RAPPORT TECHNIQUE PROUVÉ

The PROUVÉ manual: specifications, semantics, and logics

Auteur	:	Steve Kremer, Yassine Lakhnech, Ralf Treinen

Date	:	December 2, 2005
Rapport PROUVÉ numéro	:	7
Version	:	2.0.0

Loria
CNRS UMR 7503,
Campus Scientifique - BP 239
54506 Vandoeuvre-lès-nancy cedex
www.loria.fr

Laboratoire Spécification Vérification
CNRS UMR 8643, ENS Cachan
61, avenue du président-Wilson
94235 Cachan Cedex, France
www.lsv.ens-cachan.fr

Laboratoire Verimag
CNRS UMR 5104,
Univ. Joseph Fourier, INPG
2 av. de Vignate,
38610 Gières, France
www-verimag.imag.fr

Cril Technology
9/11 rue Jeanne Braconnier
92360 Meudon La Foret Cedex, France
www.cril.fr

France Telecom
Div. Recherche et Développement
38, 40 rue du Général Leclerc
92794 Issy Moulineaux Cedex
www.rd.francetelecom.fr

Adresse : Laboratoire Spécification et Vérification,
CNRS UMR 8643, ENS de Cachan
61, Avenue du Président Wilson, 94230 Cachan, France

VERIMAG
2 Av. de Vignate, F-38610 Gieres, France.

Résumé : In this report we describe the PROUVÉ specification language for cryptographic protocols. A main feature of the language is that it separates the roles of a protocol, which are defined in a simple imperative programming language, from the scenario which defines how instances of the roles are created.

We give a formal semantics of the protocol specification language, and define both an expressive logics for safety conditions of protocols and a more limited assertion language.

This version of the report (2.0.x) describes version 2.0 of the PROUVÉ language.

Contents

1 Introduction **3**

2 The Syntax of Specification Files **4**

2.1 Introduction 4

2.2 Elements of the Grammar 4

2.3 The Five Main Sections of a Protocol Specification 8

2.4 Lexical Tokens 15

2.5 Infix Operators 15

2.6 The Default Signature 15

3 Semantics **19**

3.1 Preliminaries 20

3.2 Scenarios 20

3.3 Roles 23

3.4 Semantic Domains 24

4 A Logic for Traces **26**

4.1 Syntax 26

4.2 Semantics 27

5 Security Assertions **31**

5.1 A Language for Security Assertions 31

5.2 Examples 32

6 Examples **35**

6.1 The IKA-1 Protocol 35

6.2 The Needham-Schroeder Public Key Protocol 36

6.3 The TMN Protocol 38

6.4 The Electronic Purse Protocol with Symmetric Keys 40
6.5 The Electronic Purse Protocol with Asymmetric Keys 42

Chapter 1

Introduction

The purpose of the PROUVÉ protocol specification language is to give means to describe both protocols and the context in which they are used. It should allow to specify a signature of message constructors together with equational axioms defining the semantics of message constructors.

The sub-language used to define protocols is oriented on existing imperative programming languages. Other projects, like EVA, choose to use a more abstract, supposedly user-friendly language using the so-called Alice-Bob notation. This approach raises the need for a compiler from the user language to some kind of programming language that serves as input language for the verification tools. It is useful to understand how a correct protocol run is supposed to look like but does not describe in an unambiguous way the actions of the protocol participants. We felt that this approach hides from the shortcomings of the Alice-Bob notation, and choose to have the user specify the protocol as seen by the protocol agents, rather then from the point of view of an outside observer of a correct protocol execution. As a consequence, this sub-language describes *roles* which are the programs executed by (legitimate) protocol participants.

We used constructs from imperative programming languages as building blocks for the sub-language describing roles. However, our language is deliberately not Turing-complete in that we excluded constructs that are either not useful for describing protocols or too complex to handle by our verification tools. The basic instructions are sending and receiving messages and pattern matching. Instructions are composed by serial composition and conditional branching. All variables have to be explicitly declared (either as local variables, or as parameters). Variables can be write-once (that is, logical variables) or mutable variables.

Roles are composed in so-called *Scenarios*. The sub-language for scenarios contains constructs for parallel and sequential composition, and non-deterministic choice of values. It has assignment statements, and allows to instantiate roles that have been previously defined.

The accompanying assertion language allows to express safety properties of execution traces of protocol scenarios.

The design of the PROUVÉ specification language has been inspired by the languages muCAPSL [MD02] and CASRUL [JRV00].

Chapter 2

The Syntax of Specification Files

2.1 Introduction

In this chapter we present the concrete syntax of specification files, as well as the pre-defined operators and their types. The context-free syntax will be formally defined while restrictions which go beyond context-free grammars, as for instance prohibition of multiple definition of the same variable in one scope, will only be informally defined. In this chapter we will only give some informal explanations of the intended semantics; a formal semantics will be defined in Chapter 3. Some examples of protocol specifications are given in Chapter 6.

Notation We use a *generalized context-free grammar* notation for the formal definition of the concrete syntax, that is a context-free grammar where we allow on the right-hand-side of rules disjunctions (noted $|$ in infix notation), and unary operators in postfix notation for an arbitrary number of iterations ($*$), an arbitrary non-null number of iterations ($+$), and for optional parts ($?$). These notations are used for the sake of conciseness of the presentation but do not add to the expressiveness of context-free grammars.

Non-terminals (i.e., symbols defined by grammar rules) and tokens (i.e., symbols which denote an abstraction of some regular language described by a regular expression) are written between angular brackets $\langle$ and $\rangle$. Keywords are $\underline{\text{underlined}}$.

2.2 Elements of the Grammar

2.2.1 Variable Declarations

Variable declarations are employed by several syntactic constructs. They are used to declare the variables used in equational axioms (Section 2.3.2), to declare variables in the top-level scope or in nested scopes of roles and to declare parameters of roles (Section 2.3.3), and to declare variables in nested scopes used in scenarios (Section 2.3.5). A more general form of variable declarations is defined in Section 2.3.4.

One *declaration* ($\langle$declaration$\rangle$) introduces a non-empty list of variable names separated by commas, together with an optional type and an optional specifier $\underline{\text{mutable}}$. A declaration list ($\langle$decllist$\rangle$) is a possibly empty list of declarations that are each terminated by a semi-colon, while a *non-terminated declaration list* ($\langle$nt-decllist$\rangle$) is a non-empty list of declarations that are separated by semi-colons. This latter form of declaration lists is used to declare parameters of roles.

Context-Free Syntax

$$
\begin{array}{rcl}
\langle\text{ne-identlist}\rangle & \rightarrow & (\ \langle\text{ident}\rangle\ ,\ )*\ \langle\text{ident}\rangle \\
\langle\text{declaration}\rangle & \rightarrow & \langle\text{ne-identlist}\rangle\ |\ \langle\text{ne-identlist}\rangle\ :\ \langle\text{typeexp}\rangle \\
 & | & \langle\text{ne-identlist}\rangle\ :\ \underline{\text{mutable}}\ \ \langle\text{typeexp}\rangle\ ? \\
\langle\text{decllist}\rangle & \rightarrow & (\ \langle\text{declaration}\rangle\ ;\ )* \\
\langle\text{nt-decllist}\rangle & \rightarrow & (\ \langle\text{declaration}\rangle\ ;\ )*\ \langle\text{declaration}\rangle
\end{array}
$$

Static Semantics Checks

- All type expressions in a declaration must be arrow-free and variable-free.
- No variable may be declared twice in a declaration list.

Indications on the Semantics

A variable can be declared as *logical* (this is the default) or as *mutable* (indicated by the key word mutable). Logical variables can be assigned only once while mutable variables can be assigned any number of times. If no type is given in the declaration then the universal type *message* is assumed.

2.2.2 Type Expressions

Type expressions ($\langle$typeexp$\rangle$) are composed out of basic types and type variables by arrow, tupling, and application of named type constructors. Type variables are distinguished from names of basic types by a leading apostrophe (').

A *non-empty type expression list* ($\langle$ne-typeexp-list$\rangle$) is a non-empty and comma-separated list of type expressions.

Context-Free Grammar

$$
\begin{array}{rcl}
\langle\text{typeexp}\rangle & \rightarrow & \langle\text{ident}\rangle \\
 & | & '\,\langle\text{ident}\rangle \\
 & | & (\ \langle\text{ne-typeexp-list}\rangle\ ) \\
 & | & \langle\text{typeexp}\rangle\ -\!>\ \langle\text{typeexp}\rangle \\
 & | & \langle\text{ident}\rangle\ (\ \langle\text{ne-typeexp-list}\rangle\ ) \\
\langle\text{ne-typeexp-list}\rangle & \rightarrow & (\ \langle\text{typeexp}\rangle\ ,\ )*\ \langle\text{typeexp}\rangle
\end{array}
$$

2.2.3 Static Semantics Checks

- Only basic types and type constructors listed in Section 2.6.1 are permitted. Type constructors may only be used according to their arity.
- Only arrow-free type expressions, or type expressions of the form $t_1 \to t_2$, where t_1 and t_2 are arrow-free, are allowed. The rationale of this restriction is that it guarantees any well-typed expression to have a non-functional type.
- The set of free variables of a type expression is defined as follows:

$$\begin{aligned}
\mathcal{V}(\text{basictype}) &= \emptyset \\
\mathcal{V}(\text{typevariable}) &= \{\text{typevariable}\} \\
\mathcal{V}((t_1,\ldots,t_n)) &= \bigcup_{i=1}^{n} \mathcal{V}(t_i) \\
\mathcal{V}(\text{typeconstructor}(t_1,\ldots,t_n)) &= \bigcup_{i=1}^{n} \mathcal{V}(t_i) \\
\mathcal{V}(t_1 \to t_2) &= \mathcal{V}(t_2) - \mathcal{V}(t_1)
\end{aligned}$$

  Every type expression *which is not part of another type expression* must not have any free variables. The rationale of this restriction is that it guarantees every well-typed expression to have a variable-free type.

2.2.4 Expressions

Expressions denote values. They occur in equational axioms (Section 2.3.2) and as part of instructions used in roles (Section 2.3.3) and scenarios (Section 2.3.5). They are a special case of patterns (Section 2.2.5) and are a constituent of l-expressions (Section 2.2.6).

An *expression* ($\langle$expr$\rangle$) is either an integer constant, an identifier (which can either be a constant or a variable), an identifier applied to a non-empty list of expressions (denoting function application), two expressions joined by a binary infix operator, a parenthesized expression, a tuple expression, a list expression, or an association list expression. A *tuple expression* is a comma-separated list of expressions (at least two expressions are required) surrounded by [and]. A *list expression* is a possibly empty and colon-separated list of expressions surrounded by double brackets [[and]]. An *association list expression* is a possibly empty list of pairs of expressions (denoting key and value, respectively) surrounded by braces { and }. A *non-empty list of expressions* ($\langle$ne-exprlist$\rangle$) is a comma-separated non-empty list of expressions.

Parsing of expressions constructed with infix operators is governed by priorities and associativity rules listed in Section 2.5.

Note that, due to the fact that tuple expressions have at least two component expressions, an expression like `[[e]]` denotes a list with one single element denoted by the expression e (and *not* a hypothetical one-tuple containing the one-tuple e).

