Diploma Thesis

A Formalization of an Operational Semantics of Security Protocols

Simon Meier

Information Security
Swiss Federal Institute of Technology (ETH)
CH-8092 Zürich

July 30, 2007

Abstract

As a result of the last twenty years of research on the verification of security protocols, there exists now a range of protocol models, security properties, logics and verification tools. Finding attacks on a flawed protocol can nowadays be done efficiently using tools such as Scyther [12]. However, the highest level of the Common Criteria (ISO 15408) requires a formal *verification* of protocol correctness. Doing this strictly formal is still not possible, because of the lack of strictly formal protocol models with an associated verification technique.

In this diploma thesis, we present a conservative embedding in Isabelle/HOL [20] of the protocol model given by the operational semantics of security protocols proposed in [11]. This formalization is an important milestone towards the strictly formal verification of security protocols. Its benefits are twofold: First, it is an *unambiguous* description of a protocol model, which greatly facilitates communicating results. Second, it enables the development of a protocol verification technique with *machine checkable* proofs. This technique may make *logically sound* use of specialized protocol logics as well as automation provided by tools like Scyther.

Contents

1 Introduction

The research on the verification of security protocols has been evolving during the last twenty years, and now includes a range of protocol models, security properties, logics and verification tools. The goals of these methods are twofold: First, to determine whether a given protocol is correct or not, and second to arrive at a methodology to design correct protocols. If a protocol is flawed, there exist efficient tools to find attacks, such as Scyther [12], OFMC [4], or ProVerif [6]. An attack found by such a tool can easily be checked to see whether it is an attack on the protocol, or whether the attack is false, which might occur, if the tool uses approximation techniques or contains bugs. If a protocol is correct, the same tool can be used to establish correctness. However, here we are dependent on the correct modeling of the protocol in the input language of the tool, and the correctness of the tool algorithm as well as its implementation.

In mission-critical industrial environments, there is a demand for protocols that have been *certified*, i.e., proven correct using formal methods, as required by the highest level of the Common Criteria (ISO 15408). However despite the high stakes, formal verification of security protocols is still very uncommon in industrial environments. In our opinion, the main problem is that, for current verification techniques, the *guarantees* one gets from using them *do not justify the effort*. We see three main reasons for this fact: First, proofs are not machine checkable and as such prone to error. A current example are the proofs in the description of PCL [13], whose problems are discussed in 4.3. Second, the degree of automation is low due to lacking tool support. Third, the foundations these formalism rest on are not strong enough or not justified well enough.

Our ultimate aim is to work towards the certification of protocols. We address the problems mentioned above by combining a strictly formal protocol model based on the operational semantics of security protocols (OSSP) proposed in [11] with the unbounded verification tool Scyther [12]. In previous work [18], its static protocol description language was formalized in Isabelle/HOL. In this work, we formalize its execution model together with the security objective of secrecy and a notion of protocol correctness.

This formalization is an important milestone towards certified security protocols. Its benefits are twofold: First, it is an *unambiguous* description of a semantics of security protocols. This unambiguity not only helps understanding the nature of these special types of distributed algorithms, but also makes communicating results easier. Second, it enables the development security protocol verification technique with *machine checkable* proofs. This technique may make *logically sound* use of automation provided by tools like Scyther as well as specialized logics like PCL [13]. This soundness is achieved by conservatively embedding these extensions into Isabelle/HOL on top of our formalization of the OSSP. Further *formal*

justification for the Dolev-Yao assumptions [14] used in our protocol model can be gained by connecting our work with the formalization of the Dolev-Yao model of Backes, Pfitzmann, and Waidner in Isabelle/HOL [22].

1.1 Outline of this Thesis

The presentation of the developed material proceeds as follows: First, the necessary background is presented in 2. It also contains an explanation of the Isabelle/HOL notation we are using to present all formal material (cf. 2.2.1). All definitions and lemmas presented in this text are directly copied from the Isabelle theories developed in this thesis. These theories also contain all proofs of the presented lemmas and can be downloaded from `http://n.ethz.ch/student/meiersi/projects/fossp/`.

Afterwards our formalization of the OSSP proposed in [11] is presented in chapter 3. The aim of this presentation is to serve as a reference for our formalization. All definitions are given, but only some lemmas are presented and proofs are left out. The missing material can be found in the Isabelle theories.

