X⟩ = ⟨X⟩`,` ... `,`⟨X⟩. (It can be one or several elements of ⟨X⟩ separated by commas.) Text in typewriter style should appear as it is in the input file. Text between ⟨ and ⟩ represents non-terminals of the grammar. In particular, we will use:

- ⟨ident⟩ to denote identifiers (Section 3.1.4) 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.
- ⟨nat⟩ to range over natural numbers.
- ⟨typeid⟩ to denote types (Section 3.1.1), which can be identifiers ⟨ident⟩ or the reserved word `channel`.
- ⟨options⟩ ::= [`[`seq$^+$⟨ident⟩`]`], where the allowed identifiers in the sequence are `data`, `private`, and `typeConverter` for the `fun` and `const` declarations; `private` for the `reduc` and `free` declarations; `memberOptim` and `block` for the `pred` declaration; `precise` for processes (Figure A.7); `noneSat`, `discardSat`, `instantiateSat`, `fullSat`, `noneVerif`, `discardVerif`, `instantiateVerif` and `fullVerif` for the query, lemma and axiom declarations; `induction` and `noInduction` for the lemma and query declarations; `maxSubset` for the lemma declaration; `proveAll` for the query declaration; `reachability`, `pv_reachability`, `real_or_random`, `pv_real_or_random`, and all options starting with `cv_` for the `secret` query.
- ⟨infix⟩ ::= `||` | `&&` | `=` | `<>` | `<=` | `>=` | `<` | `>` to denote some infix symbols on terms.

The input file consists of a list of declarations, followed by the keyword `process` and a process:

⟨decl⟩* `process` ⟨process⟩

or a list of declarations followed by an equivalence query between two processes (see end of Section 4.3.2):

⟨decl⟩* `equivalence` ⟨process⟩ ⟨process⟩

Libraries (loaded with the command-line option `-lib`) are lists of declarations ⟨decl⟩*.

We start by presenting the grammar for terms in Figure A.1. The grammar for declarations is considered in Figure A.2. Finally, Figure A.7 covers the grammar for processes.

Figure A.1 Grammar for terms (see Sections 3.1.4, 4.1.3, and 4.1.4)

```
⟨term⟩ ::= ⟨ident⟩
         | ⟨nat⟩
         | (seq⟨term⟩)
         | ⟨ident⟩(seq⟨term⟩)
         | ⟨term⟩ (+ | -) ⟨nat⟩
         | ⟨nat⟩ + ⟨term⟩
         | ⟨term⟩ ⟨infix⟩ ⟨term⟩
         | not (⟨term⟩)
⟨pterm⟩ ::= ⟨ident⟩
         | (seq⟨pterm⟩)
         | ⟨ident⟩(seq⟨pterm⟩)
         | choice[⟨pterm⟩,⟨pterm⟩]                                  (see Section 4.3.2)
         | ⟨pterm⟩ (+ | -) ⟨nat⟩
         | ⟨nat⟩ + ⟨pterm⟩
         | ⟨pterm⟩ ⟨infix⟩ ⟨pterm⟩
         | not (⟨pterm⟩)
         | new ⟨ident⟩[[seq⟨ident⟩]]: ⟨typeid⟩; ⟨pterm⟩
         | ⟨ident⟩ <-R ⟨typeid⟩; ⟨pterm⟩                             (see Section 6.8)
         | if ⟨pterm⟩ then ⟨pterm⟩ [else ⟨pterm⟩]
         | let ⟨pattern⟩ = ⟨pterm⟩ in ⟨pterm⟩ [else ⟨pterm⟩]
         | ⟨ident⟩[:⟨typeid⟩] <- ⟨pterm⟩;⟨pterm⟩                     (see Section 6.8)
         | let ⟨typedecl⟩ suchthat ⟨pterm⟩ in ⟨pterm⟩ [else ⟨pterm⟩]  (see Section 6.3)
         | insert ⟨ident⟩(seq⟨pterm⟩); ⟨pterm⟩                       (see Section 4.1.5)
         | get ⟨ident⟩(seq⟨pattern⟩) [suchthat ⟨pterm⟩] ⟨options⟩ in ⟨pterm⟩ [else ⟨pterm⟩]
                                                                     (see Section 4.1.5)
         | event ⟨ident⟩[(seq⟨pterm⟩)]; ⟨pterm⟩                      (see Section 3.2.2)
⟨pattern⟩ ::= ⟨ident⟩[: ⟨typeid⟩]
         | ⟨nat⟩
         | ⟨pattern⟩ + ⟨nat⟩
         | ⟨nat⟩ + ⟨pattern⟩
         | (seq⟨pattern⟩)
         | ⟨ident⟩(seq⟨pattern⟩)
         | =⟨pterm⟩
⟨mayfailterm⟩ ::= ⟨term⟩
         | fail
⟨typedecl⟩ ::= seq+⟨ident⟩: ⟨typeid⟩[,⟨typedecl⟩]
⟨failtypedecl⟩ ::= seq+⟨ident⟩: ⟨typeid⟩[or fail][,⟨failtypedecl⟩]
```

The precedences of infix symbols, from low to high, are: `||`, `&&`, `=`, `<>`, `<=`, `>=`, `<`, `>`, and `+`, `-`, which both have the same precedence and associate to the left as usual. The grammar of terms ⟨term⟩ is further restricted after parsing. In `reduc` and `equation` declarations, the only allowed function symbols are constructors, so `||`, `&&`, `=`, `<>`, `<=`, `>=`, `<`, `>`, `-`, `not` are not allowed, and names are not allowed as identifiers. In `noninterf` declarations, the only allowed function symbols are constructors and names are allowed as identifiers. In `elimtrue` declarations, the term can only be a fact of the form $p(M_1, \ldots, M_k)$ for some predicate p; names are not allowed as identifiers. In clauses (Figure A.6), the hypothesis of clauses can be conjunctions of facts of the form $p(M_1, \ldots, M_k)$ for some predicate p, equalities, disequalities, or inequalities; the conclusion of clauses can only be a fact of the form $p(M_1, \ldots, M_k)$ for some predicate p; names are not allowed as identifiers.

Figure A.2 Grammar for declarations

```
⟨decl⟩ ::= type ⟨ident⟩ ⟨options⟩.                                   (see Section 3.1.1)
        |  channel seq+⟨ident⟩.                                      (see Section 6.8)
        |  free seq+⟨ident⟩: ⟨typeid⟩ ⟨options⟩.                     (see Section 3.1.1)
        |  const seq+⟨ident⟩: ⟨typeid⟩ ⟨options⟩.                    (see Section 4.1.1)
        |  fun ⟨ident⟩(seq⟨typeid⟩): ⟨typeid⟩ ⟨options⟩.             (see Section 3.1.1)
        |  letfun ⟨ident⟩[([⟨typedecl⟩])] = ⟨pterm⟩.                 (see Section 4.2.3)
        |  reduc ⟨reduc⟩ ⟨options⟩.                                  (see Section 3.1.1)
           where ⟨reduc⟩ ::= [forall ⟨typedecl⟩;] ⟨term⟩ = ⟨term⟩ [; ⟨reduc⟩]
        |  fun ⟨ident⟩(seq⟨typeid⟩): ⟨typeid⟩ reduc ⟨reduc'⟩ options.   (see Section 4.2.1)
           where ⟨reduc'⟩ ::= [forall ⟨failtypedecl⟩;]⟨ident⟩(seq⟨mayfailterm⟩) = ⟨mayfailterm⟩
                              [otherwise ⟨reduc'⟩]
        |  equation ⟨eqlist⟩ ⟨options⟩.                             (see Section 4.2.2)
           where ⟨eqlist⟩ ::= [forall ⟨typedecl⟩ ;] ⟨term⟩ = ⟨term⟩ [;⟨eqlist⟩]
        |  pred ⟨ident⟩[(seq⟨typeid⟩)] ⟨options⟩.                    (see Section 6.3)
        |  table ⟨ident⟩(seq⟨typeid⟩).                              (see Section 4.1.5)
        |  let ⟨ident⟩[([⟨typedecl⟩])] = ⟨process⟩.                  (see Section 3.1.3)
           where ⟨process⟩ is specified in Figure A.7.
        |  set ⟨name⟩ = ⟨value⟩.                                    (see Section 6.6.2)
           where the possible values of ⟨name⟩ and ⟨value⟩ are listed in Section 6.6.2.
        |  event ⟨ident⟩[(seq⟨typeid⟩)].                            (see Section 3.2.2)
        |  query [⟨typedecl⟩;] ⟨query⟩ ⟨options⟩.                    (see Sections 3.2, 4.3.1)
           where ⟨query⟩ is defined in Figure A.3.
        |  (axiom | restriction | lemma) [⟨typedecl⟩;] ⟨lemma⟩ ⟨options⟩.   (see Section 6.2)
           where ⟨lemma⟩ is defined in Figure A.3.
        |  noninterf [⟨typedecl⟩;] seq⟨nidecl⟩.                     (see Section 4.3.2)
           where ⟨nidecl⟩ ::= ⟨ident⟩ [among (seq+⟨term⟩)]
        |  weaksecret ⟨ident⟩.                                      (see Section 4.3.2)
        |  not [⟨typedecl⟩;] ⟨gterm⟩.                               (see Section 6.7.2)
           where ⟨gterm⟩ is defined in Figure A.3.
        |  nounif [⟨typedecl⟩;] ⟨nounifdecl⟩ [[seq+⟨nounifoption⟩]].   (see Section 6.7.2)
           where ⟨nounifdecl⟩ and ⟨nounifoption⟩ are defined in Figure A.5.
        |  elimtrue [⟨failtypedecl⟩;] ⟨term⟩.                       (see Section 6.3)
        |  clauses ⟨clauses⟩.                                       (see Section 6.3)
           where ⟨clauses⟩ is defined in Figure A.6.
        |  param seq+⟨ident⟩ ⟨options⟩.                             (see Section 6.8)
        |  proba ⟨ident⟩.                                           (see Section 6.8)
        |  proof {⟨proof⟩}                                          (see Section 6.8)
        |  def ⟨ident⟩(seq⟨typeid⟩) {⟨decl⟩*}                       (see Section 6.8)
        |  expand ⟨ident⟩(seq⟨typeid⟩).                             (see Section 6.8)
```