Context-Free Syntax

$$
\begin{array}{rcl}
\langle\text{expr}\rangle & \rightarrow & \langle\text{integer}\rangle \mid \langle\text{ident}\rangle \\
& \mid & \langle\text{ident}\rangle \; (\; \langle\text{ne-exprlist}\rangle \;) \\
& \mid & \langle\text{expression}\rangle \;\; \langle\text{infix-operator}\rangle \;\; \langle\text{expression}\rangle \\
& \mid & (\; \langle\text{expression}\rangle \;) \\
& \mid & [\; \langle\text{expression}\rangle \; , \; \langle\text{ne-exprlist}\rangle \;] \\
& \mid & [[\;]] \mid [[\; \langle\text{ne-exprlist}\rangle \;]] \\
& \mid & \{ \; \} \mid \{ (\; \langle\text{expression}\rangle \; : \; \langle\text{expression}\rangle \; ,) * \; \langle\text{expression}\rangle \; : \; \langle\text{expression}\rangle \; \} \\
\langle\text{ne-exprlist}\rangle & \rightarrow & (\; \langle\text{expression}\rangle \; ,) * \; \langle\text{expression}\rangle
\end{array}
$$

Static Semantics Checks

- The expression must be typable taking into consideration the default signature (Section 2.6), the signature part of the protocol specification (Section 2.3.1), and the current environment (variable bindings).
- All variables used in an expression must be defined by the current environment (the restriction on roles and scenarios guarantees that this can be checked statically - see Section 2.3.3 and 2.3.5).

Indications on the Semantics

Some indications on the semantics of operators are given in Section 2.6.

2.2.5 Patterns

Patterns are used for filtering values. They are used in recv instructions where they serve to specify the form of a message, in match instructions, and to define the different cases in a case instruction (see Section 2.3.3).

A *pattern* (⟨pattern⟩) consists of an expression and an optional type expression. A *pattern sequence* (⟨patterns⟩) is a possibly empty sequence of patterns which are terminated by semi-colons.

Context-Free Syntax

$$
\begin{array}{rcl}
\langle\text{pattern}\rangle & \rightarrow & \langle\text{expression}\rangle \mid (\; \langle\text{expression}\rangle \; : \; \langle\text{typeexp}\rangle \;) \\
\langle\text{patterns}\rangle & \rightarrow & (\; \langle\text{pattern}\rangle \; ;) * \; \langle\text{pattern}\rangle
\end{array}
$$

Static Semantics Checks

Every pattern in a pattern sequence must have the same set of non-instantiated variables (the restriction on roles and scenarios guarantees that this can be verified statically).

Indications on the Semantics

- The variables of a pattern which are not instantiated by the current variable binding are instantiated by a successful pattern matching. The resulting binding (that is, the matching substitution) may not be uniquely defined for certain equational theories. In this case the matching operation is non-deterministic.
- All *mutable* variables occurring in a pattern must be instantiated.
- The key argument of a crypt operation must always be instantiated.
- The construction $(exp : type)$ serves to check the type of an expression (this is useful for checking the type of a value obtained in a recv instruction).
- A pattern sequence is applied sequentially from left to right. The matching substitution resulting from a pattern sequence is the one returned from the left-most pattern in the sequence that matches.

2.2.6 L-Expressions

L-expression are only used in scenarios (Section 2.3.5). Contrary to expressions which denote values, an l-expression denotes a memory location. An l-expression may occur in a scenario on the left-hand side of an assignment, or as argument of a new instruction.

An *l-expression* (⟨l-expression⟩) is either an identifier, or an identifier applied to a non-empty list of expressions.

Context-Free Syntax

$$\langle\text{l-expression}\rangle \quad \rightarrow \quad \langle\text{ident}\rangle \; (\; (\; \langle\text{ne-exprlist}\rangle \;) \;)?$$

Static Semantics Checks

The identifier must have been declared in the global variable section of the protocol specification (see Section 2.3.4), and must be used with arguments corresponding to its type.

2.3 The Five Main Sections of a Protocol Specification

A specification consists of an optional signature which, when given, extends the default signature (see Chapter 2.6), an optional list of equational axioms, a list of role specifications, an optional declaration of global variables, and a scenario. The scope of the global variables extends over the scenario but not over the role definitions. In fact, access to global variables from inside the roles is only possible via parameters of roles.

$$\langle\text{specification}\rangle \quad \rightarrow \quad \langle\text{signature}\rangle\,? \;\; \langle\text{axioms}\rangle\,? \;\; \langle\text{role}\rangle * \;\; \langle\text{globals}\rangle\,? \;\; \langle\text{scenario}\rangle$$

2.3.1 Signature

This part is optional in a protocol specification. Its purpose is to extend the default signature (Section 2.6) by user-defined function symbols.

A *signature* ($\langle$signature$\rangle$) consists of two simple signatures. A *simple signature* ($\langle$simple-signature$\rangle$) is a list of symbol declarations with their type. A single symbol declaration may define several symbols at once which are given by a comma-separated list.

Context-free Grammar

$$\begin{array}{rcl} \langle\text{signature}\rangle & \rightarrow & \underline{\text{signature}}\ \langle\text{simple-signature}\rangle\ (\ \underline{\text{hashfunctions}}\ \langle\text{simple-signature}\rangle\ )?\ \underline{\text{end}} \\ \langle\text{simple-signature}\rangle & \rightarrow & (\ \langle\text{ne-identlist}\rangle\ :\ \langle\text{typeexp}\rangle\ ;\ )* \end{array}$$

Static Semantics Checks

Pre-defined functions (see Section 2.6.2) must not be re-defined in the signature, and no function symbol must be defined twice. The rationale of this restriction is that it guarantees any well-typed expression to have exactly one minimal type[1].

Indications on the Semantics

The first simple signature contains the list of *decomposable* function symbols, while the second signature contains the list of hash function symbols which are not decomposable.

The *complete signature* of a protocol specification consists of the union of the default signature and the user-defined signature.

2.3.2 Axioms

This part is optional in a protocol specification. Its purpose is to give a semantics to the user-defined symbols (Section 2.3.1) through equational axioms.

An *axiom listing* ($\langle$axioms$\rangle$) consists of a list of equalities between expressions, preceded by an optional list of variable declarations.

Context-Free Syntax

$$\langle\text{axioms}\rangle \rightarrow \underline{\text{axioms}}\ (\ \underline{\text{declare}}\ \langle\text{decllist}\rangle\ \underline{\text{begin}}\ )?\ (\ \langle\text{expr}\rangle = \langle\text{expr}\rangle\ ;\ )*\ \underline{\text{end}}$$

[1]This restriction may in the future be replaced by a weaker one if necessary

Static Semantics Checks

- All variables used in the axioms must be declared (hence, the declaration part may only be dropped when all axioms are ground).
- All variable declarations must be logical.
- The two sides of an equational axiom must be of the same type.

Indications on the Semantics

The equational axioms define a *class* of algebras in which the expressions are evaluated. Roughly, this class consists of all algebras that conservatively extend the algebra corresponding to the default signature and that are models of the equational axioms. An algebra A is a *conservative extension* of the standard algebra if

1. any two terms over the standard signature which are evaluated to different values in the standard algebra are also evaluated to different terms in A ("no confusion" - no new equalities between standard terms).
2. For every sort, the carrier set of A is the same as the carrier set of the standard algebra ("no junk" - no new values).

This class may be empty in case of an inconsistency in the equational axioms which makes the standard algebra collapse, as for instance for the equational axiom $0 = 1$. This class may also contain more than one algebra, for instance if we declare a new constant symbol of type `int` without giving any equation definining its value.

2.3.3 Roles

The purpose of this section is to define the different roles which are executed by the protocol participants..

A *role definition* ($\langle$role$\rangle$) consists of the keyword <u>role</u> followed by the name of the role, a non-terminated declaration list surrounded by parentheses which serves to specify the list of parameters, and an instruction block. An *instruction block* ($\langle$block$\rangle$) consists of an optional variable declaration (which, if present, consists of the keyword <u>declare</u> followed by a declaration list) and an instruction list surrounded by <u>begin</u> ... <u>end</u> .

An *instruction list* ($\langle$instlist$\rangle$) is a possibly empty list of instructions that are terminated by a semi-colon. An *instruction* is a *raw instruction* ($\langle$raw-instruction$\rangle$) which may be preceded by an identifier which, in this case, is separated from the following raw instruction by a colon.

A *raw instruction* ($\langle$raw-instruction$\rangle$) is either an instruction block, a *send* instruction, a *receive* instruction, a <u>fail</u> instruction, an *assignment instruction*, a *new* instruction, a *match* instruction, or a composite instruction. A composite instruction is either a conditional instruction <u>if</u> ... <u>then</u> ... <u>else</u> ... <u>fi</u> where the *else* part is optional, a *choice* instruction which consists of a non-empty sequence of instructions lists which are separated by the symbol | and surrounded by the keywords <u>choice</u> and <u>end</u> , or a case instruction. A *case instruction* consists, besides the keywords, of an expression, a non-empty list of cases which are separated by semi-colons, and a final instruction list. A *case* is a pair consisting of a pattern sequence and an instruction list, joined by the symbol `->`.

Context-Free Syntax

$$\langle \text{role} \rangle \;\rightarrow\; \underline{\text{role}}\ \langle \text{ident} \rangle\ (\ \langle \text{nt-decllist} \rangle\ ?)\ \langle \text{block} \rangle$$

$$\begin{array}{rcl}
\langle \text{block} \rangle & \rightarrow & (\ \underline{\text{declare}}\ \langle \text{nt-decllist} \rangle\ ;\ )?\ \underline{\text{begin}}\ \langle \text{instlist} \rangle\ \underline{\text{end}} \\
\langle \text{instruction} \rangle & \rightarrow & \langle \text{raw-instruction} \rangle \mid \langle \text{ident} \rangle\ :\ \langle \text{raw-instruction} \rangle \\
\langle \text{raw-instruction} \rangle & \rightarrow & \langle \text{block} \rangle \\
& \mid & \underline{\text{send}}\ (\ \langle \text{expression} \rangle\ ) \mid \underline{\text{recv}}\ (\ \langle \text{patterns} \rangle\ ) \\
& \mid & \underline{\text{fail}} \\
& \mid & \langle \text{ident} \rangle\ :=\ \langle \text{expression} \rangle \\
& \mid & \underline{\text{new}}\ (\ \langle \text{ident} \rangle\ ) \\
& \mid & \underline{\text{match}}\ \langle \text{expression} \rangle\ \underline{\text{with}}\ \langle \text{patterns} \rangle\ \underline{\text{end}} \\
& \mid & \underline{\text{if}}\ \langle \text{expression} \rangle\ \underline{\text{then}}\ \langle \text{instlist} \rangle\ (\ \underline{\text{else}}\ \langle \text{instlist} \rangle\ )?\ \underline{\text{fi}} \\
& \mid & \underline{\text{choice}}\ \langle \text{instlist} \rangle\ (\mid \langle \text{instlist} \rangle\ )*\ \underline{\text{end}} \\
& \mid & \underline{\text{case}}\ \langle \text{expression} \rangle\ \underline{\text{of}}\ (\ \langle \text{case} \rangle\ ;\ )*\ \langle \text{case} \rangle\ \underline{\text{else}}\ \langle \text{instlist} \rangle\ \underline{\text{esac}} \\
\langle \text{case} \rangle & \rightarrow & \langle \text{patterns} \rangle\ -\!>\ \langle \text{instlist} \rangle \\
\langle \text{instlist} \rangle & \rightarrow & (\ \langle \text{instruction} \rangle\ ;\ )*
\end{array}$$

Static Semantics Checks

- The name of a role must be distinct from the names of the pre-defined type constants. This is necessary since in the logics the names of roles are use as types of process variables. All roles must carry different names.
- The same name of a variable may occur in several variable declarations in a role definition. The usual scoping roles apply.
- Parameters of roles are call-by-value, except when declared as mutable in which case they are call-by-reference.
- The identifiers occurring as prefixes of instructions must be unique in each role definition. Their purpose is to define control points which are referenced in the logics (see Section 4).
- In case of a branching instruction (like case) all branches must instantiate the same variables.
- The argument to new must be of one of the types *nonce*, *int*, or *message*.