There are no big differences between the original description and our formalization of the OSSP. However we have made some extensions and simplifications with respect to the original description. These and the problems we found in the original are reported in section 4.1.

We also compare our formalization to the inductive approach to security protocol verification pioneered by Paulson in [21] and extended by Bella in [5]. With respect to their approach, our formalization uses an extended term structure such that we are no longer required to introduce additional axioms to model protocols using public key cryptography or composed keys. We also refine his model by defining a mapping from protocols to inductive sets of traces such that we are able to formally compare different protocols and different execution models (cf. 3.3.4, 3.3.5 and 3.4.1). The relation between the two approaches is described in section 4.2.

Reasoning on the trace level can be tedious, because of its low level of abstraction. One way to get more abstraction is to use a protocol logic. However, a protocol logic is only as strong as the model of its semantics. Providing a formal semantics is therefore crucial for a logic to make it a sound tool for protocol verification. As we explain in section 4.3, the current description of PCL, a protocol logic which was proposed in [13] and has recently received considerable attention, does not provide such a semantics. Thus by using PCL, one risks verifying a *flawed* protocol to be correct, as it was possible in the case of the Needham-Schroeder protocol [17] and the BAN logic [7, 9]. One way to amend this problem is to conservatively embed PCL on top of our formalization of the OSSP. This would also provide tool support for proofs in PCL, because they are not easy to get right as flaws in their examples show.

Formal methods allow us to apply the strength of mathematics to real world problems. However, they are only as strong as their model is. The relation between our model and the

real world is described in section 5.1.1. We discuss the design decisions we made in section 5.1.2 and the known issues of our formalization in section 5.1.3.

As we mentioned in the introduction, our ultimate aim is to work towards the certification of protocols. This thesis is a milestone on the way towards this goal and the contributions this thesis provides are explained in section 5.2. However, as it is only a milestone, there remains much to do. We plan to extend the model (cf. 5.3.1) and simplify verification by providing more abstraction (cf. 5.3.2) and automation (cf. 5.3.3).

Acknowledgments

This thesis would never have been possible without the help of many others. Much credit goes to:

Cas Cremers, my supervisor, whose ability to turn the difficult topic of security protocols into a large playground with many nifty puzzles made writing this thesis a lot of fun. It is a pleasure to profit from his knowledge and without his experienced guidance, this thesis would be *very* different.

Achim Brucker and Burkhart Wolff, for introducing me to the theorem prover Isabelle. This thesis would not have been possible without this fine piece of software. Despite its high quality, this lady has a lot of rough edges and I greatly appreciate the time these two brave men took during my semester thesis about HOL-OCL [8] to teach me how to deal with her properly; because otherwise, I would not have dared to undertake this endeavor.

Prof. David Basin my supervising professor at ETH.

Patrick Schaller my office mate, for making our office a very comfortable place to work and dwell.

Hans Dubach, Studiensekretär of the D-INFK, whose administrative talent made stories about the red tape at ETH seem to me like ancient tales.

This is not just some thesis, but it is my diploma thesis. As such it also represents the end of five years of studying computer science at ETH Zürich. Although computer science is a very interesting subject, life would not have been fun without the great time I spent with other people. Most notably:

Daniel Kuettel, Michael Lang, Silvan Villiger, Robert Carnecky, Tomas Dikk, and Yves Ineichen, for laughing, traveling, climbing, and discussing all the curious things that belong to life; and sometimes even to computer science.

Céline Ramseier, Fabian Hurter, Jan Gubler, Thorsten Meyer, Varja Nikolic, my flatmates, for making our flat share a real home with a great spirit.

Christina and Roberto, my parents, and Jonas and Christof, my brothers, for making us a family I could not imagine better.

Andrea, my girlfriend, for her love and constant support. It is a great honor and a lot of fun to explore life together with you.

Last but not least, I also want to thank you, dear reader, for the interest in my work. Would it not be for you, my effort put into this work would be in vain. If you have any questions or comments, do not hesitate and contact me under `iridcode@gmail.com`.

2 Background

2.1 Security Protocols

A *protocol* is a set of rules that determine the exchange of messages over an *insecure network* between two or more principals. *Security protocols* use cryptographic mechanisms to achieve *security objectives* (e. g., secrecy or authentication) which ensure some form of secure communication. They underly most of our current communication systems, such as secure internet communication, cell phone networks, as well as the communication between credit cards, ATM machines, and banks. For these applications, it is crucial that no malicious party can disturb the intended workings of the protocol, or eavesdrop on something he was not supposed to hear.