Figure A.3 Grammar for `not` (see Section 6.7.2), queries (see Sections 3.2 and 4.3.1), and lemmas (see Section 6.2)

```
⟨query⟩ ::= ⟨gterm⟩ [public_vars seq+⟨ident⟩] [; ⟨query⟩]
          | secret ⟨ident⟩ [public_vars seq+⟨ident⟩] ⟨options⟩ [; ⟨query⟩]
          | putbegin event:seq+⟨ident⟩ [; ⟨query⟩]                    (see Section 6.5)
          | putbegin inj-event:seq+⟨ident⟩ [; ⟨query⟩]                (see Section 6.5)
⟨lemma⟩ ::= ⟨gterm⟩ [; ⟨lemma⟩]
          | ⟨gterm⟩ for { public_vars seq+⟨ident⟩ } [; ⟨lemma⟩]
          | ⟨gterm⟩ for { secret ⟨ident⟩ [public_vars seq+⟨ident⟩] [real_or_random] } [; ⟨lemma⟩]
⟨gterm⟩ ::= ⟨ident⟩
          | ⟨ident⟩(seq⟨gterm⟩) [phase ⟨nat⟩] [@⟨ident⟩]
          | choice[⟨gterm⟩,⟨gterm⟩]
          | ⟨gterm⟩ ⟨infix⟩ ⟨gterm⟩
          | ⟨gterm⟩ (+ | -) ⟨nat⟩
          | ⟨nat⟩ + ⟨gterm⟩
          | event(seq⟨gterm⟩) [@⟨ident⟩]
          | inj-event(seq⟨gterm⟩) [@⟨ident⟩]
          | ⟨gterm⟩ ==> ⟨gterm⟩
          | (seq⟨gterm⟩)
          | new ⟨ident⟩[[⟨gbinding⟩]]                                 (see Section 6.4)
          | let ⟨ident⟩ = ⟨gterm⟩ in ⟨gterm⟩                          (see Section 6.4)
⟨gbinding⟩ ::= !⟨nat⟩ = ⟨gterm⟩ [; ⟨gbinding⟩]
          | ⟨ident⟩ = ⟨gterm⟩ [; ⟨gbinding⟩]
```

The precedences of infix symbols, from low to high, are: `==>`, `||`, `&&`, `=`, `<>`,`<=`, `>=`, `<`, `>`, and `+`, `-`, which both have the same precedence and associate to the left as usual. The grammar above is useful to know exactly how terms are parsed and where parentheses are needed. However, it is further restricted after parsing, so that the grammar of ⟨gterm⟩ in queries and lemmas is in fact the one of q in Figure A.4 and the grammar of ⟨gterm⟩ in `not` declarations is the one of F in Figure A.4 excluding events, equalities, disequalities, and inequalities.

Figure A.4 Grammar for `not`, queries, and lemmas restricted after parsing

q ::=	query
F_1 && ... && F_n	reachability
F_1 && ... && F_n ==> H	correspondence
let $x = A$ **in** q	let binding, see Section 6.4
H ::=	hypothesis
F	fact
M=N	equality
M<>N	disequality
M>N	greater
M<N	smaller
M>=N	greater or equal
M<=N	smaller or equal
is_nat (M)	M is a natural number
H && H	conjunction
H || H	disjunction
false	constant false
F ==> H	nested correspondence
let $x = A$ **in** H	let binding, see Section 6.4
F ::=	fact
AF	action fact
AF@x	action fact executed at time x
$p(M_1, \ldots, M_n)$	user-defined predicate, see Section 6.3
let $x = A$ **in** F	let binding, see Section 6.4
AF ::=	action fact
attacker(A)	the adversary has A (in any phase)
attacker(A) **phase** n	the adversary has A in phase n
mess(B, A)	A is sent on channel B (in the last phase)
mess(B, A) **phase** n	A is sent on channel B in phase n
event($e(A_1, \ldots, A_n)$)	non-injective event
inj−event($e(A_1, \ldots, A_n)$)	injective event
M, N ::=	term
x, a, c	variable, free name, or constant
$0, 1, \ldots$	natural numbers
$f(M_1, \ldots, M_n)$	constructor application
$(M_1, \ldots, M_n)$	tuple
$M + i$	addition, $i \in \mathbb{N}$
$i + M$	addition, $i \in \mathbb{N}$
$M - i$	subtraction, $i \in \mathbb{N}$
new $a[g_1 = M_1, \ldots, g_k = M_k]$	bound name (g ::= $!n \mid x$), see Section 6.4
let $x = M$ **in** N	let binding, see Section 6.4
A, B ::=	biterm
...	same cases as terms
choice[A,B]	choice

Figure A.5 Grammar for `nounif` (see Section 6.7.2)

```
⟨nounifdecl⟩ ::= let ⟨ident⟩ = ⟨gformat⟩ in ⟨nounifdecl⟩
               |  ⟨ident⟩[(seq⟨gformat⟩) [phase ⟨nat⟩]] [/⟨nat⟩]
⟨gformat⟩ ::= ⟨ident⟩
            |  *⟨ident⟩
            |  ⟨ident⟩(seq⟨gformat⟩)
            |  choice[⟨gformat⟩,⟨gformat⟩]
            |  not(seq⟨gformat⟩)
            |  (seq⟨gformat⟩)
            |  new ⟨ident⟩[[[⟨fbinding⟩]]]
            |  let ⟨ident⟩ = ⟨gformat⟩ in ⟨gformat⟩
⟨fbinding⟩ ::= !⟨nat⟩ = ⟨gformat⟩ [; ⟨fbinding⟩]
             |  ⟨ident⟩ = ⟨gformat⟩ [; ⟨fbinding⟩]
⟨nounifoption⟩ ::= hypothesis
                 |  conclusion
                 |  ignoreAFewTimes
                 |  inductionOn = ⟨ident⟩
                 |  inductionOn = {seq+⟨ident⟩}
```

Figure A.6 Grammar for `clauses` (see Section 6.3)

```
⟨clauses⟩ ::= [forall ⟨failtypedecl⟩;] ⟨clause⟩ [; ⟨clauses⟩]
⟨clause⟩ ::= ⟨term⟩
           |  ⟨term⟩ -> ⟨term⟩
           |  ⟨term⟩ <-> ⟨term⟩
           |  ⟨term⟩ <=> ⟨term⟩
```