Indications on the Semantics

- A recv instruction never fails. It blocks until a value is received that is matched by one of the patterns.
- The different cases of a case instruction are tried in sequential order. If none of the pattern sequences matches then the final instruction (given after the keyword else) fires.

2.3.4 Declaration of Global Variables

This part is optional in role specifications. Its purpose is to declare the variables which are used in the scenario. These variables may also take the form of multi-dimensional arrays.

Context-Free Syntax

$$
\begin{array}{rcl}
\langle \text{globals} \rangle & \rightarrow & \underline{\text{variables}}\ \langle \text{skolem-decllist} \rangle\ \underline{\text{end}} \\
\langle \text{skolem-declaration} \rangle & \rightarrow & \langle \text{skolem-identlist} \rangle \mid \langle \text{skolem-identlist} \rangle\ :\ \langle \text{typeexp} \rangle \\
 & \mid & \langle \text{skolem-identlist} \rangle\ :\ \underline{\text{mutable}}\ \ \langle \text{typeexp} \rangle\ ? \\
\langle \text{skolem-decllist} \rangle & \rightarrow & (\ \langle \text{skolem-declaration} \rangle\ ;\ )* \\
\langle \text{skolem-identlist} \rangle & \rightarrow & (\ \langle \text{skolem-ident} \rangle\ ,\ )*\ \langle \text{skolem- ident} \rangle \\
\langle \text{skolem-ident} \rangle & \rightarrow & \langle \text{ident} \rangle\ (\ \langle \text{ne-identlist} \rangle\ )
\end{array}
$$

Static Semantics Checks

- All type expressions in a declaration must be arrow-free and variable-free.
- No variable may be declared twice in a declaration list.

Indications on the Semantics

The explanation given in Section 2.2 on logical and mutable variables apply.

A declaration like

```
variables
  a(int,principal) : bool
end
```

can be seen as the declaration of a two-dimensional array of Boolean values, indexed by pairs of integers and principals. Another way of seeing this kind of variable declaration is saying that we explicitly declare Skolem functions. For example, a quantification structure of a first-order logic formula like $\forall x_1 \exists y_1 \forall x_2 \exists y_2 \ldots \phi$ could be expressed in our framework by declaring the two variables y_1 and y_2 as global variables (assuming that the domain of discourse is the set of integers):

```
variables
  y1(int)     : int;
  y2(int,int) : int;
end
```

Our form of global variable definition allows to define dependencies of variables which cannot be obtained through Skolemisation of first-order formulas, like for instance

```
variables
  y1(int) : int;
  y2(int) : int;
end
```

This corresponds to the so-called *Härtig-quantifier* which is not expressible in first-order logic [Ebb85].

2.3.5 Scenario

The purpose of the scenario is to define how the role instances are instantiated, and how they interact through global variables.

A *scenario* (⟨scenario⟩) consists of a scene instruction (⟨sceneinst⟩) surrounded by the keywords scenario and end . A *scene instruction* is a raw scene instruction, possibly prefixed by an identifier. A *raw scene instruction* is one of a role instantiation, a call to new or makekey , an assignment (where the left-hand side may be any l-expression, contrary to assignment in roles where the left-hand side must be an identifier), a conditional expression (with an optional else part), an infinite loop forever the body of which is a scenario list, a parallel instruction, the body of which is a non-empty list of scenario lists which are separated by the character |, an existential quantification consisting of a value declaration and a scenario, a universal declaration which has an analogous form as an existential quantification but may carry in addition a relational constraint, and a block which consists of a local declaration part and a scenario list.

A *scenario list* is a possibly empty list of scenarios which are terminated by semi-colons. A *simple value declaration* is a pair of a non-empty list of identifiers and a type expression, and a *value declaration* is a non-empty list of simple value declarations which are separated by commas. A *relational constraint* (⟨relation-constraint⟩) is a non-empty sequence of intervals which are separated by commas. An *interval* is either an omega-number or a pair of omega numbers. An *omega number* is either a positive integer or the symbol $*$.

$$
\begin{array}{rcl}
\langle\text{scenario}\rangle & \rightarrow & \underline{\text{scenario}}\ \langle\text{sceneinst}\rangle\ \underline{\text{end}} \\
\langle\text{sceneinst}\rangle & \rightarrow & \langle\text{raw-sceneinst}\rangle \mid \langle\text{ident}\rangle\ :\ \langle\text{raw-sceneinst}\rangle \\
\langle\text{raw-sceneinst}\rangle & \rightarrow & \langle\text{ident}\rangle\ (\ \langle\text{exprlist}\rangle\ ) \\
& \mid & \langle\text{l-expression}\rangle\ :=\ \langle\text{expression}\rangle \\
& \mid & \underline{\text{new}}\ (\ \langle\text{l-expression}\rangle\ ) \\
& \mid & \underline{\text{makekey}}\ (\ \langle\text{expression}\rangle\ ,\ \langle\text{l-expression}\rangle\ ) \\
& \mid & \underline{\text{if}}\ \langle\text{expression}\rangle\ \underline{\text{then}}\ \langle\text{scenelist}\rangle\ (\ \underline{\text{else}}\ \langle\text{scenelist}\rangle\ )?\ \underline{\text{fi}} \\
& \mid & \underline{\text{forever}}\ \langle\text{scenelist}\rangle\ \underline{\text{end}} \\
& \mid & \underline{\text{parallel}}\ (\ \langle\text{sceneinst}\rangle\ |\ )*\ \langle\text{sceneinst}\rangle\ \underline{\text{end}} \\
& \mid & \underline{\text{begin}}\ \langle\text{scenelist}\rangle\ \underline{\text{end}} \\
& \mid & \underline{\text{exists}}\ \langle\text{valuedec}\rangle\ .\ \langle\text{sceneinst}\rangle \\
& \mid & \underline{\text{forall}}\ \langle\text{valuedec}\rangle\ (\ \langle\text{relation-constraint}\rangle\ )?.\ \langle\text{sceneinst}\rangle \\
& \mid & \underline{\text{declare}}\ \langle\text{nt-decllist}\rangle\ ;\ \underline{\text{begin}}\ \langle\text{scenelist}\rangle\ \underline{\text{end}} \\
\langle\text{valuedec}\rangle & \rightarrow & \langle\text{ne-identlist}\rangle\ :\ \langle\text{typeexp}\rangle\ (\ ,\ \langle\text{ne-identlist}\rangle\ :\ \langle\text{typeexp}\rangle\ )?
\end{array}
$$

$$
\begin{aligned}
\langle\text{scenelist}\rangle &\rightarrow (\ \langle\text{sceneinst}\rangle\ ;\ )* \\
\langle\text{relation-constraint}\rangle &\rightarrow [\ \langle\text{interval}\rangle\ (\ ,\ \langle\text{interval}\rangle\ )*] \\
\langle\text{interval}\rangle &\rightarrow \langle\text{omega-number}\rangle \mid \langle\text{omega-number}\rangle\ ..\ \langle\text{omega-number}\rangle \\
\langle\text{omega-number}\rangle &\rightarrow \langle\text{positive-integer}\rangle \mid *
\end{aligned}
$$

Scenarios may be prefixed with a label which serves to identify control points in a scenario.

Static Semantics Checks

- The same name of a variable may occur in several variable declarations in a scenario. The usual scoping roles apply.
- The identifiers occurring as prefixes of instructions must be unique in the scenario. Their purpose is to define control points which are referenced in the logics (see Section 4).
- The argument to new must be of one of the types *nonce*, *int*, or *message*.
- In case of a branching instruction (like case) all branches must instantiate the same variables.
- Roles must be invoked according to their declaration.

Indications on the Semantics

- The first case in the definition of a scenario serves for invoking roles.
- The key word forever indicates an infinite sequential loop.
- The key words exists and forall allow to quantify over values of a type.
- The meaning of the relational constraint is the following: A quantified expression of the form

  $$\underline{\text{forall}}\ x_1 : t_1, \ldots, x_n : t_n\ \textit{constraint}\ .\ S$$

  should be understood as

  $$\exists R :\ \textit{constraint}\ \forall x_1 : t_1, \ldots, x_n : t_n . R(x_1, \ldots, x_n) \Rightarrow S$$

  where we say that a n-ary relation R satisfies a constraint $[l_1..u_1, \ldots, l_n..u_n]$ iff

  $$\bigwedge_{i=1}^{n} \forall x_1 : t_1, \ldots, x_{i-1} : t_{i-1}, x_{i+1} : t_{i+1}, \ldots, x_n : t_n . l_i \leq \#\{(x_1, \ldots, x_n) \mid R(x_1, \ldots, x_n)\} \leq u_i$$

  We allow to abbreviate an interval $n..n$ as n, and $*$ stands for infinity. Hence, in case of binary relations (n=2), $[1,1]$ denotes a relation which is a bijective function, $[1..*,1]$ a function which is onto, and $[0..1,1]$ a function which is one-to-one. If no constraint is given then the default is $[0..*, \ldots, 0..*]$, which is satisfied by any relation.

2.4 Lexical Tokens

A *number* is any character between '0' and '9'. A *letter* is any character between 'a' and 'z', either in lower case or in upper case. An *identifier* is a non-empty sequence of numbers, letters, and the special character '_' that starts on a letter. An *integer* is a non-empty sequence of numbers (a negative integer constant is considered as an expression composed of the prefix operator '-' and a non-negative integer constant).

$$
\begin{array}{rcl}
\langle\text{number}\rangle & \rightarrow & \underline{0} \mid \ldots \mid \underline{9} \\
\langle\text{letter}\rangle & \rightarrow & \underline{a} \mid \ldots \mid \underline{z} \mid \underline{A} \mid \ldots \mid \underline{Z} \\
\langle\text{ident}\rangle & \rightarrow & \langle\text{letter}\rangle\ (\ \langle\text{letter}\rangle \mid \langle\text{number}\rangle \mid \underline{\_}\ )* \\
\langle\text{integer}\rangle & \rightarrow & \langle\text{number}\rangle\ +
\end{array}
$$

2.5 Infix Operators

All infix operators associate to the left. Following is a table of infix operators in increasing order of precedence (that is, operators at the top of the table are stronger binding than operators at the bottom of the table).

`.`
`+`, `-`, `::`
`=`, `<`, `>`, `=<`, `>=`, `!=`
`&&`
`|`

For instance, the expression

```
x + y - z && b
```

is parsed in the same way as

```
((x + y) - z) && b
```

2.6 The Default Signature

2.6.1 Pre-Defined Types

Following is the list of pre-defined type constants. *message* is the universal type comprising all non-functional types.

Name	Comment
`message`	the type of all (non-functional) values
`int`	the type of integers
`bool`	the type of boolean values
`nonce`	the type of nonces
`algo`	the enumeration type of asymmetric encryption algorithms
`symalgo`	the enumeration type of symmetric encryption algorithms
`pubkey`	the type of public encryption keys for a non-specified asymmetric encryption method
`privkey`	the type of private decryption keys for a non-specified asymmetric encryption method
`symkey`	the type of en/decryption keys for a non-specified symmetric encryption method
`principal`	the type of principals

Furthermore, we have the following type constructors:

Name	Arity	Comment
`list`	1	polymorphic lists
`table`	2	polymorphic association lists

2.6.2 Pre-Defined Constants and Functions

Bool

The domain of the type `bool` consists of the two values *true* and *false*. The following operations yield boolean values:

Name	Type	Comment
`true`, `false`,	`bool`	Constant values
`&&`	(`bool`, `bool`) → `bool`	Conjunction (infix operator)
`||`	(`bool`, `bool`) → `bool`	Disjunction (infix operator)
`not`	`bool` → `bool`	Negation
=, <, >, =<, >=, `!=`	(`'a`, `'a`) → `bool`	Polymorphic comparison operators

Int

Numerical constants, binary operators +, -.