We are thus not content to have a security protocol of which one is pretty sure that it is secure. What we want is a *protocol together with a machine-checked proof* of its security, the strongest correctness guarantee currently known to the research community. In order to provide such a proof, we need a strictly formal protocol model on top of which we can develop the proofs for all the security objectives the protocol tries to achieve.

Example: The Needham-Schroeder-Lowe protocol

Message Sequence Charts (MSC) are a common way to informally specify protocols. MSC is an ITU-standardized protocol specification language [16] and we will use it throughout this thesis to illustrate security protocols and attacks.

As a running example, we will use the short version of the Needham-Schroeder-Lowe protocol (NSL) from [17]. Figure 2.1 contains a MSC of the NSL protocol taken from [11]. The initiator i holds her own secret key $sk(i)$ and the public key $pk(r)$ of the responder r. Symmetrically, the responder r possesses his own secret key $sk(r)$ and the public key $pk(i)$ of the initiator i. Encryption of a message m with a key k is denoted as $\{\!|\ m\ |\!\}_k$. The initiator first creates a new nonce ni, denoted by the box, and then sends her name i together with the nonce ni, encrypted with the public key $pk(r)$, to the responder. After receipt of this, the responder generates a new nonce nr and sends it, together with the earlier nonce ni and his name r covered by the public key $pk(i)$ to the initiator. She, in turn, unpacks the message and returns the nonce nr of the responder, encrypted with his public key. Security claims are denoted by hexagons. Both the initiator and the responder claim that both nonces are secret.

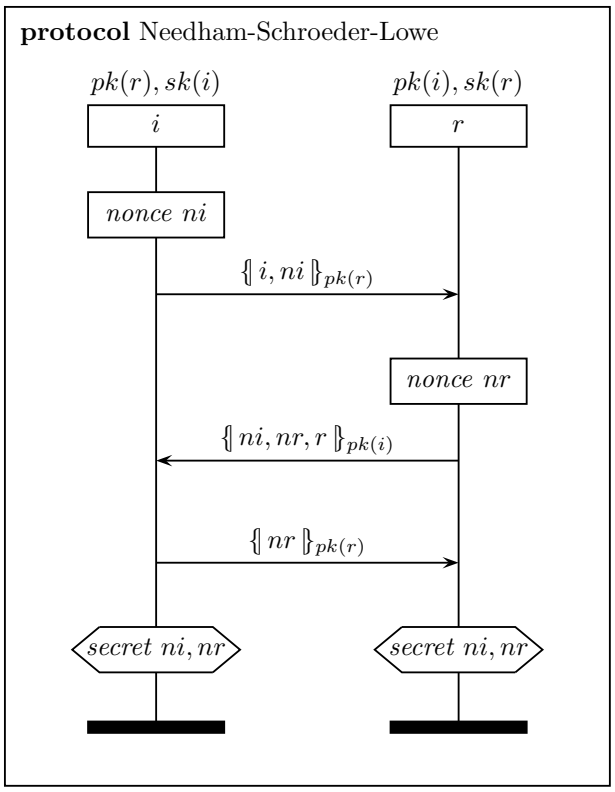

Figure 2.1: The Needham-Schroeder-Lowe public key authentication protocol

2.2 Isabelle/HOL

Isabelle [20] is a generic, LCF-style theorem prover implemented in SML and offers support for checks for conservatism of definitions and powerful generic proof engines based on rewriting and tableaux provers. Furthermore, Isabelle can be extended by user defined SML programs performing symbolic computations over theorems in a logically safe way. While this extensibility is not needed in this work, it will be very important with respect to a future integration of Scyther. Because the automatic check of Scythers certificates will be implemented as an SML program and being able to do this in a logically safe way is crucial to obtain the strong soundness guarantees we are aiming for.