Figure A.7 Grammar for processes (see Section 3.1.4)

```
⟨process⟩ ::= 0
            | yield                                                    (see Section 6.8)
            | ⟨ident⟩[(seq⟨pterm⟩)]
            | (⟨process⟩)
            | ⟨process⟩ | ⟨process⟩
            | !⟨process⟩
            | ! ⟨ident⟩ <= ⟨ident⟩ ⟨process⟩                            (see Section 6.8)
            | foreach ⟨ident⟩ <= ⟨ident⟩ do ⟨process⟩                   (see Section 6.8)
            | new ⟨ident⟩[[seq⟨ident⟩]]: ⟨typeid⟩ [; ⟨process⟩]
            | ⟨ident⟩ <-R ⟨typeid⟩ [; ⟨process⟩]                        (see Section 6.8)
            | if ⟨pterm⟩ then ⟨process⟩ [else ⟨process⟩]
            | in(⟨pterm⟩,⟨pattern⟩) ⟨options⟩ [; ⟨process⟩]
            | out(⟨pterm⟩,⟨pterm⟩)) [; ⟨process⟩]
            | let ⟨pattern⟩ = ⟨pterm⟩ [in ⟨process⟩ [else ⟨process⟩]]
            | ⟨ident⟩[:⟨typeid⟩] <- ⟨pterm⟩ [;⟨process⟩]                (see Section 6.8)
            | let ⟨typedecl⟩ suchthat ⟨pterm⟩ ⟨options⟩ [in ⟨process⟩ [else ⟨process⟩]]
                                                                       (see Section 6.3)
            | insert ⟨ident⟩(seq⟨pterm⟩) [; ⟨process⟩]                 (see Section 4.1.5)
            | get ⟨ident⟩(seq⟨pattern⟩) [suchthat ⟨pterm⟩] ⟨options⟩ [in ⟨process⟩ [else ⟨process⟩]]
                                                                       (see Section 4.1.5)
            | event ⟨ident⟩[(seq⟨pterm⟩)] [; ⟨process⟩]                (see Section 3.2.2)
            | phase ⟨nat⟩ [; ⟨process⟩]                                (see Section 4.1.6)
            | sync ⟨nat⟩ [[⟨tag⟩]] [; ⟨process⟩]                       (see Section 4.1.7)
```

Appendix B

Semantics

In this appendix, we provide the semantics for the input language of ProVerif. A simpler semantics for the core language of ProVerif can be found in research papers, for instance [Bla16].

We consider an infinite set of names $\mathcal{N}$. We denote by $\mathcal{N}_{\text{pub}}$ all names declared with the `channel` and `free` (without option private) declarations from Figure A.2. We denote by $\mathcal{F}_c$ the set containing the constructor function symbols declared with the `fun` declaration without `reduc` from Figure A.2 as well as built-in constructors: `true`, `false`, `0`, `+1`, tuples. Similarly, we denote by $\mathcal{F}_d$ the set containing destructor function symbols declared by the `reduc` declaration or the `fun` declaration with `reduc` as well as built-in destructors: `not`, `is_nat`, `||`, `&&`, `=`, `<>`, `<=`, `>=`, `<`, `>`, `-`k for each k natural number. (More built-in constructors and destructors are used by ProVerif for some encodings; we omit them here.)

We denote by $M, N, \ldots$ a ⟨term⟩ that does not contain destructor function symbols. The equational theory is denoted $\mathcal{E}$ and is the set of all equations defined with the `equation` declaration from Figure A.2. We will denote by $M =_{\mathcal{E}} N$ when M and N are equal modulo the equational theory $\mathcal{E}$.

Note that in Figure A.1, ⟨term⟩ includes the syntax for natural numbers. Internally, they are represented by a constant `0` and a unary constructor function symbol `+1` such that a natural number n is in fact n successive applications of `+1` to `0`. For example, 2 is the term `+1(+1(0))`. The term $M + k$ and $k + M$ are therefore syntactic sugar for k applications of `+1` to the term M. A similar comment applies to process terms ⟨pterm⟩ and patterns ⟨pattern⟩.

Rewrite rules We denote by $U, V, \ldots$ a ⟨mayfailterm⟩ that does not contain destructor function symbols. A conditional rewrite rule is a classic rewrite rule $h(U_1, \ldots, U_n) \to U$ to which a conditional formula ϕ is added, denoted $h(U_1, \ldots, U_n) \to U \mid\mid \phi$. Note that the formula is always of the form $\bigwedge_{i=1}^{n} M_i \geq N_i \wedge \bigwedge_{i'=1}^{n'} \neg nat(M'_i) \wedge \bigwedge_{i''=1}^{n''} nat(M''_i) \wedge \bigwedge_{i'''=1}^{n'''} \forall \tilde{x}_i . M'''_i \neq N'''_i$ where $\tilde{x}$ stands for a sequence of variables and *nat* is the predicate returning true iff its argument is a natural number. From the rewrite rules ⟨reduc⟩ and ⟨reduc'⟩ in the declaration of a destructor from Figure A.2, ProVerif generates a set of equivalent conditional rewrite rules.

For example, assume that $\texttt{enc}/2 \in \mathcal{F}_c$ and consider the destructor `dec` declared by

```
reduc forall x:bitstring, y:bitstring; dec(enc(x,y),y) = x.
```

ProVerif generates the following set of conditional rewrite rules:

$$\begin{array}{l}
\texttt{dec}(\texttt{enc}(x, y), y) \to x \\
\texttt{dec}(x, y) \to \texttt{fail} \mid\mid \forall z.x \neq \texttt{enc}(z, y) \\
\texttt{dec}(\texttt{fail}, u) \to \texttt{fail} \\
\texttt{dec}(x, \texttt{fail}) \to \texttt{fail}
\end{array}$$

where x, y are variables and u is a may-fail variable.

The infix symbols `&&`, `||`, `=`,... defined in ⟨infix⟩ are in fact represented internally by destructor function symbols. For instance,

$$\begin{array}{l}
\texttt{\&\&}(\texttt{true}, u) \to u \\
\texttt{\&\&}(x, u) \to \texttt{false} \mid\mid x \neq \texttt{true} \\
\texttt{\&\&}(\texttt{fail}, u) \to \texttt{fail}
\end{array}$$

where x is a variable and u is a may-fail variable.

Similarly, the infix operators on natural numbers `>=`, `>`, ... are also represented by destructor function symbols:

$$\begin{aligned}
&\texttt{>=}(x,y) \to \texttt{true} \mid\mid x \geq y\\
&\texttt{>=}(x,y) \to \texttt{false} \mid\mid y \geq x+1\\
&\texttt{>=}(x,y) \to \texttt{fail} \mid\mid \neg nat(x)\\
&\texttt{>=}(x,y) \to \texttt{fail} \mid\mid \neg nat(y)\\
&\texttt{>=}(\texttt{fail},u) \to \texttt{fail}\\
&\texttt{>=}(x,\texttt{fail}) \to \texttt{fail}
\end{aligned}$$

The symbols `not` and `is_nat` are also destructor function symbols. `is_nat`(M) is true iff M is the term for some natural number and false otherwise. `is_nat(fail)` is `fail`.

The term $M - k$ is the application of the destructor `-`k to M, which one might write `-`$k(M)$. `-`$k(M)$ returns N when M consists of k applications of `+1` to the term N. Otherwise, `-`$k(M)$ returns `fail`.

For each destructor $h \in \mathcal{F}_d$, we denote by $\text{def}(h)$ the associated set $\{h(U_{i,1},\ldots,U_{i,n}) \to U_i \mid\mid \phi_i\}_{i=1}^m$ of rewrite rules.

Predicates We denote by $\mathcal{F}_p$ the set of predicates defined by the `pred` declaration. For all $p/n \in \mathcal{F}_p$, we denote by $\text{def}(p)$ the set (possibly infinite) of tuples $(M_1,\ldots,M_n)$ such that $p(M_1,\ldots,M_n)$ is true. When the predicate p is defined with the option `[block]`, $\text{def}(p)$ is not given to ProVerif and so ProVerif will try to prove the query for all possible $\text{def}(p)$. Otherwise $\text{def}(p)$ is defined as the set of tuples $(M_1,\ldots,M_n)$ such that $p(M_1,\ldots,M_n)$ is derivable by the clauses ⟨clauses⟩ defined with the `clauses` declaration from Figure A.2.

Semantics of process terms and pattern-matching A process term corresponds to an element ⟨pterm⟩ from Figure A.1. To describe the semantics of process terms, we extend the definition of ⟨pterm⟩ to allow the `fail` constant, i.e.

$$\langle\text{pterm}\rangle ::= \ldots \mid \texttt{fail}$$

Note that in Figure A.1, some constructs of ⟨pterm⟩ are optional, i.e. `else` ⟨pterm⟩ and `suchthat` ⟨pterm⟩. When omitted, they are respectively syntactic sugar for `else fail` and `suchthat true`.