Algo and Symalgo

These are the enumeration types for different encryption and signature algorithms. The following constants are defined:

Name	Type	Comment
`asym`	`algo`	A not further specified asymmetric encryption method
`rsa`	`algo`	Rivest-Shamir-Adleman method
`sym`	`symalgo`	A not further specified symmetric encryption method
`des`	`symalgo`	Data Encryption Standard

Keys

The instruction `makekey(a, var)`, where *a* is instantiated and either of type *algo* or of type *symalgo*, assigns to *var* a new key.

If the type of *a* is *symalgo* then *var* must be of type *symkey*.

If the type of *a* is *algo* then *var* must be of type *privkey*. That is, in case of asymmetric encryption the variable *var* will then hold the *private* part of the key pair. The *public* part of the key pair can be obtained as $inv(key)$. Static analysis will ensure that *inv* can only be applied when a makekey operation was executed on the same *var*. The `inv` operation is an exception in that its type is overloaded.

Name	Type
`sign`	`(algo,message,privkey)` → `message`
`crypt`	`(algo,message,pubkey)` → `message`
`symcrypt`	`(symalgo,message,symkey)` → `message`
`inv`	`pubkey` → `privkey` `privkey` → `pubkey`

Nonce

There are no constants of this type, or functions that yield these values (a nonce is created by new). The only test operation on nonces is equality test (explicit, or implicit in pattern matching).

Lists

$[[]]$ is the empty list

$[[a_1, a_2, a_3]]$ is the list consisting of the three values a_1, a_2, a_3.

$x :: y$ yields the list with head x and rest y.

$element(x, y)$ tests whether x is contained in the list y.

Lists are monomorphic, that is all elements must be of the same type. Note that this type may be the universal type *message*. Hence, $[[1, true]]$ is of type *list(message)*.

Tuples

The set of types is closed under tupling. For instance, $[a_1, a_2, a_3]$ is the triple consisting of the three values a_1, a_2, a_3. Tuples always contain at least two elements, that is there are no zero-tuples or one-tuples. As a consequence, the expression $[[42]]$ denotes a list of length 1, and not a one-tuple of a one-tuple.

There are no operations defined on tuples. The only way to access the components of a tuple is to use pattern-matching.

Association Lists

Association lists are written in the form

$$\{key_1 : value_1, key_2 : value_2, \ldots, key_n : value_n\}$$

If l is an association list then $l.x$ yields the value of l for key x.

Chapter 3

Semantics

In this chapter we define the semantics of the cryptographic protocol language of the project PROUVÉ.

The semantics is described in a modular way, in that the semantics for scenarios is based on some semantics for roles that is given by a transition system on role configurations, but without assuming any particular properties of the semantics of roles. Also, the semantics of roles is defined while assuming a model of an intruder and a class of algebras that are the semantics of the equational axioms without making any particular assumption on them.

Other noticeable choices that have been made are the following.

Scopes There are several choices that can be made to reflect and take care of the scope of variables. The choice made in our semantic definition is to keep two levels of scoping. We flatten the scopes introduced at scenarios level but keep the scopes at role level. Variable renaming and α-conversion are needed to maintain coherence of variable bindings.

Lazy evaluation of the forall and iteration commands The forall operator (written $\forall$ in our abstract syntax) can be unfolded at once or in a lazy manner introducing one new scenario in parallel at a time. The same remark applies to iteration. The lazy interpretation is chosen in our semantics. Concerning the forall operator, the unfolded scenario is chosen non-deterministically.

Initial values and variable initialization A particularity of the communication in cryptographic protocols is that the reception of a message may assign values to some variables while the values of the remaining variables are used for synchronization. In fact, variables that have not yet been assigned a value are of the first category while the other variables are of the second. So, one has to keep track in a way or another which variables have already a value assigned to them. There are different ways how to keep this information also depending upon how one wants to handle initial values.

The choice made in our semantics is that variables not yet affected by a receive or an assignment are not in the domain of the functions that associate values to variables.

$$
\begin{array}{rcl}
S & ::= & R(x_1,\ldots,x_n,e_1,\cdots,e_m) \\
& | & \mathsf{makekey}(algo,x) \\
& | & x := e \\
& | & \mathsf{new}(x) \\
& | & S_1;\ldots;S_n \\
& | & \textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi} \\
& | & S^* \\
& | & S_1\|\ldots\|S_n \\
& | & \exists x:\tau.S \\
& | & \forall x:\tau, I.S, \text{ where } I \text{ is a relational constraint} \\
& | & \text{Var } x:\tau.S
\end{array}
$$

Figure 3.1: Scenarios

3.1 Preliminaries

Given a mapping $f : A \to B$, $a \in A$ and $b \in B$, we denote by $f[b/a]$ the mapping that to any a' in the domain of f with $a' \neq a$ assign the value $f(a')$; and which assigns b to a.

Given mappings $f : A \to B$ and $g : A \to B$ with disjoint domains, $f \oplus g$ denotes the mapping that associates $g(a')$ to a' in the domain of g and $f(a')$ to a' in the domain of f.

For a given algebra $\mathcal{A}$ over the complete signature of the protocol specification we denote by $=_{\mathcal{A}}$ and $\approx_{\mathcal{A}}$ the equality, respectively unification, in $\mathcal{A}$.

3.2 Scenarios

We consider scenarios in a simplified abstract syntax as defined by the grammar in Fig 3.1. We assume for the sake of simplification of notation that the parameter list of any role consists of a sequence of call-by-reference parameters, followed by a sequence of call-by-value parameters. We may assume that every conditional instruction has an else part (if missing it can be taken to be the empty instruction list). As explained above, we assume a flattened block structure, and hence do not have to cope with begin ... end and with declare ... begin ... end .

3.2.1 Configuration

In the following, we will suppose w.l.o.g. that bound variables have distinct names.

The current state of a scenarios is described by a pair (σ, Conf) where σ is a mapping associating values to variables and Conf is a configuration. Configurations reflect the control part. Details for configurations are given in Figure 3.2. Each role R has defined configurations C_R and an initial configuration C_R^i. The value of variable x_M is going to be the set of messages sent during execution. Given a state (σ, Conf), the value of variable x_M as well as of all variables but those defined in roles are given

$$
\begin{array}{rcll}
\text{Conf} & ::= & (\text{Role}(p, C_R), & \text{where } p \in \mathbb{P} \text{ is a process identifier} \\
& | & \text{Conf}_1 \,\|\ldots\|\, \text{Conf}_n & \\
& | & \text{Conf}; S & \\
& | & \forall x : \tau, I.S & \\
& | & \forall x : \tau, R, U.S, & \text{where } U \text{ is a finite subset of } R \\
& | & S & \\
& | & \varepsilon &
\end{array}
$$

Figure 3.2: Configurations

by σ. Hence, in particular, the values of global variables are defined in the state. Hence the initial state of a scenarios S is $([x_M \mapsto E], S)$, where E is a set of messages initially known by the intruder.

Configuration Transition System

We assume given a transition system for each role R. This is essentially given by a transition relation $\longrightarrow_R$ between pairs (σ, Conf). Moreover, we assume given an interpretation function which associates to each expression e and variable assignment σ a value $[\![e]\!]\sigma$.

The transition relation associated to scenarios is inductively defined by the following inferences, where S_α is the α conversion (i.e. renaming) of S using fresh names for variables bound in S.

-
$$\frac{(\sigma, C_R^i[y_1/x_1, \cdots, y_n/x_n, v_1/z_1, \cdots, v_m/z_m]) \longrightarrow_R (\sigma', C_R),\ [\![e_i]\!]\sigma =_{\mathcal{A}} v_i\ (i = 1, \cdots, m)}{(\sigma, R(y_1, \ldots, y_n)) \longrightarrow (\sigma', \text{Role}(p, C_R))} \quad [\text{ROLE INIT}]$$

  where $y_1, \cdots, y_n, e_1, \cdots, e_m$ are the actual parameters and $x_1, \cdots, x_n, z_1, \cdots, z_m$ the formal ones, and where $p \in \mathbb{P}$ is a process identifier (the INTERLEAVING role given below will ensure that processes have unique process identifiers).

  Notice that since we assumed that local and global variables have different names there is no risk of name clash. Also, notice that we have call-by-name semantics.

-
$$\frac{(\sigma, C_R) \longrightarrow_R (\sigma', C_R')}{(\sigma, \text{Role}(p, C_R)) \rightarrow (\sigma', \text{Role}(p, C_R'))} \quad [\text{ROLE STEP}]$$

-
$$\frac{(\sigma, \text{Conf}) \longrightarrow (\sigma', \text{Conf}')}{(\sigma, \text{Conf}; S) \rightarrow (\sigma', \text{Conf}'; S)} \quad [\text{SCENARIOS SEQUENTIAL COMPOSITION}]$$

-
$$\frac{(\sigma, \text{Conf}) \longrightarrow (\sigma', \varepsilon)}{(\sigma, \text{Conf}; S) \rightarrow (\sigma', S)} \quad [\text{SCENARIOS SEQUENTIAL COMPOSITION}]$$

- $$\frac{(\sigma, \mathrm{Conf}_i) \longrightarrow (\sigma', \mathrm{Conf}'_i)}{(\sigma, \mathrm{Conf}_1 \| \ldots \| \mathrm{Conf}_i \| \ldots \| \mathrm{Conf}_n) \longrightarrow (\sigma', \mathrm{Conf}_1 \| \ldots \| \mathrm{Conf}'_i \| \ldots \| \mathrm{Conf}_n)} \quad [\textsc{Interleaving}]$$
  where for all $i \neq j$ no process identifier occurs both in Conf_i and Conf_j.

- $$\frac{[\![e]\!]\sigma =_{\mathcal{A}} v}{(\sigma, x := e) \longrightarrow (\sigma[v/x], \varepsilon)} \quad [\textsc{Assignment}]$$

- The instruction $\mathsf{new}(x)$ associates a fresh value to the variable x. To define the semantics of this instruction, we assume that we are given a function D such that D_x is the value domain of x and that we are given a function *new* that when applied to a domain D_x yields a fresh value $new(D_x)$. Strictly, speaking this is not a function as the value generated depends on the previously generated values. But the idea is that this function can be seen as an oracle that has a memory.
  $$(\sigma, \mathsf{new}(x)) \rightarrow (\sigma \oplus [x \mapsto new(D_x)], \varepsilon) \qquad [\textsc{New}]$$

- The case of $\mathsf{makekey}(algo, x)$ is treated similarly to $\mathsf{new}(x)$. That is, we assume that we are given an oracle $\mathsf{newkey}(\cdot)$ such that $\mathsf{newkey}(algo)$ generates a fresh key compatible with *algo*, i.e., an RSA key in case *algo* is RSA, a DES key in case it is DES etc.... . Hence, semantics of $\mathsf{makekey}(algo, x)$ is given by
  $$(\sigma, \mathsf{makekey}(algo, x))) \rightarrow (\sigma \oplus [x \mapsto \mathsf{newkey}(algo)], \varepsilon)$$

- $$\frac{[\![e]\!]\sigma =_{\mathcal{A}} \top}{(\sigma, \textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi }) \longrightarrow (\sigma, S_1)} \quad [\textsc{If-Then}]$$

- $$\frac{[\![e]\!]\sigma =_{\mathcal{A}} \bot}{(\sigma, \textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi }) \longrightarrow (\sigma, S_2)} \quad [\textsc{If-Then}]$$

- $$\frac{-}{(\sigma, S^*) \longrightarrow (\sigma, S_\alpha; S^*)} \quad [\textsc{Iteration unfolding}]$$

- $$\frac{-}{(\sigma, \forall x : \tau, I.S) \longrightarrow (\sigma, S_\alpha[u/x] \| \forall x : \tau, R, \{u\}.S)} \quad [\forall \textsc{ unfolding}]$$
  where R is a relation whose arity is the length of x and that satisfies the constraint I and $u \in R$.