Isabelle/HOL is Isabelles instance for Higher-order logic (HOL) [2] which can roughly be seen as a logic atop of functional programming. It is assumed that the reader has basic familiarity with both logic and typed functional programming. In Isabelle notation, $t :: T$ denotes a term t of type T. This can be used to declare new constants as in **consts** *size* :: *'a btree* $\Rightarrow$ *nat*. The expression **defs** $c \equiv t$ defines the constant c as the term t. A functional constant f can be defined by $f\ x \equiv t$ instead of $f \equiv \lambda x.\ t$. Definitions constitute the principal mechanism for producing conservative extensions of HOL. Constant declaration and definition can also be done in one step using **constdefs**. Type variables are

identified by a leading apostrophe, as in $'a$. Given types $'a$ and $'b$, $'a \Rightarrow 'b$ is the type of (total) functions from $'a$ to $'b$, $'a \times 'b$ is the product type, and $'a\ set$ is the type of sets of elements of type $'a$.

There are several mechanisms to define new types. A **datatype** definition introduces an inductive data type. For example, the option type is defined by **datatype** $'a\ option = None \mid Some\ 'a$, which is polymorphic in the type variable $'a$. This definition also introduces the constructors $None :: 'a\ option$ and $Some :: 'a \Rightarrow 'a\ option$. Pattern matching is used to decompose elements of inductive types. For example, the expression **case** o **of** $None \Rightarrow t \mid Some\ x \Rightarrow f\ x$ evaluates to t if o evaluates to $None$ and to $f\ x$ if o evaluates to $Some\ x$. Functions of type $'a \Rightarrow 'b\ option$ are used to model partial functions from $'a$ to $'b$. The declaration **types** $T1 = T2$ merely introduces a new name for the type $T2$, possibly with parameters, as in **types** $'a \rightharpoonup 'b = 'a \Rightarrow 'b\ option$.

Datatype definitions can also be recursive. For example, a binary tree can be defined by **datatype** $'a\ btree = Leaf\ 'a \mid Node\ ('a\ btree)\ ('a\ btree)$. To define functions over recursive datatypes, primitive recursion is employed as follows:

primrec
$size\ (Leaf\ x) = 0$
$size\ (Node\ l\ r) = size\ l + size\ r$

Note that exactly this definition of *size* which counts the number of recursive constructors is defined automatically by Isabelle for all datatype definitions. It is often used as a measure for Noetherian induction over recursive datatypes.

Isabelle/HOL includes a package supporting inductive sets defined by a set of monotone introduction rules. For example, the set of all binary trees that can be built with the trees in T as "terminations" is defined as follows:

consts $btrees :: 'a\ btree\ set \Rightarrow 'a\ btree\ set$
inductive $btrees\ T$
intros
$Inj: t \in T \Longrightarrow t \in btrees\ T$
$Node: [\![t1 \in btrees\ T;\ t2 \in btrees\ T]\!] \Longrightarrow (Node\ t1\ t2) \in btrees\ T$

Isabelle/HOL includes also a package supporting record types. For example, **record** $point = pointX :: nat\ pointY :: nat$ defines a record type for points, of which the record $(\!|pointX{=}1,\ pointY{=}2|\!)$ is an element. For each field of the record there is a selector function of the same name. The field $pointY$ of a point p is accessed for example as $pointY\ p$. Records also support updates of individual fields like for example in $p(\!|pointX{:=}0|\!)$, which sets the value of the field $pointX$ of the point p to zero.

Note that with respect to our formalization there is not much difference between records and datatypes. Indeed, the decision on where to use which construct was mainly guided by notational brevity. The main difference between records and datatypes is that records support extensionality. But currently, we do not use this feature.

Records, datatypes, primitive recursion, and inductive sets are everything we need from Isabelle/HOL for our formalization of the OSSP.

2.2.1 Mathematical Notation

Isabelle has a rich input language which can be extended with custom syntax when defining new theories. The Isabelle/HOL libraries containing the formalization of mathematical constructs like sets, lists, multisets, maps, and the like make heavy use of this feature to achieve an input language as close to standard mathematical notation as possible, while still being unambiguous. Please note that several arithmetic operators are overloaded. This is the reason for the double occurrence of the operator $-$.