In what follows, we will denote by D a process term. Some process terms are in fact syntactic sugar for others:

- k `<-R` t`;` D is syntactic sugar for `new` k`:`t`;` D
- $x[:t]$ `<-` D`;` D' is syntactic sugar for `let` $x[:t] = D$ `in` D' `else 0`

We will also denote by $M, N, \ldots$ a process term that can be cast as a ⟨term⟩ that does not contain destructor function symbols; and by $U, V, \ldots$ a process term that can be cast as a ⟨mayfailterm⟩ that does not contain destructor function symbols.

A pattern *pat* corresponds to an element ⟨pattern⟩ from Figure A.1. We will denote by *cpat* a pattern such that for all occurrences of `=`D in *cpat*, D is a may-fail constructor term U. We say that the pattern is *simple* in such a case.

The semantics of *simple pattern-matching* can be formally defined using a function matches such that $\text{matches}(cpat, U)$ returns a substitution σ that gives the value of the variables bound by the pattern, when pattern-matching succeeds. This function is defined as follows:

$$\begin{aligned}
&\text{matches}(x, M) = \{^M/_x\}\\
&\text{matches}(\texttt{=}M, N) = \emptyset \text{ if } M =_{\mathcal{E}} N\\
&\text{matches}(f(cpat_1,\ldots,cpat_n), f(M_1,\ldots,M_n)) = \bigcup_{i\in\{1,\ldots,n\}} \sigma_i\\
&\qquad \text{if matches}(cpat_i, M_i) = \sigma_i \text{ for all } i \in \{1,\ldots,n\}
\end{aligned}$$

Note that if *cpat* contains an occurrence of `=` `fail` or if $U =$ `fail` then $\text{matches}(cpat, U)$ is not defined.

The semantics of process terms and pattern-matching are respectively given by two labeled transition relations $\xrightarrow{\ell}_t$ and $\xrightarrow{\ell}_p$ on term and pattern configurations where ℓ can either be empty or an event

$ev(M_1, \ldots, M_n)$ with ev being one of the events declared with the `event` declaration from Figure A.2. A *term configuration* and *pattern configuration* are respectively the tuples $\mathcal{T}, \mathcal{N}_{\mathrm{pr}}, D$ and $\mathcal{T}, \mathcal{N}_{\mathrm{pr}}, pat$ where

- $\mathcal{T}$ is the state of tables, that is a set of elements of the form $tbl(M_1, \ldots, M_n)$ where tbl is one of the tables declared with the `table` declaration from Figure A.2.
- $\mathcal{N}_{\mathrm{pr}}$ is the set of private names.

We define the relations $\xrightarrow{\ell}_t$ and $\xrightarrow{\ell}_p$ in Figure B.1, where $\mathcal{C}_t$ is the set containing the following contexts of process terms

- `if` $_$ `then` D `else` D'
- `let` $pat = _$ `in` D `else` D'
- `let` $x_1, \ldots, x_m$ `suchthat` p`(`U_1`,` $\ldots$`,`U_{i-1}`,`$_$`,`D_{i+1}`,` $\ldots$`,`D_n`)` `in` D `else` D'
- g`(`U_1`,` $\ldots$`,`U_{i-1}`,`$_$`,`D_{i+1}`,` $\ldots$`,`D_n`)` with $g \in \mathcal{F}_c \cup \mathcal{F}_d$
- `event` ev`(`U_1`,` $\ldots$`,`U_{i-1}`,`$_$`,`D_{i+1}`,` $\ldots$`,`D_n`);`D
- `insert` tbl`(`U_1`,` $\ldots$`,`U_{i-1}`,`$_$`,`D_{i+1}`,` $\ldots$`,`D_n`);`D

and $\mathcal{C}_{pat}$ is the set containing the following contexts of process terms

- `let` $_ = U$ `in` D `else` D'
- `get` tbl`(`$cpat_1$`,` $\ldots$`,`$cpat_{i-1}$`,`$_$`,`pat_{i+1}`,` $\ldots$`,`pat_n`)` `suchthat` D `in` D_1 `else` D_2

The conditions D of `get` never contain `new`, `<-R`, `let ... suchthat`, `event`, `insert`, so in the rules for `get`, $\mathcal{T}$ and $\mathcal{N}_{\mathrm{pr}}$ are indeed unchanged during the evaluation of D.

Semantics of processes

A process corresponds to an element of ⟨process⟩ from Figure A.7. Some processes are in fact syntactic sugar for others:

- `yield` is syntactic sugar for 0
- `!`i `<=` j P and `foreach` i `<=` j `do` P are syntactic sugar for `!` P
- k `<-R` t`;` P is syntactic sugar for `new` k`:`t`;` P
- $x[:t]$ `<-` D`;` P is syntactic sugar for `let` $x[:t] = D$ `in` P `else 0`

Similarly to process terms, when `else` ⟨process⟩, `in` ⟨process⟩ `else` ⟨process⟩, `;` ⟨process⟩, and `suchthat` ⟨pterm⟩ are omitted, they are respectively syntactic sugar for `else 0`, `in 0 else 0`, `; 0`, and `suchthat true`.

Given a process P, we define synchro(P) the set collecting all synchronization parameters that occur in P. Hence, synchro(`sync` n `[`tag`];`Q) $= \{(n, tag)\} \cup$ synchro(Q) and in all other cases, synchro(P) is the union of the synchronization parameters of the immediate subprocesses of P. When the tag `[`tag`]` of a synchronization is omitted, ProVerif uses a new fresh tag.

A process configuration is a tuple $\mathcal{S}, ph, \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \mathcal{P}$ where $\mathcal{S}$ is a set of pairs (integer,tag) representing the synchronization parameters, ph is an integer representing the phase, and $\mathcal{P}$ is a multiset of processes. The initial configuration for a closed process P is synchro(P), $0, \emptyset, \mathcal{N}'_{\mathrm{pr}}, \{P\}$ where $\mathcal{N}'_{\mathrm{pr}}$ is the set of names declared with the `free` declaration from Figure A.2 with the option `[private]`. The semantics of processes is given by a labeled transition relation $\xrightarrow{\ell}$ between process configurations where ℓ can either be empty or an event $ev(M_1, \ldots, M_n)$ with ev being one of the events declared with the `event` declaration from Figure A.2.

This labeled transition relation is defined in Figure B.2, where $\mathcal{C}_t$ is the set containing the following contexts of process terms

Figure B.1 Semantics of process terms and patterns

$\mathcal{T}, \mathcal{N}_{\mathrm{pr}}, f\texttt{(}cpat_1\texttt{,}\ldots\texttt{,}cpat_{i-1}\texttt{,}pat_i\texttt{,}\ldots\texttt{,}pat_n\texttt{)} \xrightarrow{\ell}_p \mathcal{T}', \mathcal{N}'_{\mathrm{pr}}, f\texttt{(}cpat_1\texttt{,}\ldots\texttt{,}cpat_{i-1}\texttt{,}pat'_i\texttt{,}pat_{i+1}\texttt{,}\ldots\texttt{,}pat_n\texttt{)}$
  if $\mathcal{T}, \mathcal{N}_{\mathrm{pr}}, pat_i \xrightarrow{\ell}_p \mathcal{T}', \mathcal{N}'_{\mathrm{pr}}, pat'_i$

$\mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \texttt{=}D \xrightarrow{\ell}_p \mathcal{T}', \mathcal{N}'_{\mathrm{pr}}, \texttt{=}D'$ if $\mathcal{T}, \mathcal{N}_{\mathrm{pr}}, D \xrightarrow{\ell}_t \mathcal{T}', \mathcal{N}'_{\mathrm{pr}}, D'$

$\mathcal{T}, \mathcal{N}_{\mathrm{pr}}, f\texttt{(}M_1\texttt{,}\ldots\texttt{,}M_{i-1}\texttt{,fail,}U_{i+1}\texttt{,}\ldots\texttt{,}U_n\texttt{)} \rightarrow_t \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \texttt{fail}$ if $f \in \mathcal{F}_c$

$\mathcal{T}, \mathcal{N}_{\mathrm{pr}}, h\texttt{(}U_1\texttt{,}\ldots\texttt{,}U_n\texttt{)} \rightarrow_t \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, U'\sigma$
  if $h \in \mathcal{F}_d$, and there exists a rewrite rule $h(U'_1, \ldots, U'_n) \rightarrow U' \mid\mid \phi$ in $\mathrm{def}(h)$
    such that $U'_1\sigma =_{\mathcal{E}} U_1, \ldots, U'_n\sigma =_{\mathcal{E}} U_n$, and $\phi\sigma$ is true