- $$\frac{u \notin U}{(\sigma, \forall x : \tau, R, U.S) \longrightarrow (\sigma, S_\alpha[u/x] \| \forall x : \tau, x \notin (u :: U).S)} \quad [\forall \textsc{ unfolding}]$$
  where $u \in R \setminus U$.

- $$\frac{u : \tau}{(\sigma, \exists x : \tau.S) \longrightarrow (\sigma, S[u/x])}$$

- $$\frac{}{(\sigma, \mathrm{Var}\ x : \tau.S) \longrightarrow (\sigma, S)}$$

Equations involving ε: $\varepsilon; S = S$ and $\varepsilon \| \mathrm{Conf} = \mathrm{Conf}$ apply.

3.3 Roles

As roles are sequences of actions, their semantics is simply the execution of these actions from left to right. Each action has its intuitive semantics. Consider a role $R(x_1,...,x_n,z_1,\cdots,z_m)$, where the x_i's are reference parameters while the z_i's value parameters. Let also $y_1...y_p$ be its local variables. For simplicity and without loss of generality, we assume that the formal parameters and local parameters are disjoint. The step semantics of role R has the form $(\sigma, C_R) \rightarrow_R (\sigma', C'_R)$. Henceforth, we simply write $\rightarrow$ instead of $\rightarrow_R$. The mapping σ assigns values to global variables (including x_M); while C_R is a triple (R, σ_L, I), R is the role name, σ_L assigns values to the local variables $y_1,\cdots,y_p$ and value parameters $z_1,\cdots,z_m$ and I is the list of instruction still to be executed. The initial configuration of a role C^i_R is any triple $(R,[],I)$, where $[]$ is the empty mapping (the everywhere undefined function) and I is the set of instruction of R.

We consider roles in a simplified abstract syntax defined as follows:

$$
\begin{array}{rcl}
I & ::= & I_1;I_2 \\
 & | & send(e) \\
 & | & recv(r) \\
 & | & x := e \\
 & | & \textbf{if } e \textbf{ match } p \textbf{ then } I_1 \textbf{ else } I_2 \textbf{ fi} \\
 & | & \mathsf{new}(x) \\
 & | & \varepsilon \\
 & | & fail
\end{array}
$$

In this definition, we have ignored labels of instructions since they are not relevant for the definition of the semantics. It is easy to see that any role definition according to Section 2.3.3 can be transformed in this simplified syntax.

The transition relation $(\sigma,(R,\sigma_L,I)) \rightarrow (\sigma',(R,\sigma'_L,I'))$ is defined by induction over I. As the role name R is constant it will be omitted in the following.

- $$\frac{(\sigma,(\sigma_L,I_1)) \rightarrow (\sigma',(\sigma'_L,I'_1))}{(\sigma,(\sigma_L,I_1;I_2)) \rightarrow (\sigma',(\sigma'_L,I'_1;I_2))} \qquad \text{[Sequential composition]}$$

- $$\frac{[\![e]\!](\sigma \oplus \sigma_L) = v, \quad \sigma' = (\sigma \oplus \sigma_L)[\sigma(x_M) \cup \{v\}/x_M]}{(\sigma,(\sigma_L,send(e))) \rightarrow (\sigma',(\sigma_L,\varepsilon))} \qquad \text{[Sending a message]}$$

- When receiving a message, variables that are neither in the domain of σ nor in the domain of σ_L get a value assigned to them. This value is given by any mappings σ' and σ'_L whose domains are disjoint from the domains of σ and σ_L and such that the message $[\![p_i]\!]\sigma''$ with $\sigma'' = \sigma \oplus \sigma_L \oplus \sigma' \oplus \sigma'_L$ is deducible from $\sigma(x_M)$, i.e., $\sigma(x_M) \vdash m$.

  $$\frac{\sigma(x_M) \vdash [\![p_i]\!]\sigma'',\ \forall j < i \cdot p_j \not\approx_{\mathcal{A}} v}{(\sigma,(\sigma_L,recv(p_0,\cdots,p_n))) \rightarrow (\sigma,(\sigma_L \oplus \sigma'_L,\varepsilon))} \qquad \text{[Reception of message]}$$

  where $\sigma'' = \sigma \oplus \sigma_L \oplus \sigma' \oplus \sigma'_L$ and the domains of σ' and σ'_L contain only variables that occur in p_i.

- Depending whether the variable x is a parameter or a local variable, σ or σ_L is modified by affectation $x := e$.

$$\frac{[\![e]\!](\sigma \oplus \sigma_L) = v,\ \sigma'_L = \sigma_L[v/x],\ \sigma' = \sigma[v/x]}{(\sigma, (\sigma_L, x := e)) \to (\sigma', (\sigma'_L, \varepsilon))} \quad \text{[Assignment]}$$

- The **match** instruction is treated in the exact same way as in recv.

$$\frac{[\![e]\!](\sigma \oplus \sigma_L) = v\ [\![p]\!]\sigma'' = v',\ v' =_{\mathcal{A}} v,\ \sigma'' = \sigma \oplus \sigma_L \oplus \sigma' \oplus \sigma'_L}{(\sigma, (\sigma_L, \textbf{if } e \textbf{ match } p \textbf{ then } I_1 \textbf{ else } I_2 \textbf{ fi })) \to (\sigma \oplus \sigma', (\sigma_L \oplus \sigma'_L, I_1))} \quad \text{[Successful match]}$$

where the domains of σ' and σ'_L contain only variables that occur in e.

$$\frac{\text{The premise of Rule [Successful match] is not satisfiable}}{(\sigma, (\sigma_L, \textbf{if } e \textbf{ match } p \textbf{ then } I_1 \textbf{ else } I_2 \textbf{ fi })) \to (\sigma, (\sigma_L, I_2))} \quad \text{[Unsuccessful match]}$$

- $$(\sigma, (\sigma_L, \mathsf{new}(x))) \to (\sigma \oplus [x \mapsto new(D_x)], (\sigma_L \oplus [x \mapsto new(D_x)], \varepsilon))$$

- $$(\sigma, (\sigma_L, \mathsf{makekey}(algo, x))) \to (\sigma \oplus [x \mapsto \mathsf{newkey}(algo)], (\sigma_L \oplus [x \mapsto \mathsf{newkey}(algo)], \varepsilon))$$

Equations involving ε still apply.

3.4 Semantic Domains

In this section we define the class of algebras in which the expressions are evaluated, and in which pattern matching is done. This section uses notions from the field of Algebraic Specifications, see for instance [EM85].

Let S be the set of all closed and arrow-free type expressions, and Σ the complete signature of the protocol specification. We assume in this section that all types in the signature Σ are closed, that is that a declaration of a polymorphic function like for instance $=: ('a,'a) \to \texttt{bool}$ is replaced by the family of closed type declarations $\{=: (\tau, \tau) \to \texttt{bool} \mid \tau \in T\}$.

A multi-sorted algebra $\mathcal{A}$ over the set of sorts S and signature Σ is called *admissible* if the following conditions hold:

- The carrier set of sort `int` is $\mathcal{A}^{\texttt{int}} = \mathbb{Z}$.
- The carrier set of sort `bool` is $A^{\texttt{bool}} = \{\texttt{true}, \texttt{false}\}$.
- The carrier sets of the sorts `nonce`, `algo`, `symalgo`, `pubkey`, `privkey`, `symkey`, and `principal` are of infinite cardinality.
- For every type $\tau \in T$ there exists an injective function $\iota_\tau : \mathcal{A}^\tau \to \mathcal{A}^{\texttt{message}}$.
- For every type $\tau \in T$, the carrier set of type $\texttt{list}(\tau)$ is the set of finite sequences of the carrier set of type τ :

$$\mathcal{A}^{\texttt{list}(\tau)} = (\mathcal{A}^\tau)^*$$

- For every $\tau, \sigma \in T$, the carrier set of type `table` (τ, σ) is the set of finite sequences of pairs of values from the carrier set of τ and the carrier set of σ :

$$\mathcal{A}^{\mathtt{table}(\tau,\sigma)} = (\mathcal{A}^{\tau} \times \mathcal{A}^{\sigma})^{*}$$

- For every $\tau_1, \tau_2, \ldots, \tau_n \in T$ the carrier set of type $(\tau_1, \tau_2, \ldots, \tau_n)$ is the set of tuples of elements of the carrier sets of types $\tau_1, \tau_2, \ldots, \tau_n$.

$$\mathcal{A}^{(\tau_1, \tau_2, \ldots, \tau_n)} = \mathcal{A}^{\tau_1} \times \mathcal{A}^{\tau_2} \times \ldots \times \mathcal{A}^{\tau_n}$$

- The operations on integers ($+$, $*$, numerical constants, and the comparison operators of type $(\mathtt{int}, \mathtt{int}) \rightarrow \mathtt{bool}$) and the operations on Boolean values (`true`, `false`, `&&`, `||`, `not`, and the comparison operators of type $(\mathtt{bool}, \mathtt{bool}) \rightarrow \mathtt{bool}$) are interpreted in $\mathcal{A}$ as usual.

- For all $x \in \mathcal{A}^{\mathtt{privkey}} \cup \mathcal{A}^{\mathtt{pubkey}}$ we have that $\mathtt{inv}^{\mathcal{A}}(\mathtt{inv}^{\mathcal{A}}(x)) = x$.

- For any type τ, the operator $=$ of type $(\tau, \tau) \rightarrow \mathtt{bool}$ is interpreted as equality on A^{τ}, the operator $<$ of type $(\tau, \tau) \rightarrow \mathtt{bool}$ is a strict total order on A^{τ}, and the following equivalences hold:

$$\begin{aligned} x > y &\Leftrightarrow y < x \\ x =< y &\Leftrightarrow x = y \vee x < y \\ x >= y &\Leftrightarrow y =< x \\ x != y &\Leftrightarrow \neg x = y \end{aligned}$$

Given a set Ax of equational axioms for a given signature Σ, the *semantics of Ax* is the set of all admissible Σ algebras that are models of the equational axioms Ax.

Chapter 4

A Logic for Traces

In this section we present a logic which can express assertions about traces. We assume given a specification of the form

$$Spec = \langle signature, axioms, roles, variables, scenario \rangle$$

The signature Σ contains all the constants and function symbols available in the specification, that is those explicitly mentioned in the `signature` section of the specification, and the default symbols as described in Chapter 2.6.

4.1 Syntax

We use the following syntactic categories:

- P, P_1, P_2, ... denote variables for processes. These variables are intended to denote unique process identifiers.
- R, R_1, R_2, ... denote roles defined by the specification.
- c denotes a control point defined by the specification, either in a role or in the scenario. Every role and the scenario are assumed to define control points $\underline{\text{start}}$ and $\underline{\text{end}}$.
- x denotes a top-level variable of a role (that is either a parameter, or a variable defined at the beginning of the role), or a global variable of the scenario (that is, a variable defined in the `variables` section of the specification).
- f denotes a function symbol of the signature Σ.
- a denotes a constant of the signature Σ
- X denotes a variable of the logic.
- T denotes a type.