Infinite Sets

$'a\ set$	— type of an infinite set with elements of type $'a$
$x \in A$	— is x an element of the set A
$\{\}$	— empty set
$\{x_1\ ,...,\ x_n\}$	— set of elements x_1 to x_n
$x\ insert\ A$	— set A extended by the element x
$f\ `\ A$	— image of the set A under the function f
$UNIV :: 'a$	— all objects of type $'a$
$range\ f$	— range of the function f, i.e., $f\ `\ UNIV$
$\{x.\ P\ x\}$	— collect all elements x having property P
$\{f\ x \mid x.\ P\ x\}$	— collect and map in one, i.e., $f\ `\ \{x.\ P\ x\}$
$\{x \in A.\ P\ x\}$	— collect all elements $x \in A$ having property P
$A \cup B$	— union of the sets A and B
$A - B$	— set difference subtracting the set A from the set B
$\bigcup S$	— union of the sets in the set of sets S

Finite Lists

$'a\ list$	— type of a finite list with elements of type $'a$
$[]$	— empty list
$(x\#xs)$	— list with head x and tail xs
$[x_1\ ,...,\ x_n]$	— list of elements x_1 to x_n
$xs\ @\ ys$	— append the list ys to xs
$hd\ xs$	— head of the list xs
$tl\ xs$	— tail of the list xs
$distinct\ xs$	— are all elements of the list xs distinct?

Finite Multisets

$'a\ multiset$	— type of a finite multiset with elements of type $'a$
$\{\#\}$	— empty multiset
$\{\#\ x\ \#\}$	— singleton multiset containing x exactly once

$x :\# M$	— is x present in the multiset M ?
$M + N$	— multiset union of the two multisets M and N
$M - N$	— multiset difference subtracting multiset N from multiset M
$set\_of\ M$	— convert the multiset M to its corresponding set

Pairs

$('a \times 'b)$	—type of a pair of an element of type $'a$ and an element of type $'b$
(x, y)	— pair of x and y
$fst\ p$	— first element of a pair
$snd\ p$	— second element of a pair

Relations

Relations are modeled as a set of pairs.

$('a \times 'b)\ set$	— type of a relation between elements of type $'a$ and elements of type $'b$
R^+	— transitive closure of the relation R
$Domain\ R$	— domain of the relation R
$Range\ R$	— range of the relation R

Options

$'a\ option$	— type of an optional value of type $'a$
$o2s\ o$	— change monad representation from option to set $o2s\ (Some\ x) = \{x\}$, $o2s\ (None) = \{\}$
$the\ o$	— extract element, if present $the\ (Some\ x) = x$, $the\ (None)$ is not evaluable
$option\_map\ f\ o$	— apply f to the element in o, if it is present

Maps (Partial Functions)

$'a \rightharpoonup 'b$	— type of a partial function with arguments of type $'a$ mapping to values of type $'b$
$empty$	— empty map
$[x_1 \mapsto a_1, ..., x_n \mapsto a_n]$	— finite map, mapping x_i to a_i
$m(x \mapsto a)$	— update map m at position x with value a
$m_1 \subseteq_m m_2$	— m_1 is a submap of m_2
$dom\ m$	— domain of the map m
$ran\ m$	— range of the map m

Definite Description Operator and Hilbert's Epsilon-operator

Both of these operators are used to identify specific elements in the universe of a type.

THE x. P x — definite description operator

The definite description operator identifies the element having the property P, if *exactly one* such element exists. If there is no such element or if there are more than one of them, an arbitrary element of the correct type is returned. A typical example of this operator is the definition of the if-then-else construct. ("(**if** P **then** x **else** y) ≡ THE z. (P=True ⟶ z=x)∧ (P=False ⟶ z=y)")

SOME x. P x — Hilbert's Epsilon-operator

Hilbert's Epsilon-operator identifies an arbitrary element having property P. If no such element exists, an arbitrary element of the correct type is returned. A typical example of this operator is the definition of the inverse of a function. ("inv (f :: 'a ⇒ 'b) ≡ λy. SOME x. f x = y")

3 Formalization

3.1 Overview

In this chapter we present the formalization of an operational semantics of security protocols (OSSP) developed in this thesis and implemented as a conservative theory extension of Isabelle/HOL. This formalization is an *unambiguous* description of an OSSP derived from the semantics proposed in [11]. The intent of this chapter is to serve as a reference for this description. Therefore all definitions constituting the OSSP are given and explained. Furthermore, some properties of these definitions are highlighted by exhibiting the most important lemmas. This is done by showing and explaining the relevant snippets of the Isabelle theory files. Thus this chapter is quite technical and may not be suitable for every reader. For a high-level overview and more verbose descriptions of the concepts, we refer to chapters 2 and 3 of the original description in [11].