$\mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \texttt{new}\ a; D \rightarrow_t \mathcal{T}, \mathcal{N}_{\mathrm{pr}} \cup \{a'\}, D\{^{a'}/_{a}\}$ where $a' \in \mathcal{N} \setminus (\mathcal{N}_{\mathrm{pub}} \cup \mathcal{N}_{\mathrm{pr}})$

$\mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \texttt{if}\ M\ \texttt{then}\ D\ \texttt{else}\ D' \rightarrow_t \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, D$ if $M =_{\mathcal{E}} \texttt{true}$

$\mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \texttt{if}\ M\ \texttt{then}\ D\ \texttt{else}\ D' \rightarrow_t \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, D'$ if $M \neq_{\mathcal{E}} \texttt{true}$

$\mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \texttt{if fail then}\ D\ \texttt{else}\ D' \rightarrow_t \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \texttt{fail}$

$\mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \texttt{let}\ cpat\ \texttt{=}\ U\ \texttt{in}\ D\ \texttt{else}\ D' \rightarrow_t \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, D\sigma$ if $\mathrm{matches}(cpat, U) = \sigma$

$\mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \texttt{let}\ cpat\ \texttt{=}\ U\ \texttt{in}\ D\ \texttt{else}\ D' \rightarrow_t \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, D'$ if $\mathrm{matches}(cpat, U)$ is not defined

$\mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \texttt{let}\ x_1, \ldots, x_m\ \texttt{suchthat}\ p(M_1, \ldots, M_n)\ \texttt{in}\ D\ \texttt{else}\ D' \rightarrow_t \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, D\sigma$
  if $(M_1, \ldots, M_n)\sigma \in \mathrm{def}(p)$ and for all i, $x_i\sigma$ is a ground constructor term

$\mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \texttt{let}\ x_1, \ldots, x_m\ \texttt{suchthat}\ p(M_1, \ldots, M_n)\ \texttt{in}\ D\ \texttt{else}\ D' \rightarrow_t \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, D'$
  if $(M_1, \ldots, M_n)\sigma \notin \mathrm{def}(p)$ for all σ such that for all i, $x_i\sigma$ is a constructor term

$\mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \texttt{let}\ x_1, \ldots, x_m\ \texttt{suchthat}\ p\texttt{(}M_1\texttt{,}\ldots\texttt{,}M_{i-1}\texttt{,fail,}U_{i+1}\texttt{,}\ldots\texttt{,}U_n\texttt{)}\ \texttt{in}\ D\ \texttt{else}\ D' \rightarrow_t \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \texttt{fail}$

$\mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \texttt{event}\ ev\texttt{(}M_1\texttt{,}\ldots\texttt{,}M_n\texttt{)}; D \xrightarrow{ev(M_1,\ldots,M_n)}_t \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, D$

$\mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \texttt{event}\ ev\texttt{(}M_1\texttt{,}\ldots\texttt{,}M_{i-1}\texttt{,fail,}U_{i+1}\texttt{,}\ldots\texttt{,}U_n\texttt{)}; D \rightarrow_t \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \texttt{fail}$

$\mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \texttt{insert}\ tbl\texttt{(}M_1\texttt{,}\ldots\texttt{,}M_n\texttt{)}; D \rightarrow_t \mathcal{T} \cup \{tbl\texttt{(}M_1\texttt{,}\ldots\texttt{,}M_n\texttt{)}\}, \mathcal{N}_{\mathrm{pr}}, D$

$\mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \texttt{insert}\ tbl\texttt{(}M_1\texttt{,}\ldots\texttt{,}M_{i-1}\texttt{,fail,}U_{i+1}\texttt{,}\ldots\texttt{,}U_n\texttt{)}; D \rightarrow_t \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \texttt{fail}$

$\mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \texttt{get}\ tbl\texttt{(}cpat_1\texttt{,}\ldots\texttt{,}cpat_n\texttt{)}\ \texttt{suchthat}\ D\ \texttt{in}\ D_1\ \texttt{else}\ D_2 \rightarrow_t \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, D_1\sigma$
  if there exists $tbl\texttt{(}M_1\texttt{,}\ldots\texttt{,}M_n\texttt{)} \in \mathcal{T}$ such that
    $\mathrm{matches}((cpat_1, \ldots, cpat_n), (M_1, \ldots, M_n)) = \sigma$ and $\mathcal{T}, \mathcal{N}_{\mathrm{pr}}, D\sigma \rightarrow_t^* \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, M$ with $M =_{\mathcal{E}} \texttt{true}$

$\mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \texttt{get}\ tbl\texttt{(}cpat_1\texttt{,}\ldots\texttt{,}cpat_n\texttt{)}\ \texttt{suchthat}\ D\ \texttt{in}\ D_1\ \texttt{else}\ D_2 \rightarrow_t \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, D_2$
  if for all $tbl\texttt{(}M_1\texttt{,}\ldots\texttt{,}M_n\texttt{)} \in \mathcal{T}$,
    $\mathrm{matches}((cpat_1, \ldots, cpat_n), (M_1, \ldots, M_n)) = \sigma$ and $\mathcal{T}, \mathcal{N}_{\mathrm{pr}}, D\sigma \rightarrow_t^* \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, M$ implies $M \neq_{\mathcal{E}} \texttt{true}$

$\mathcal{T}, \mathcal{N}_{\mathrm{pr}}, C[D] \xrightarrow{\ell}_t \mathcal{T}', \mathcal{N}'_{\mathrm{pr}}, C[D']$ if $\mathcal{T}, \mathcal{N}_{\mathrm{pr}}, D \xrightarrow{\ell}_t \mathcal{T}', \mathcal{N}'_{\mathrm{pr}}, D'$ and $C[_] \in \mathcal{C}_t$

$\mathcal{T}, \mathcal{N}_{\mathrm{pr}}, C[pat] \xrightarrow{\ell}_t \mathcal{T}', \mathcal{N}'_{\mathrm{pr}}, C[pat']$ if $\mathcal{T}, \mathcal{N}_{\mathrm{pr}}, pat \xrightarrow{\ell}_p \mathcal{T}', \mathcal{N}'_{\mathrm{pr}}, pat'$ and $C[_] \in \mathcal{C}_{pat}$

Figure B.2 Semantics of processes

$\mathcal{S}, ph, \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \mathcal{P} \cup \{\mathtt{0}\} \to \mathcal{S}, ph, \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \mathcal{P}$

$\mathcal{S}, ph, \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \mathcal{P} \cup \{P \mid Q\} \to \mathcal{S}, ph, \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \mathcal{P} \cup \{P, Q\}$

$\mathcal{S}, ph, \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \mathcal{P} \cup \{\mathtt{!}P\} \to \mathcal{S}, ph, \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \mathcal{P} \cup \{\mathtt{!}P, P\}$

$\mathcal{S}, ph, \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \mathcal{P} \cup \{\mathtt{new}\ a; P\} \to \mathcal{S}, ph, \mathcal{T}, \mathcal{N}_{\mathrm{pr}} \cup \{a'\}, \mathcal{P} \cup \{P\{^{a'}/_{a}\}\}$

where $a' \in \mathcal{N} \setminus (\mathcal{N}_{\mathrm{pub}} \cup \mathcal{N}_{\mathrm{pr}})$

$\mathcal{S}, ph, \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \mathcal{P} \cup \{\mathtt{out}(N, M); Q, \mathtt{in}(N', pat); P\} \to \mathcal{S}, ph, \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \mathcal{P} \cup \{Q, \mathtt{let}\ pat\ \mathtt{=}\ M\ \mathtt{in}\ P\}$

if $N =_{\mathcal{E}} N'$

$\mathcal{S}, ph, \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \mathcal{P} \cup \{\mathtt{if}\ M\ \mathtt{then}\ P\ \mathtt{else}\ Q\} \to \mathcal{S}, ph, \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \mathcal{P} \cup \{P\}$ if $M =_{\mathcal{E}} \mathtt{true}$

$\mathcal{S}, ph, \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \mathcal{P} \cup \{\mathtt{if}\ M\ \mathtt{then}\ P\ \mathtt{else}\ Q\} \to \mathcal{S}, ph, \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \mathcal{P} \cup \{Q\}$ if $M \neq_{\mathcal{E}} \mathtt{true}$