The set of terms is given as

$$t \quad ::= \quad a \mid f(t_1, \ldots, t_n) \mid x[t_1, \ldots, t_n] \mid P.x \mid X$$

A variable x without prefix denotes a global variable of the scenario, while a prefixed variable $P.x$ denotes the variable x of the process P. Variables of processes are available in the logics only when they are top-level variables, that is parameters of a role or variables declared in the declaration part of the block (but not variables which are defined in local scopii). Variables with a local scope are a mere convenience for the programmer; they are not visible in the logics.

Now, the set of formulas is defined as

$$\begin{array}{rcl}
\phi & ::= & t_1 = t_2 \\
& \mid & t_1 \leq t_2 \\
& \mid & P_1 = P_2 \\
& \mid & \underline{\text{secret}}\ (t) \\
& \mid & P@c \mid \underline{\text{scenario}}\ @c \\
& \mid & \phi_1 \wedge \phi_2 \mid \neg\phi \\
& \mid & \boxminus\phi \\
& \mid & \forall P \lhd R : \phi \\
& \mid & \forall X \in T : \phi
\end{array}$$

Process variables are always typed by a role, that is the type of a process variable indicates of which role the process is an instance.

We employ the usual abbreviations from first-order logic:

$$\begin{array}{rcl}
t_1 \neq t_2 & := & \neg t_1 = t_2 \\
\phi_1 \vee \phi_2 & := & \neg(\neg\phi_1 \wedge \neg\phi_2) \\
\phi_1 \rightarrow \phi_2 & := & (\neg\phi_1) \vee \phi_2 \\
\phi_1 \leftrightarrow \phi_2 & := & (\phi_1 \rightarrow \phi_2) \wedge (\phi_2 \rightarrow \phi_1) \\
\exists P \lhd R : \phi & := & \neg\forall P \lhd R : \neg\phi \\
\exists X \in T : \phi & := & \neg\forall X \in T : \neg\phi \\
\exists\, !P \lhd R : \phi & := & \exists P \lhd R : \phi \wedge \forall P, Q \lhd R : (\phi \wedge \phi[Q/P] \rightarrow P = Q)
\end{array}$$

where $\phi[Q/P]$ denotes the variant of ϕ obtained by replacing all free occurrence of P by Q, probably renaming bound variables in order to avoid capture. The formula $\exists\, !P \lhd R : \phi$ means "there is a unique P of role R such that ϕ".

4.2 Semantics

We assume an infinite domain $\mathbb{P}$ of process identifiers (for instance the set of natural numbers).

We recall from Section 3.3 that a *process configuration* is a triple (R, σ, I) where

- R is a name of a role defined by the specification
- σ is an assignment of the top-level variables of role r (that is, the parameters of the role, and the variables declared at the beginning of the role)
- I is the list of instructions that are still to be executed by the process.

We denote by PC the set of process configurations. Given a process configuration $pconf$ we write $pconf_{name}$ for its name, $pconf_{env}$ for its variable binding, and $pconf_{label}$ for the label of the next instruction to be executed (or `end` if the list of instructions I is empty).

We recall from Section 3.2 that a *global state* is a pair (σ, Conf) where

- σ is a partial mapping from global variables to values
- Conf is a scenario configuration of the form defined in Figure 3.2.

Every configuration which can be reached from the initial configuration of a protocol specification has, modulo associativity and commutativity of the operator $||$, the form

$$\mathrm{Conf}_{s_1} || \ldots || \mathrm{Conf}_{s_m} || (p_1, \mathrm{Role}(C_{R_1})) || \ldots || (p_n, \mathrm{Role}(C_{R_n}))$$

where the $\mathrm{Conf}_{s_i} \neq \varepsilon$ are expressions of the abstract syntax of scenarios and where $p_i \neq p_j$ for $i \neq j$. Given a global state *gstate* with a configuration in the form as written above, we write $gconf_{labels}$ for the set of labels of $\mathrm{Conf}_{s_1}, \ldots, \mathrm{Conf}_{s_m}$, $gconf_{env}$ ($gconf_{labels} = \{\mathtt{end}\}$ in case $m = 0$) for its variable binding, and $gconf_{processes}$ for its so-called *process table* which is defined as the mapping associating C_{R_i} to p_i.

A *trace* is a non-empty and finite sequence of global configurations. The set of traces Tr is hence

$$Tr = GC^+$$

First we define $[[t]]_{\mathcal{A},\gamma,\alpha,\pi}$, which is the evaluation of term t in algebra $\mathcal{A}$, global configuration γ, assignment α of the logical variables, and assignment π of the process variables:

$$\begin{array}{rcl}
[[a]]_{\mathcal{A},\gamma,\alpha,\pi} & := & a^{\mathcal{A}} \\
[[f(t_1, \ldots, t_n)]]_{\mathcal{A},\gamma,\alpha,\pi} & := & f^{\mathcal{A}}([[t_1]]_{\mathcal{A},\gamma,\alpha,\pi}, \ldots, [[t_n]]_{\mathcal{A},\gamma,\alpha,\pi}) \\
[[x[t_1, \ldots, t_n]]]_{\mathcal{A},\gamma,\alpha,\pi} & := & \gamma env(x[[[t_1]]_{\mathcal{A},\gamma,\alpha,\pi}, \ldots, [[t_n]]_{\mathcal{A},\gamma,\alpha,\pi}]) \\
[[P.x]]_{\mathcal{A},\gamma,\alpha,\pi} & := & (\gamma processes(\pi(P)))env(x) \\
[[X]]_{\mathcal{A},\gamma,\alpha,\pi} & := & \alpha(X)
\end{array}$$

The value of $[[t]]_{\mathcal{A},\gamma,\alpha,\pi}$ is undefined in case t contains a sub-expression $P.x$, where P is a process variable, such that P is not in the domain of π.

In other words,

- constants a and function symbols f are interpreted in the algebra $\mathcal{A}$,

- in order to evaluate $x(t_1,\ldots,t_n)$, that is the value of a global variable, we first evaluate the index terms t_i, and then look up the value of x at this location according to the environment component of γ,

- in order to evaluate $P.x$ we obtain the process configuration denoted by P, and look up the value of x in the environment component of this process configuration.

- X is a logical variable which obtains its value from α

For a trace τ we denote $\tau^{\diamondsuit}$ its last element. We define, for any Σ-algebra $\mathcal{A}$, for any trace τ, any valuation α of the variables (of the logic), and any partial mapping of process variables to $\mathbb{P}$, what it means that a formula ϕ holds, in signs $\mathcal{A},\tau,\alpha,\pi \models \phi$.

$$
\begin{array}{rcl}
\mathcal{A},\tau,\alpha \models t_1 = t_2 & \Leftrightarrow & [[t_1]]_{\mathcal{A},\tau^{\diamondsuit},\alpha,\pi} = [[t_2]]_{\mathcal{A},\tau^{\diamondsuit},\alpha,\pi} \\
\mathcal{A},\tau,\alpha \models t_1 \leq t_2 & \Leftrightarrow & \text{exists a context } C \text{ with } [[C[t_1]]]_{\mathcal{A},\tau^{\diamondsuit},\alpha,\pi} = [[t_2]]_{\mathcal{A},\tau^{\diamondsuit},\alpha,\pi} \\
\mathcal{A},\tau,\alpha,\pi \models P_1 = P_2 & \Leftrightarrow & \pi(P_1) = \pi(P_2) \\
\mathcal{A},\tau,\alpha,\pi \models \underline{\text{secret}}\ (t) & \Leftrightarrow & \tau^{\diamondsuit}_{env}(x_M) \not\vdash [[t]]_{\mathcal{A},\tau^{\diamondsuit},\alpha,\pi} \\
\mathcal{A},\tau,\alpha,\pi \models P@c & \Leftrightarrow & (\tau^{\diamondsuit}_{processes}(\pi(P)))_{label} = c \\
\mathcal{A},\tau,\alpha,\pi \models \underline{\text{scenario}}\ @c & \Leftrightarrow & c \in \tau^{\diamondsuit}_{label} \\
\mathcal{A},\tau,\alpha,\pi \models \phi_1 \wedge \phi_2 & \Leftrightarrow & \mathcal{A},\tau,\alpha,\pi \models \phi_1 \text{ and } \mathcal{A},\tau,\alpha,\pi \models \phi_2 \\
\mathcal{A},\tau,\alpha,\pi \models \neg\phi & \Leftrightarrow & \mathcal{A},\tau,\alpha,\pi \not\models \phi \\
\mathcal{A},\tau,\alpha,\pi \models \boxminus\phi & \Leftrightarrow & \text{for all nonempty prefixes } \tau' \text{ of } \tau\text{: } \mathcal{A},\tau',\alpha,\pi \models \phi \\
\mathcal{A},\tau,\alpha,\pi \models \forall P \lhd R : \phi & \Leftrightarrow & \text{for all } p \in domain(\pi) \text{ with } (\tau^{\diamondsuit}_{processes}(p))name = R\text{: } \mathcal{A},\tau,\alpha,[p/P]\pi \models \phi \\
\mathcal{A},\tau,\alpha,\pi \models \forall X \in T : \phi & \Leftrightarrow & \text{for all } d \in \mathcal{A}^T\text{: } \mathcal{A},\tau,[d/X]\alpha,\pi \models \phi
\end{array}
$$

The truth of $\mathcal{A},\tau,\alpha \models \phi$ is undefined in case evaluation of $\mathcal{A},\tau,\alpha \models \phi$ leads to some $[[t]]_{\mathcal{A},\gamma,\alpha,\pi}$ that is undefined.

In other words,

- $t_1 = t_2$ denotes equality of the values denoted by the two terms

- $t_1 \leq t_2$ denotes that t_1 is a semantic subterm of t_2 (*semantic* since context and term are evaluated in the algebra $\mathcal{A}$)

- $P_1 = P_2$ denotes identity of process identifiers

- $\underline{\text{secret}}\ (t)$ denotes that the intruder cannot deduce t from the current value of x_M

- the operator @ denotes that a process (or the scenario) currently is at a certain control point

- $\wedge$ and $\neg$ have the usual meaning

- $\boxminus$ means "always in the past"

- $\forall P \lhd R$: is a universal quantification over all currently existing processes of type R

- $\forall X \in T$: is a universal quantification over all values of type T

We say that a closed formula ϕ is valid in the specification *Spec* as defined above if

$$\mathcal{A}, \tau, \emptyset, \emptyset \models \phi$$

for all algebras $\mathcal{A}$ permitted by the semantics, and all traces τ, where $\emptyset$ denotes the empty assignment.

Chapter 5

Security Assertions

5.1 A Language for Security Assertions

In this section we define a language of *security assertions*. A security assertion consists of

- a subset of the signature Σ; this subset indicates which function symbols are available to the intruder when constructing messages
- a formula from a restricted fragment of the logic defined in Section 4.

We call a formulas *quantifier-free* if it contains none of $\forall$, $\exists$, $\exists!$, or $\boxminus$.

Formally, an assertion is a triple

$$(\Gamma, \phi_{init}, \phi_{always})$$

Where $\Gamma \subseteq \Sigma$, and $\phi_{init}, \phi_{always}$ are formulas as defined in Section 4 and are of the following form:

1. ϕ_{init} is the universal closure of quantifier-free formula,
2. ϕ_{always} is the universal closure of a formula of the form $\beta \to \gamma$, where β is quantifier-free, and γ is a positive boolean combination of formulas of Σ_1-formulas. A Σ_1-formula is a quantifier-free formula prefixed by a possibly empty sequence of quantifiers $\exists P \triangleleft R :$ and $\exists ! P \triangleleft R :$.