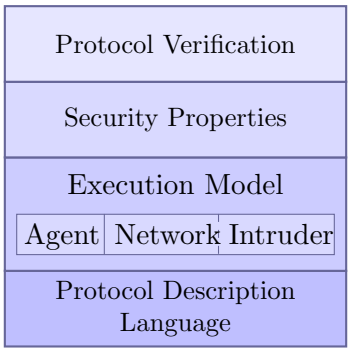

Figure 3.1: Components needed for the verification of security protocols.

The presentation will proceed according to the layers presented in figure 3.1. In 3.2, the protocol description language is explained. Then in 3.3 the operational semantics are given. This is done by first explaining the agents execution model in 3.3.2, then explaining the model for the network and the intruder in 3.3.3, and finally presenting the mapping from a protocol to the traces representing its execution in 3.3.4. We are using an execution model which is parametrized over the type flaw model and the intruder to be used. The consequences of changing them are explained in 3.3.4 and 3.3.5. A static over-approximation to the intruder knowledge which simplifies proofs about the intruder not knowing some term is presented in 3.3.6. Afterwards security properties are explained in 3.4. Finally in 3.5, it is shown what it means for a protocol to be correct.

As a running example we will use the Needham-Schroeder-Lowe protocol (see figure 2.1) with the security objective of nonce secrecy.

3.2 Protocol Description Language

This section describes the protocol description language of our formalization of the OSSP. Most of it was already formalized as part of the semester thesis described in [18]. However, its description has been rewritten in a hopefully more concise and readable style. Also there have been made some extensions to the theories, which render the description given in [18] outdated.

The ultimate goal of our protocol description language is to define a type *'a proto* which designates a protocol. A protocol essentially consists of a set of role specifications. They in turn essentially consist of a list of events or actions that have to be executed by an agent executing a role of the protocol. The two most important events are sending and reading a message. The content of these messages is just a term. We call these terms roleterms, because they occur only in role specifications.

During the execution of a protocol a different sort of terms is used. They are called runterms and are built such that nonces and variables from different executions of a protocol role are different by construction. Runterms are generated by instantiating[1] the roleterms given in the role specifiction. When an agent receives a message, i. e., executes a read event, he matches it against the message pattern he is expecting. All variables in the pattern that have not yet been instantiated are assigned parts of the message, such that the pattern equals the received message. In order to not only be able to specify *what* the expected message has to look like, but also *what type* of a message part is expected, a type system is introduced.

Obviously the definition of roleterms depends on this type system. This is the reason why it is introduced, before defining the rest of the constituents of a protocol. However, one does not have to fully understand it to understand the rest of this chapter, because only the definition of matching (3.3.1) and the investigation of the relation between different typeflaw models (3.3.4) depend on it. It can therefore be safely skipped on the first reading.

3.2.1 Type System

The type of a term essentially reflects the structure of the its constructor applications. The only exception is the type *UserT 'a*, which allows having user defined types for constants. The polymorphic type variable *'a* specifies the type of the identifiers.

datatype *'a type* =
  UserT 'a
| *ConstT* | *VarT* | *AgentT* | *FuncT*

[1] See 3.3.1 for an example.

| *PairT* "'a type" "'a type"
| *EncT* "'a type" "'a type"
| *AppT* "'a type"

The influence of the types given in the protocol specification during matching is controlled by a *type conformance relation*. It answers the question, if a term t of type ϕ can be assigned to a variable v of type ψ. This would be the case, if $(\phi,\ \psi)$ was in the type conformance relation.

types *'a conformance* = "('a type × 'a type) set"

By specifying different type conformance relations, it is possible to investigate *type flaw attacks* [15]. The most simple relation which allows for absolutely no type flaws is the *reflective conformance*. An assignment is only possible, if the types are equal.

constdefs
reflConf :: "'a conformance"
— Types conform only to themselves
reflConf_def : "reflConf ≡ {(t, t) | t. True}"

The reflective conformance does not allow for variables which are able to receive any value. However the intent of the variable type (*VarT*) is to designate a variable that can be assigned *any* term. This is for example used to model model tickets. Therefore we extend the reflective conformance with all pairs of any type and the variable type.

constdefs
noTypeFlaws :: "'a conformance"
— Types conform to themselves and to VarT
noTypeFlaws_def : "noTypeFlaws ≡ reflConf ∪ {(t, VarT) | t. True}"