$\mathcal{S}, ph, \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \mathcal{P} \cup \{\mathtt{let}\ cpat\ \mathtt{=}\ U\ \mathtt{in}\ P\ \mathtt{else}\ Q\} \to \mathcal{S}, ph, \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \mathcal{P} \cup \{P\sigma\}$

if $\mathrm{matches}(cpat, U) = \sigma$

$\mathcal{S}, ph, \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \mathcal{P} \cup \{\mathtt{let}\ cpat\ \mathtt{=}\ U\ \mathtt{in}\ P\ \mathtt{else}\ Q\} \to \mathcal{S}, ph, \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \mathcal{P} \cup \{Q\}$

if $\mathrm{matches}(cpat, U)$ is not defined

$\mathcal{S}, ph, \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \mathcal{P} \cup \{\mathtt{let}\ x_1, \ldots, x_m\ \mathtt{suchthat}\ p(M_1, \ldots, M_n)\ \mathtt{in}\ P\ \mathtt{else}\ Q\} \to \mathcal{S}, ph, \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \mathcal{P} \cup \{P\sigma\}$

if $(M_1, \ldots, M_n)\sigma \in \mathrm{def}(p)$ and for all i, $x_i\sigma$ is a ground constructor term

$\mathcal{S}, ph, \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \mathcal{P} \cup \{\mathtt{let}\ x_1, \ldots, x_m\ \mathtt{suchthat}\ p(M_1, \ldots, M_n)\ \mathtt{in}\ P\ \mathtt{else}\ Q\} \to \mathcal{S}, ph, \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \mathcal{P} \cup \{Q\}$

if $(M_1, \ldots, M_n)\sigma \notin \mathrm{def}(p)$ for all σ such that for all i, $x_i\sigma$ is a constructor term

$\mathcal{S}, ph, \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \mathcal{P} \cup \{\mathtt{event}\ ev(M_1, \ldots, M_n); P\} \xrightarrow{ev(M_1, \ldots, M_n)} \mathcal{S}, ph, \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \mathcal{P} \cup \{P\}$

$\mathcal{S}, ph, \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \mathcal{P} \cup \{\mathtt{insert}\ tbl(M_1, \ldots, M_n); P\} \to \mathcal{S}, ph, \mathcal{T} \cup \{tbl(M_1, \ldots, M_n)\}, \mathcal{N}_{\mathrm{pr}}, \mathcal{P} \cup \{P\}$

$\mathcal{S}, ph, \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \mathcal{P} \cup \{\mathtt{get}\ tbl(cpat_1, \ldots, cpat_n)\ \mathtt{suchthat}\ D\ \mathtt{in}\ P\ \mathtt{else}\ Q\} \to \mathcal{S}, ph, \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \mathcal{P} \cup \{P\sigma\}$

if there exists $tbl(M_1, \ldots, M_n) \in \mathcal{T}$ such that

$\mathrm{matches}((cpat_1, \ldots, cpat_n), (M_1, \ldots, M_n)) = \sigma$ and $\mathcal{T}, \mathcal{N}_{\mathrm{pr}}, D\sigma \to_t^* \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, M$ with $M =_{\mathcal{E}} \mathtt{true}$

$\mathcal{S}, ph, \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \mathcal{P} \cup \{\mathtt{get}\ tbl(cpat_1, \ldots, cpat_n)\ \mathtt{suchthat}\ D\ \mathtt{in}\ P\ \mathtt{else}\ Q\} \to \mathcal{S}, ph, \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \mathcal{P} \cup \{Q\}$

if for all $tbl(M_1, \ldots, M_n) \in \mathcal{T}$,

$\mathrm{matches}((cpat_1, \ldots, cpat_n), (M_1, \ldots, M_n)) = \sigma$ and $\mathcal{T}, \mathcal{N}_{\mathrm{pr}}, D\sigma \to_t^* \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, M$ implies $M \neq_{\mathcal{E}} \mathtt{true}$

$\mathcal{S}, ph, \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \mathcal{P} \cup \mathcal{P}_{>ph+1} \cup \{\mathtt{phase}\ (ph+1); P_i\}_{i=1}^k \to \mathcal{S}, ph+1, \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \mathcal{P}_{>ph+1} \cup \{P_i\}_{i=1}^k$

if all processes of $\mathcal{P}$ do not start with some $\mathtt{phase}\ ph'$, $ph' > ph$

and all processes of $\mathcal{P}_{ph+1}$ start with some $\mathtt{phase}\ ph'$, $ph' > ph+1$

$\mathcal{S}, ph, \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \mathcal{P} \cup \{\mathtt{sync}\ n\ [tag_i]; P_i\}_{i=1}^k \to \mathcal{S} \setminus \{(n, tag_i)\}_{i=1}^k, ph, \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \mathcal{P} \cup \{P_i\}_{i=1}^k$

if $k \geq 1$ and if $(n', tag') \in \mathcal{S} \setminus \{(n, tag_i)\}_{i=1}^k$ then $n' > n$

$\mathcal{S}, ph, \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \mathcal{P} \cup \{C[D]\} \xrightarrow{\ell} \mathcal{S}, ph, \mathcal{T}', \mathcal{N}'_{\mathrm{pr}}, \mathcal{P} \cup \{C[D']\}$

if $\mathcal{T}, \mathcal{N}_{\mathrm{pr}}, D \xrightarrow{\ell}_t \mathcal{T}', \mathcal{N}'_{\mathrm{pr}}, D'$ and $C[_] \in \mathcal{C}_t$

$\mathcal{S}, ph, \mathcal{T}, \mathcal{N}_{\mathrm{pr}}, \mathcal{P} \cup \{C[pat]\} \xrightarrow{\ell} \mathcal{S}, ph, \mathcal{T}', \mathcal{N}'_{\mathrm{pr}}, \mathcal{P} \cup \{C[pat']\}$

if $\mathcal{T}, \mathcal{N}_{\mathrm{pr}}, pat \xrightarrow{\ell}_p \mathcal{T}', \mathcal{N}'_{\mathrm{pr}}, pat'$ and $C[_] \in \mathcal{C}_{pat}$

- `out(`$_$`,`D`);`P
- `out(`U`,`$_$`);`P
- `in(`$_$`,`pat`);`P
- `if` $_$ `then` P `else` Q
- `let` $pat = _$ `in` P `else` Q
- `let` $x_1, \ldots, x_m$ `suchthat` p`(`U_1`,`$\ldots$`,`U_{i-1}`,`$_$`,`D_{i+1}`,`$\ldots$`,`D_n`)` `in` P `else` Q
- `event` ev`(`U_1`,`$\ldots$`,`U_{i-1}`,`$_$`,`D_{i+1}`,`$\ldots$`,`D_n`);`P
- `insert` tbl`(`U_1`,`$\ldots$`,`U_{i-1}`,`$_$`,`D_{i+1}`,`$\ldots$`,`D_n`);`P

and $\mathcal{C}_{pat}$ is the set containing the following contexts of process terms

- `let` $_ = U$ `in` P `else` Q
- `get` tbl`(`$cpat_1$`,`$\ldots$`,`$cpat_{i-1}$`,`$_$`,`pat_{i+1}`,`$\ldots$`,`pat_n`)` `suchthat` D `in` P `else` Q

In the rules for `get`, the same comment holds as in the semantics of process terms. Synchronizations `sync` n `[`tag`]` with the same (n, tag) are allowed to occur only in different branches of `if`, `let`, `let` ... `suchthat`, `get`, and synchronizations never occur under replications, so the tags tag_i are all distinct in the rule for `sync`.

Bibliography

[AB05a] Martín Abadi and Bruno Blanchet. Analyzing security protocols with secrecy types and logic programs. *Journal of the ACM*, 52(1):102–146, January 2005.

[AB05b] Martín Abadi and Bruno Blanchet. Computer-assisted verification of a protocol for certified email. *Science of Computer Programming*, 58(1–2):3–27, October 2005. Special issue SAS'03.

[AB05c] Xavier Allamigeon and Bruno Blanchet. Reconstruction of attacks against cryptographic protocols. In *18th IEEE Computer Security Foundations Workshop (CSFW-18)*, pages 140–154, Aix-en-Provence, France, June 2005. IEEE.

[Aba00] Martín Abadi. Security protocols and their properties. In F.L. Bauer and R. Steinbrueggen, editors, *Foundations of Secure Computation*, NATO Science Series, pages 39–60. IOS Press, 2000. Volume for the 20th International Summer School on Foundations of Secure Computation, held in Marktoberdorf, Germany (1999).