The specification of Γ in an assertion amounts to

$$\bigwedge_{f \in \Gamma} \vdash f \wedge \bigwedge_{f \in \Sigma - \Gamma} \nvdash f$$

while the formula pertaining to the assertion is

$$(\boxminus(\ \underline{\text{scenario}}\ @\ \underline{\text{start}}\ \to \phi_{init})) \to \phi_{always}$$

Due to the syntactic restriction on assertions, nameley the fact that the modality $\boxminus$ must not occur in the scope of a quantifier, truth is well-defined for all assertions.

5.2 Examples

5.2.1 Listing Initial Intruder Knowledge

We can express that "ϕ holds provided the intruder knows initially exactly the messages $m_1, \ldots, m_n$ (and everything he can deduce from this) by

$$\boxminus\Big(\ \underline{\text{scenario}} \ @ \ \underline{\text{start}}\ \rightarrow x_M = [m_1, \ldots, m_n]\Big) \rightarrow \phi$$

Note that $x_M = [m_1, \ldots, m_n]$ is *not* equivalent to $\forall x \in \texttt{message} : (\ \underline{\text{secret}}\ (x) \leftrightarrow (x \neq m_1 \vee \ldots \vee x \neq m_n))$ The reason is that the set of messages which is not secret is always closed under some set of deductions rules.

5.2.2 Negative Assertions about Initial Intruder Knowledge

We can express that "whenever the intruder does initially not know m_1 then he does not know m_2 after execution of the scenario" by

$$\boxminus\Big(\ \underline{\text{scenario}} \ @ \ \underline{\text{start}}\ \rightarrow\ \underline{\text{secret}}\ (m_1)\Big) \wedge \underline{\text{scenario}} \ @ \ \underline{\text{end}}\ \rightarrow\ \underline{\text{secret}}\ (m_2)$$

5.2.3 Conditional Secrecy

Consider the specification of the Needham-Schroeder public-key protocol in Section **??**. Note that in this version we have explicitly stated how a session is initiated: Alice will start a session with a principal x when she receives a request of the form $init\_connection\_with(x)$.

The initial intruder knowledge can be expressed by the following formula ψ:

$$\boxminus\Big(\ \underline{\text{scenario}} \ @ \ \underline{\text{start}}\ \rightarrow \\ x_M = \big[\texttt{alice}, \texttt{bob}, \texttt{intruder}, \texttt{inv}(\texttt{pubkey\_of}(\texttt{intruder})), \texttt{init\_connection\_with}\big]\Big)$$

Hence, the intruder knows his own secret key, and since he knows the function symbol *init_connection_with* he can request instances of Alice to start a session with any principal he knows the name of.

Our security assertion is a conditional secrecy property: nonces generated by instances of role `Bob` remain secret, except when the instance of `Bob` was executing a protocol session with the intruder. This can be expressed as follows:

$$\psi \rightarrow \forall P \triangleleft \texttt{Bob} : P.\texttt{alice\_name} \neq \texttt{intruder} \rightarrow \texttt{secret}(P.\texttt{my\_nonce})$$

5.2.4 Authentication

We show that the different versions of authentication described in [Low97] can be expressed in our logics. We assume a setting where we have the two following role definitions:

- A role `initiator`, with a parameter `myname: principal` indicating the name of the principal executing an instance of this role, and a variable `partner: principal` indicating the name of the principal with whom an instance of this role attempts to communicate.

  This role has a label `end` for the control point at the end of the protocol (indicating termination of the role).

- A role `responder`, with a parameter `myname: principal` indicating the name of the principal executing an instance of this role, and a variable `partner: principal` indicating the name of the principal with whom an instance of the role believes to communicate.

  Let L be the set of all control points of role `responder` that are after the last externally visible action of this role (typically, this means after the last send instruction). Being at some control point in L corresponds to "having run the protocol", as discussed in [Low97].

Aliveness

> We say that a protocol guarantees to an initiator A *aliveness* of another agent B if, whenever A (acting as an initiator) completes a run of the protocol, apparently with responder B, then B has been previously running this protocol. [Low97]

Note that this does not state that B has acted as a responder.

This is expressed as follows:

$$\begin{aligned}\forall A, B \triangleleft \mathtt{principal} : \Big(\quad & (\exists P \triangleleft \mathtt{initiator} : .(P.\mathtt{myname} = A \wedge P.\mathtt{partner} = B \wedge P@\mathtt{end})) \\ & \rightarrow (\exists Q \triangleleft \mathtt{responder} : (Q.\mathtt{myname} = B \wedge \bigvee_{c \in L} Q@c)) \\ & \quad \vee (\exists Q \triangleleft \mathtt{initiator} : (Q.\mathtt{myname} = B \wedge \bigvee_{c \in L} Q@c \Big)\end{aligned}$$

Weak Agreement

> We say that a protocol guarantees to an initiator A *weak agreement* with another agent B if, whenever A (acting as initiator) completes a run of the protocol, apparently with responder B, then B has recently been running the protocol, apparently with A. [Low97]

Note that this does not state that B has acted as a responder.

This is just a variant of the *aliveness* property, obtained by adding a condition on the variable Q.`partner`. However, we cannot express the recentness.

$$\begin{aligned}\forall A, B \triangleleft \mathtt{principal} : \Big(\quad & (\exists P \triangleleft \mathtt{initiator} : (P.\mathtt{myname} = A \wedge P.\mathtt{partner} = B \wedge P@\mathtt{end})) \\ & \rightarrow (\exists Q \triangleleft \mathtt{responder} : (Q.\mathtt{myname} = B \wedge Q.\mathtt{partner} = A \\ & \qquad \wedge \bigvee_{c \in L} Q@c)) \\ & \quad \vee (\exists Q \triangleleft \mathtt{initiator} : (Q.\mathtt{myname} = B \wedge Q.\mathtt{partner} = A \\ & \qquad \wedge \bigvee_{c \in L} Q@c)) \Big)\end{aligned}$$

Non-Injective Agreement

> We say that a protocol guarantees to an initiator A *non-injective agreement* with a responder B on a set of data items ds if, whenever A (acting as an initiator) completes a run of the protocol, apparently with responder B, then B has previously been running the protocol, apparently with A, and B was acting as responder in his run, and the two agents agreed on the data values corresponding to all variables in ds. [Low97]

This is expressed as follows:

$$\begin{aligned}\forall A, B \lhd \texttt{principal}: \quad & \forall P \lhd \texttt{initiator}: (P.\texttt{myname} = A \land P.\texttt{partner} = B \land P@\texttt{end} \\ & \rightarrow \exists Q \lhd \texttt{responder}: (Q.\texttt{myname} = B \land Q.\texttt{partner} = A \\ & \qquad \land \bigvee_{c \in L} Q@c \land \bigwedge_{x \in ds} P.x = Q.x))\end{aligned}$$

Agreement

> We say that a protocol guarantees to an initiator A *agreement* with a responder B on a set of data items ds if, whenever A (acting as an initiator) completes a run of the protocol, apparently with responder B, then B has previously been running the protocol, apparently with A, and B was acting as responder in his run, and the two agents agreed on the data values corresponding to all variables in ds, and each run of A corresponds to a *unique* run of B. [Low97]

This extends the condition of Non-injective agreement by demanding that the correspondence between runs is an injective function.

This is expressed as follows:

$$\begin{aligned}\forall A, B \lhd \texttt{principal}: \quad & \forall P \lhd \texttt{initiator}: (P.\texttt{myname} = A \land P.\texttt{partner} = B \land P@\texttt{end} \\ & \rightarrow \exists\texttt{!}Q \lhd \texttt{responder}: (Q.\texttt{myname} = B \land Q.\texttt{partner} = A \\ & \qquad \land \bigvee_{c \in L} Q@c \land \bigwedge_{x \in ds} P.x = Q.x))\end{aligned}$$

Chapter 6

Examples

6.1 The IKA-1 Protocol

```
signature
        a, b, c: principal;
        g: symkey;                # the base for the generation of the shared key
        m: message;               # the secret message
        exp: (symkey,nonce) → symkey;
end

role A (myname: principal)
        declare
        mynonce: nonce;
        begin
                new(mynonce);
                send([g,exp(g,mynonce)]);
                declare
                x,y: symkey;
                begin
                        recv([x,y]);
                        send(symcrypt(sym,exp(x,mynonce),m));
                end;
        end

role B (myname: principal)
        declare
        mynonce: nonce;
        begin
                new(mynonce);
                declare x,y: symkey;
                begin
                        recv([x,y]);
                        send([exp(x,mynonce),y,exp(y,mynonce)]);
```

```
                end;
        end

role C (myname: principal)
        declare
        mynonce: nonce;
        begin
                new(mynonce);
                declare x,y,z : symkey;
                begin
                        recv([x,y,z]);
                        send([exp(x,mynonce),exp(y,mynonce)]);
                end;
        end

scenario
        parallel A(a) | B(b) | C(c) end
end
```

6.2 The Needham-Schroeder Public Key Protocol

```
# The Needham-Schroeder Protocole with Lowe patch

# From the Spore repository :
#
# A,B,S :       Principal
# Na,Nb :       Nonce
# KPa,KPb,KPs,KSa,KSb,KSs :     Key
# KPa,KSa :     is a key pair
# KPb,KSb :     is a key pair
# KPs,KSs :     is a key pair
#
# 1.    A    →    S    :    A,B
# 2.    S    →    A    :    {KPb, B}KSs
# 3.    A    →    B    :    {Na, A}KPb
# 4.    B    →    S    :    B,A
# 5.    S    →    B    :    {KPa, A}KSs
# 6.    B    →    A    :    {Na, Nb, B}KPa
# 7.    A    →    B    :    {Nb}KPb

signature
        alice, bob, intruder: principal;
        init_connection_with: principal → message;
        pubkey_of: principal → pubkey;
end
```

```
# we assume that Alice initiates a session on receiving a request
# of the form init_connection_with(name)
role Alice (my_name: principal; my_key: privkey;
            server_signkey: pubkey)
declare
        my_nonce, bob_nonce: nonce;
        bob_pubkey: pubkey;
        bob_name: principal;
        begin
                recv(init_connection_with(bob_name));
                send([my_name,bob_name]);
                recv(sign(asym,inv(server_signkey),[bob_pubkey,bob_name]));
                new(my_nonce);
                send(crypt(asym,bob_pubkey,[my_nonce,my_name]));
                recv(crypt(asym,inv(my_key),[my_nonce,bob_nonce,bob_name]));
                send(crypt(asym,bob_pubkey,bob_nonce));
        end

role Bob (my_name: principal; my_key: privkey;
   server_signkey: pubkey)
        declare
        my_nonce, alice_nonce : nonce;
        alice_name: principal;
        alice_pubkey: pubkey;
        begin
                recv(crypt(asym,inv(my_key),[alice_nonce,alice_name]));
                send([my_name,alice_name]);
                recv(sign(asym,inv(server_signkey),[alice_pubkey,alice_name]));
                new(my_nonce);
                send(crypt(asym,alice_pubkey,[alice_nonce,my_nonce,my_name]));
                recv(crypt(asym,inv(my_key),my_nonce));
        end

role Server (my_signkey: privkey)
        declare
        requester, query: principal;
        begin
                recv([requester,query]);
                send(sign(asym,my_signkey,[pubkey_of(query),query]));
        end

role Send_initial_request (name: principal)
        begin
                send(init_connection_with(name));
        end

variables
```

```
        alice_key, bob_key, server_key: privkey;
end

scenario
        begin
                makekey(asym, server_key );
                Send_initial_request(bob);
                parallel
                        Alice  (alice,alice_key, inv(server_key))
                        | Bob  (bob,  bob_key,   inv(server_key))
                        | forever Server(server_key) end
                end
        end
end
```