[ABB+04] William Aiello, Steven M. Bellovin, Matt Blaze, Ran Canetti, John Ioannidis, Keromytis Keromytis, and Omer Reingold. Just Fast Keying: Key agreement in a hostile Internet. *ACM Transactions on Information and System Security*, 7(2):242–273, May 2004.

[ABCL09] Martín Abadi, Bruno Blanchet, and Hubert Comon-Lundh. Models and proofs of protocol security: A progress report. In Ahmed Bouajjani and Oded Maler, editors, *21st International Conference on Computer Aided Verification (CAV'09)*, volume 5643 of *Lecture Notes in Computer Science*, pages 35–49, Grenoble, France, June 2009. Springer.

[ABF07] Martín Abadi, Bruno Blanchet, and Cédric Fournet. Just Fast Keying in the pi calculus. *ACM Transactions on Information and System Security (TISSEC)*, 10(3):1–59, July 2007.

[ABF17] Martín Abadi, Bruno Blanchet, and Cédric Fournet. The applied pi calculus: Mobile values, new names, and secure communication. *Journal of the ACM*, 65(1):1:1–1:41, October 2017.

[AF01] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*, pages 104–115, London, United Kingdom, January 2001. ACM Press.

[AFP06] Michel Abdalla, Pierre-Alain Fouque, and David Pointcheval. Password-based authenticated key exchange in the three-party setting. *IEE Proceedings Information Security*, 153(1):27–39, March 2006.

[AGHP02] Martín Abadi, Neal Glew, Bill Horne, and Benny Pinkas. Certified email with a light on-line trusted third party: Design and implementation. In *11th International World Wide Web Conference*, pages 387–395, Honolulu, Hawaii, May 2002. ACM Press.

[AN95] Ross Anderson and Roger Needham. Programming Satan's computer. In J. van Leeuven, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 426–440. Springer, 1995.

[AN96] Martín Abadi and Roger Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, January 1996.

[BAF08] Bruno Blanchet, Martín Abadi, and Cédric Fournet. Automated verification of selected equivalences for security protocols. *Journal of Logic and Algebraic Programming*, 75(1):3–51, February–March 2008.

[BAN89] Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication. *Proceedings of the Royal Society of London A*, 426(1871):233–271, dec 1989. A preliminary version appeared as Digital Equipment Corporation Systems Research Center report No. 39, February 1989.

[BBK17] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *IEEE Symposium on Security and Privacy (S&P'17)*, pages 483–503, Los Alamitos, CA, May 2017. IEEE Computer Society Press.

[BC08] Bruno Blanchet and Avik Chaudhuri. Automated formal analysis of a protocol for secure file sharing on untrusted storage. In *IEEE Symposium on Security and Privacy*, pages 417–431, Oakland, CA, May 2008. IEEE.

[BCC04] Ernie Brickell, Jan Camenisch, and Liqun Chen. Direct Anonymous Attestation. In *CCS '04: 11th ACM conference on Computer and communications security*, pages 132–145, New York, USA, 2004. ACM Press.

[BCFZ08] Karthikeyan Bhargavan, Ricardo Corin, Cédric Fournet, and Eugen Zălinescu. Cryptographically verified implementations for TLS. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)*, pages 459–468. ACM, October 2008.

[BDM20] David Baelde, Stéphanie Delaune, and Solène Moreau. A method for proving unlinkability of stateful protocols. In *33rd IEEE Computer Security Foundations Symposium (CSF'20)*, pages 169–183. IEEE, June 2020.

[BFG06] Karthikeyan Bhargavan, Cédric Fournet, and Andrew Gordon. Verified reference implementations of WS-Security protocols. In Mario Bravetti, Manuel Núñez, and Gianluigi Zavattaro, editors, *3rd International Workshop on Web Services and Formal Methods (WS-FM 2006)*, volume 4184 of *Lecture Notes in Computer Science*, pages 88–106, Vienna, Austria, September 2006. Springer.

[BFGS08] Karthikeyan Bhargavan, Cédric Fournet, Andrew Gordon, and Nikhil Swamy. Verified implementations of the information card federated identity-management protocol. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS'08)*, pages 123–135, Tokyo, Japan, March 2008. ACM.

[BFGT06] Karthikeyan Bhargavan, Cédric Fournet, Andrew Gordon, and Stephen Tse. Verified interoperable implementations of security protocols. In *19th IEEE Computer Security Foundations Workshop (CSFW'06)*, pages 139–152, Venice, Italy, July 2006. IEEE Computer Society.

[BHM08] Michael Backes, Catalin Hritcu, and Matteo Maffei. Automated verification of remote electronic voting protocols in the applied pi-calculus. In *21st IEEE Computer Security Foundations Symposium (CSF'08)*, pages 195–209, Pittsburgh, PA, June 2008. IEEE Computer Society.

[Bla04] Bruno Blanchet. Automatic proof of strong secrecy for security protocols. In *IEEE Symposium on Security and Privacy*, pages 86–100, Oakland, California, May 2004.

[Bla05] Bruno Blanchet. Security protocols: From linear to classical logic by abstract interpretation. *Information Processing Letters*, 95(5):473–479, September 2005.

[Bla07] Bruno Blanchet. Computationally sound mechanized proofs of correspondence assertions. In *20th IEEE Computer Security Foundations Symposium (CSF'07)*, pages 97–111, Venice, Italy, July 2007. IEEE. Extended version available as ePrint Report 2007/128, `http://eprint.iacr.org/2007/128`.

[Bla09] Bruno Blanchet. Automatic verification of correspondences for security protocols. *Journal of Computer Security*, 17(4):363–434, July 2009.

[Bla11] Bruno Blanchet. Using Horn clauses for analyzing security protocols. In Véronique Cortier and Steve Kremer, editor, *Formal Models and Techniques for Analyzing Security Protocols*, volume 5 of *Cryptology and Information Security Series*, chapter 5, pages 86–111. IOS Press, March 2011.

[Bla16] Bruno Blanchet. Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Foundations and Trends in Privacy and Security*, 1(1–2):1–135, October 2016.

[Bla17] Bruno Blanchet. Symbolic and computational mechanized verification of the ARINC823 avionic protocols. In *30th IEEE Computer Security Foundations Symposium (CSF'17)*, pages 68–82, Los Alamitos, CA, August 2017. IEEE Computer Society Press.

[BM92] Steven M. Bellovin and Michael Merritt. Encrypted Key Exchange: Password-based protocols secure against dictionary attacks. In *Proceedings of the 1992 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 72–84, May 1992.

[BM93] Steven M. Bellovin and Michael Merritt. Augmented Encrypted Key Exchange: a password-based protocol secure against dictionary attacks and password file compromise. In *Proceedings of the First ACM Conference on Computer and Communications Security*, pages 244–250, November 1993.

[BMU08] Michael Backes, Matteo Maffei, and Dominique Unruh. Zero-knowledge in the applied pi-calculus and automated verification of the direct anonymous attestation protocol. In *29th IEEE Symposium on Security and Privacy*, pages 202–215, Oakland, CA, May 2008. IEEE. Technical report version available at `http://eprint.iacr.org/2007/289`.

[BP05] Bruno Blanchet and Andreas Podelski. Verification of cryptographic protocols: Tagging enforces termination. *Theoretical Computer Science*, 333(1-2):67–90, March 2005. Special issue FoSSaCS'03.

[BS16] Bruno Blanchet and Ben Smyth. Automated reasoning for equivalences in the applied pi calculus with barriers. In *29th IEEE Computer Security Foundations Symposium (CSF'16)*, Lisboa, Portugal, June 2016. IEEE.

[BS18] Bruno Blanchet and Ben Smyth. Automated reasoning for equivalences in the applied pi calculus with barriers. *Journal of Computer Security*, 26(3):367–422, 2018.

[CB13] Vincent Cheval and Bruno Blanchet. Proving more observational equivalences with ProVerif. In David Basin and John Mitchell, editors, *2nd Conference on Principles of Security and Trust (POST 2013)*, volume 7796 of *Lecture Notes in Computer Science*, pages 226–246, Rome, Italy, March 2013. Springer.

[CCT18] Vincent Cheval, Véronique Cortier, and Mathieu Turuani. A little more conversation, a little less action, a lot more satisfaction: Global states in proverif. In Steve Chong and Stéphanie Delaune, editors, *Proceedings of the 30th IEEE Computer Security Foundations Symposium (CSF'18)*, Oxford, UK, July 2018. IEEE Computer Society Press.