6.3 The TMN Protocol

```
# key distribution protocol by Tatebayashi, Matsuzaki, and Newman
# see for instancs CACM Nov 1994, page 58

signature
        xor: (message,message) → message;
        null: message;
        mult: (message,message) → message;
        one: message;
        inverse: message → message;
end

axioms
        declare
        bx, by, bz, x, y, z;
        key: pubkey;
        begin
                # the xor theory
                xor(bx,by) = xor(by,bx);
                xor(bx,xor(by,bz)) = xor(xor(bx,by),bz);
                xor(bx,null) = bx;
                xor(bx,bx) = null;

                # multiplication
                mult(x,y) = mult(y,x);
                mult(x,mult(y,z)) = mult(mult(x,y),z);
                mult(x,one) = x;
                mult(x,inverse(x)) = one;

                # rsa
```

```
            crypt(rsa,key,mult(x,crypt(rsa,key,y))) = crypt(rsa,key,mult(x,y));
        end

role alice(my_name,bob_name: principal; server_key: pubkey)
        declare
        my_nonce: nonce;
        shared_nonce: message;
        server_message: message;
        begin
            new(my_nonce);
            send([crypt(asym,server_key,my_nonce),my_name,bob_name]);
            recv(server_message);
            shared_nonce ← xor(server_message,my_nonce);
        end

role bob(my_name: principal; server_key: pubkey)
        declare
        shared_nonce: nonce;
        alice_name: principal;
        begin
            recv(alice_name);
            new(shared_nonce);
            send(crypt(asym,server_key,shared_nonce));
        end

role server(seen_keys: mutable list(nonce); my_secretkey: privkey)
        declare
        initiator_nonce, partner_nonce: nonce;
        initiator_name, partner_name: principal;
        begin
            recv([crypt(asym,inv(my_secretkey),initiator_nonce),
      initiator_name,partner_name]);
            if element(initiator_nonce,seen_keys)
            then
                fail;
            else
                seen_keys ← initiator_nonce :: seen_keys;
                send([initiator_name,partner_name]);
                recv(crypt(rsa,inv(my_secretkey),partner_nonce));
                send(xor(initiator_nonce,partner_nonce));
            fi;
        end

variables
        Alice, Bob: principal;
        serverkey: privkey;
        seen_keys: mutable list(nonce);
```

```
end

scenario
        begin
                makekey(rsa, serverkey);
                seen_keys ← [[ ]];
                parallel
                        alice(Alice, Bob, inv(serverkey))
                        | bob(Bob, inv(serverkey))
                        | forever server(seen_keys, serverkey); end
                end;
        end
end
```

6.4 The Electronic Purse Protocol with Symmetric Keys

```
# Electronic Purse version with symmetric keys

signature
        # the function F produces a key, which is to be shared between a purse
        # and a module, from the master key of the module and the name of the purse
        F: (message,message) → message;

        # the function FS takes a symmetric sign key k and list messages l, and
        # returns l signed with k.
        FS: (message, list(message)) → message;

end

# the role of a purse during one transaction
role purse(
   ep_acqkey, ep_isskey;        # the two keys
   myname: principal;           # the name of the purse
   balance: mutable int         # the balance of the purse
   )
        declare
        ep_challenge,sam_challenge: nonce;
        sam_name: message;
        amount: int;
        s2,s3,s6: message;
        begin
                new(ep_challenge);
                send([myname, ep_challenge]);
                recv([sam_name, sam_challenge, s2]);
                if s2 ≠ FS(ep_acqkey, [[myname, ep_challenge]]) then fail; fi;
                recv(amount);
```

```
                balance ← balance − amount;
                s6 ← FS(ep_isskey ,[[sam_name,sam_challenge ,ep_challenge ,amount]]);
                s3 ← FS(ep_acqkey ,[[sam_name,sam_challenge ,ep_challenge ,s6,amount]]);
                send([s3,s6,amount]);
        end


        # the role of an application module during one transaction
role secure_application_module(
   acqkey;                     # the key
   myname: principal;          # the name of the module
   balance: mutable int;       # the balance of the module
        trace: mutable list(message) # audit
   )
        declare
        ep_acqkey: message;
        id_ep, ep_challenge: message;
        s2,s3,s6: message;
        sam_challenge: nonce;
        amount: int;
        begin
                recv([id_ep,ep_challenge]);
                ep_acqkey ← F(acqkey,id_ep);
                s2 ← FS(ep_acqkey,[[id_ep,ep_challenge]]);
                send([myname,sam_challenge,s2]);
                recv([s3,s6,amount]);
                if
                s3 = FS(ep_acqkey,[[myname,sam_challenge,ep_challenge,s6,amount]])
                then
                        balance ← balance + amount;
                        trace ← s6::trace;
                else
                        fail;
                fi;
        end


        # the keyboard where the user types the sum
role keyboard ()
        declare amount: int;
        begin
                new(amount);
                send(amount);
        end

variables
        ep_isskey(principal): int;              # issuer key of a purse
        # alternative : one master issuer key ?
```

```
        acqkey: int;                                  # master key
        ep_balance(principal) : mutable int;          # balance of a purse
        sam_balance(principal): mutable int;          # balance of a module
        trace(principal) : mutable list(message);     # audit trail of a module

end

scenario
        begin

                # compute the master key
                new(acqkey);

                # compute the issuer keys per purse
                forall purse: principal . new(ep_isskey(purse));

                # initialize the traces
                forall module: principal . trace(module) ← [[]];

                # infinitely start parallel sessions between purses and sams, such that
                # at every moment a purse or an agent is engaged in at most one
                # transaction
                forever
                  forall purse,module: principal [ 1,1 ].
                     parallel
                       purse(F(acqkey,purse),
                            ep_isskey(purse), purse, ep_balance(purse))
                     | secure_application_module(acqkey,module,
                            sam_balance(module), trace)
                     | keyboard()
                     end;
                end;  # forever
        end
end       # scenario
```

6.5 The Electronic Purse Protocol with Asymmetric Keys

```
# Electronic Purse version with asymmetric keys

signature
        # the function F produces a key, which is to be shared between a purse
        # and a module, from the master key of the module and the name of the purse
        F: (message,message) → message;

        # the function FS takes a symmetric sign key k and list messages l, and
        # returns l signed with k.
```

```
        FS: (message, list(message)) → message;

        # the hash function
        h: list(message) → message;

        b: int; # the generator of the group

        # standard arithmetic functions
        exp: (int, int) → int;
        mult: (int, int) → int;
        minus: int → int;

        # the signing key of the key creation authorit
        certkey: privkey;

end

axioms
        declare
        x, y, z: int;
        begin
           x + y = y + x;
                x + (y + z) = (x + y) + z;
                minus(x) + x = 0;
                x + 0 = x;
                exp(exp(x,y),z) = exp(x,mult(y,z));
                mult(exp(x,y),exp(x,z)) = exp(x,y+z);
        end

        # the role of a purse during one transaction
role purse(
   ep_acqkey, ep_isskey;       # the two keys
   myname: principal;          # the name of the purse
   balance: mutable int;       # the balance of the purse
   certificat: message;        # the certificate of the purse
   s: int                      # the secret s
   )
        declare
        ep_challenge,sam_challenge: nonce;
        sam_name: message;
        amount, coupon: int;
        s2,t,s6: message;
        p: int; # the public key of the purse
        x: int;
        c: int;
        begin
                p ← exp(b,minus(s));
```

```
                new(ep_challenge);
                send([myname, ep_challenge, p, certificat]);
                recv([sam_name, sam_challenge, s2]);
                if s2 ≠ FS(ep_acqkey, [[myname, ep_challenge]]) then fail; fi;
                recv(amount);
                s6 ← FS(ep_isskey, [[sam_name, sam_challenge, ep_challenge, amount]]);
                new(x);
                coupon ← exp(b, x);
                t ← h([[coupon, sam_name, sam_challenge, ep_challenge, s6, amount]]);
                send(t);
                recv(c);
                balance ← balance − amount;
                send([x + mult(s, c), s6, amount]);
        end


        # the role of an application module during one transaction
role secure_application_module(
   acqkey;                      # the key
   myname: principal;           # the name of the module
   balance: mutable int;        # the balance of the module
   signcheckkey: pubkey;        # the key to check a signature
        trace: mutable list(message) # audit
   )
        declare
        ep_acqkey: message;
        id_ep, ep_challenge: message;
        s2, t, u, s6: message;
        s3: int;
        sam_challenge: nonce;
        amount: int;              # the amount to charge
        p: int;                   # public key of the purse
        c: int;
        begin
                recv([id_ep, ep_challenge, p,
      sign(rsa, inv(signcheckkey), [id_ep, p])]);
                ep_acqkey ← F(acqkey, id_ep);
                s2 ← FS(ep_acqkey, [[id_ep, ep_challenge]]);
                send([myname, sam_challenge, s2]);
                recv(t);
                new(c);
                recv([s3, s6, amount]);
                u ← mult(exp(p, c), exp(b, s3));
                if
                t = h([[u, myname, sam_challenge, ep_challenge, s6, amount]])
                then
                        balance ← balance + amount;
```

```
                        trace ← [s6, sam_challenge, ep_challenge, amount] :: trace;
                else
                        fail;
                fi;
        end

        # the keyboard where the user types the sum
role keyboard ()
        declare amount: int;
        begin
                new(amount);
                send(amount);
        end

variables
        ep_isskey(principal): int;                # iss key of a purse
        secret(principal) :int;                   # secret key used by the purse
        acqkey: int;                              # master key
        ep_balance(principal) : mutable int;      # balance of a purse
        sam_balance(principal): mutable int;      # balance of a module
        trace(principal) : mutable list(message); # audit trail of a module

end

scenario
        begin

                # compute the master key
                new(acqkey);

                # compute the purse keys
                forall purse: principal . begin
                        new(ep_isskey(purse));
                        new(secret(purse));
                end;

                # initialize the traces
                forall module: principal . trace(module) ← [[ ]];

                # infinitely start parallel sessions between purses and sams, such that
                # at every moment a purse or an agent is engaged in at most one
                # transaction
                forever
                        forall purse, module: principal [ 1,1 ] .
                        parallel
                                purse(F(acqkey,purse),
                                ep_isskey(purse), purse, ep_balance(purse),
```

```
                                        sign(asym, certkey ,[ purse , exp(b, minus( secret(purse )))]) ,
                                        secret(purse ))
                                        | secure_application_module(acqkey , module ,
                                        sam_balance(module) ,
                                        inv ( certkey ) ,
                                        trace (module ))
                                        | keyboard ()
                                end;
                        end;  # forever
                end     # begin
        end     # scenario
```

Bibliography

[Ebb85] H.-D. Ebbinghaus. Extended logics: The general framework. In J. Barwise and S. Feferman, editors, *Model-Theoretic Logics*, Perspectives in Mathematical Logic, chapter 2, pages 25–76. Springer-Verlag, 1985.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification, vol. 1*. EATCS-Monographs on Theoretical Computer Science. Springer-Verlag, 1985.

[JRV00] Florent Jacquemard, Michaël Rusinowitch, and Laurent Vigneron. Compiling and verifying security protocols. In Michel Parigot and Andrei Voronkov, editors, *LPAR*, volume 1955 of *Lecture Notes in Computer Science*, pages 131–160, Reunion Island, France, November 2000. Springer-Verlag.

[Low97] Gavin Lowe. A hierarchy of authentication specification. In *10th Computer Security Foundations Workshop*, pages 31–44, Rockport, MA, USA, June 1997. IEEE Computer Society.

[MD02] Jonathan K. Millen and Grit Denker. CAPSL and MuCAPSL. *Journal of Telecommunications and Information Technology*, 4:16–26, 2002.