[CHW06] Véronique Cortier, Heinrich Hördegen, and Bogdan Warinschi. Explicit randomness is not necessary when modeling probabilistic encryption. In C. Dima, M. Minea, and F.L. Tiplea, editors, *Workshop on Information and Computer Security (ICS 2006)*, volume 186 of *Electronic Notes in Theoretical Computer Science*, pages 49–65, Timisoara, Romania, September 2006. Elsevier.

[CR09] Liqun Chen and Mark Ryan. Attack, solution and verification for shared authorisation data in TCG TPM. In *Proc. Sixth Formal Aspects in Security and Trust (FAST'09)*, volume 5983 of *Lecture Notes in Computer Science*. Springer, 2009.

[CT08] Liqun Chen and Qiang Tang. Bilateral unknown key-share attacks in key agreement protocols. *Journal of Universal Computer Science*, 14(3):416–440, February 2008.

[DDKS17] Jannik Dreier, Charles Duménil, Steve Kremer, and Ralf Sasse. Beyond subterm-convergent equational theories in automated verification of stateful protocols. In Matteo Maffei and Mark Ryan, editors, *6th International Conference on Principles of Security and Trust (POST)*, volume 10204 of *Lecture Notes in Computer Science*, pages 117–140, Uppsala, Sweden, April 2017. Springer.

[DJ04] Stéphanie Delaune and Florent Jacquemard. A theory of dictionary attacks and its complexity. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSFW'04)*, pages 2–15, Asilomar, Pacific Grove, California, USA, June 2004. IEEE Computer Society Press.

[DKR09] Stéphanie Delaune, Steve Kremer, and Mark D. Ryan. Verifying privacy-type properties of electronic voting protocols. *Journal of Computer Security*, 17(4):435–487, 2009.

[DRS08] Stéphanie Delaune, Mark D. Ryan, and Ben Smyth. Automatic verification of privacy properties in the applied pi-calculus. In Yuecel Karabulut, John Mitchell, Peter Herrmann, and Christian Damsgaard Jensen, editors, *Proceedings of the 2nd Joint iTrust and PST Conferences on Privacy, Trust Management and Security (IFIPTM'08)*, volume 263 of *IFIP Conference Proceedings*, pages 263–278. Springer, June 2008. An extended version of this paper appears in [Smy11, Chapters 4 & 5].

[DS81] Dorothy E. Denning and Giovanni Maria Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8):533–536, August 1981.

[DvOW92] Whitfield Diffie, Paul C. van Oorschot, and Michael J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2(2):107–125, June 1992.

[DY83] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(12):198–208, March 1983.

[GJ03] Andrew Gordon and Alan Jeffrey. Authenticity by typing for security protocols. *Journal of Computer Security*, 11(4):451–521, 2003.

[HBD19] Lucca Hirschi, David Baelde, and Stéphanie Delaune. A method for unbounded verification of privacy-type properties. *Journal of Computer Security*, 27(3):277–342, June 2019.

[HLS00] James Heather, Gavin Lowe, and Steve Schneider. How to prevent type flaw attacks on security protocols. In *13th IEEE Computer Security Foundations Workshop (CSFW-13)*, pages 255–268, Cambridge, England, July 2000.

[KBB17] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *2nd IEEE European Symposium on Security and Privacy (EuroS&P'17)*, pages 435–450, Los Alamitos, CA, April 2017. IEEE Computer Society Press.

[KR05] Steve Kremer and Mark D. Ryan. Analysis of an electronic voting protocol in the applied pi calculus. In Mooly Sagiv, editor, *Programming Languages and Systems: 14th European Symposium on Programming, ESOP 2005*, volume 3444 of *Lecture Notes in Computer Science*, pages 186–200, Edimbourg, UK, April 2005. Springer.

[Kra96] Hugo Krawczyk. SKEME: A versatile secure key exchange mechanism for internet. In *Internet Society Symposium on Network and Distributed Systems Security*, February 1996. Available at `http://bilbo.isu.edu/sndss/sndss96.html`.

[KRS+03] Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kvin Fu. Plutus: Scalable secure file sharing on untrusted storage. In *2nd Conference on File and Storage Technologies (FAST'03)*, pages 29–42, San Francisco, CA, April 2003. Usenix.

[KT08] Ralf Küsters and Tomasz Truderung. Reducing protocol analysis with XOR to the XOR-free case in the Horn theory based approach. In *Proceedings of the 15th ACM conference on Computer and communications security (CCS'08)*, pages 129–138, Alexandria, Virginia, USA, October 2008. ACM.

[KT09] Ralf Küsters and Tomasz Truderung. Using ProVerif to analyze protocols with Diffie-Hellman exponentiation. In *22nd IEEE Computer Security Foundations Symposium (CSF'09)*, pages 157–171, Port Jefferson, New York, USA, July 2009. IEEE.

[Low96] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer, 1996.

[Low97] Gavin Lowe. A hierarchy of authentication specifications. In *10th Computer Security Foundations Workshop (CSFW '97)*, pages 31–43, Rockport, Massachusetts, June 1997. IEEE Computer Society.

[MvOV96] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of applied cryptography*. CRC, 1996.

[NS78] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[NS87] Roger M. Needham and Michael D. Schroeder. Authentication revisited. *Operating Systems Review*, 21(1):7, 1987.

[OR87] Dave Otway and Owen Rees. Efficient and timely mutual authentication. *Operating Systems Review*, 21(1):8–10, 1987.

[Pau98] Larry C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1–2):85–128, 1998.

[RS11] Mark Ryan and Ben Smyth. Applied pi calculus. In Véronique Cortier and Steve Kremer, editor, *Formal Models and Techniques for Analyzing Security Protocols*, volume 5 of *Cryptology and Information Security Series*, chapter 6, pages 112–142. IOS Press, March 2011.

[Sch13] Stephan Schulz. Simple and efficient clause subsumptionwith feature vector indexing. In Maria Paola Bonacina and Mark E. Stickel, editors, *Automated Reasoning and Mathematics, Essays in Memory of William W. McCune*, volume 7788 of *Lecture Notes in Computer Science*, pages 45–67. Springer, 2013.

[Smy11] Ben Smyth. *Formal verification of cryptographic protocols with automated reasoning*. PhD thesis, School of Computer Science, University of Birmingham, 2011.

[SRC07] Ben Smyth, Mark Ryan, and Liqun Chen. Direct Anonymous Attestation (DAA): Ensuring privacy with corrupt administrators. In *ESAS'07: 4th European Workshop on Security and Privacy in Ad hoc and Sensor Networks*, volume 4572 of *Lecture Notes in Computer Science*, pages 218–231. Springer, 2007. An extended version of this paper appears in [Smy11, Chapter 4].

[SRK10] Ben Smyth, Mark Ryan, and Steve Kremer. Election verifiability in electronic voting protocols. In Dimitris Gritzalis, Bart Preneel, and Marianthi Theoharidou, editors, *15th European Symposium on Research in Computer Security (ESORICS'10)*, volume 6345 of *Lecture Notes in Computer Science*, pages 389–404, Athens, Greece, September 2010. Springer. An extended version of this paper appears in [Smy11, Chapter 3].

[SRKK10] Ben Smyth, Mark Ryan, Steve Kremer, and Mounira Kourjieh. Towards automatic analysis of election verifiability properties. In Alessandro Armando and Gavin Lowe, editors, *ARSPA-WITS'10: Proceedings of the Joint Workshop on Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security*, volume 6186 of *Lecture Notes in Computer Science*, pages 165–182. Springer, March 2010. An extended version of this paper appears in [Smy11, Chapter 3].

[WL92] Thomas Y. C. Woo and Simon S. Lam. Authentication for distributed systems. *Computer*, 25(1):39–52, January 1992.

[WL93] Thomas Y. C. Woo and Simon S. Lam. A semantic model for authentication protocols. In *Proceedings IEEE Symposium on Research in Security and Privacy*, pages 178–194, Oakland, California, May 1993.

[WL97] Thomas Y. C. Woo and Simon S. Lam. Authentication for distributed systems. In Dorothy Denning and Peter Denning, editors, *Internet Besieged: Countering Cyberspace Scofflaws*, pages 319–355. ACM Press and Addison-Wesley, October 1997.

[Yub10] Yubico AB. *The YubiKey Manual - Usage, configuration and introduction of basic concepts (Version 2.2)*, 2010.