A more lenient conformance relation is the following. It differs only between pairs, encryptions, and literals (i. e., everything except pairs and encryptions). Thus basic type flaws are possible.

constdefs
basicTypeFlaws :: "'a conformance"
— Basic type flaw attacks possible
basicTypeFlaws_def : "basicTypeFlaws ≡ noTypeFlaws ∪
{(EncT t1 t2, EncT t1' t2') | t1 t2 t1' t2'. True} ∪
{(PairT t1 t2, PairT t1' t2') | t1 t2 t1' t2'. True} ∪
{(t, t') | t t'. ∀ t1 t2. t ≠ PairT t1 t2 ∧ t ≠ EncT t1 t2 ∧
t' ≠ PairT t1 t2 ∧ t' ≠ EncT t1 t2}"

In order to make all type flaw attacks possible, all types have to conform to each other.

constdefs
fullTypeFlaws :: "'a conformance"
— All type flaw attacks possible
fullTypeFlaws_def : "fullTypeFlaws ≡ UNIV"

Verifying a protocol under the *fullTypeFlaws* type conformance relation gives the strongest guarantees with respect to type flaw attacks. See 3.3.4 for a formal justification of this statement.

3.2.2 Static Message Terms

Messages in the specification of a role of a protocol are described with *roleterms*. Since roleterms are used in the static protocol description, all of its constructors are prefixed with an 'S'. The polymorphic type variable *'a* specifies the type of the identifiers.

datatype *'a roleterm* =
  SVar 'a "'a type"
| *SConst 'a* "'a option"
| *SRole 'a*
| *SFunc 'a*
| *SApp 'a* "'a roleterm"
| *SPair* "'a roleterm" "'a roleterm"
| *SEnc* "'a roleterm" "'a roleterm"

As *literals* we have variables, nonces (*SConst*), roles, and hash functions (*SFunc*). As *composed terms* there are hash function applications (*SApp*), pairs, and encryptions of an arbitrary roleterm as the message with another arbitrary roleterm as the key. Pairs can also be written as $\langle A, B, NA \rangle$ which corresponds to (*SPair A* (*SPair B NA*)). Although the above notation suggests it, pairing is *not associative*.

The Type of a Roleterm

Every roleterm is assigned a unique type. This is the reason for the second argument of the variable and constant constructors. While variables can have an arbitrary type, the type of constants is either *ConstT* or *UserT name*. The computation of the type of a roleterm is defined as follows.

consts
  getTypeS :: "'a roleterm ⇒ 'a type"
    — Compute the type of a roleterm
primrec
  "getTypeS (SVar v ty) = ty"
  "getTypeS (SConst c uty) = (**case** uty **of**
                                Some ty ⇒UserT ty
                              | None   ⇒ ConstT)"
  "getTypeS (SRole r) = AgentT"
  "getTypeS (SFunc f) = FuncT"
  "getTypeS (SApp f arg) = AppT (getTypeS arg)"
  "getTypeS (SPair x y) = PairT (getTypeS x) (getTypeS y)"
  "getTypeS (SEnc x k) = EncT (getTypeS x) (getTypeS k)"

Key Inversion Functions

There exists only one encryption operator in our model. This operator is used to model signing, asymmetric encryption, and symmetric encryption. All roleterms except hash function applications are regarded as symmetric keys. To specify which hash function is mapped to which hash function upon key inversion, a map from identifiers to identifiers is used. This map is what we call a *key inversion function* and every protocol can specify its own.

types *'a inverse* = "'a ⇀ 'a"

The *key inversion* is parametrized over the key inversion function (*kif*). The syntax defined such that $kif(\!|k|\!)_s$ denotes the inverse key of k under the key inversion function *kif*.

constdefs
invertKeyS :: "['a inverse, 'a roleterm] ⇒ 'a roleterm" ("_(|_|)$_s$")
— Compute the inverse of a key under a given keying scheme
invertKeyS_def :
"invertKeyS kif key ≡ **case** key **of**
(SApp f arg) ⇒ (**case** kif f **of**
Some f' ⇒ SApp f' arg
| None ⇒ SApp f arg)
| (_) ⇒ key"

The typical public key scheme with private keys being denoted by *sk* and public keys by *pk* can be modeled as the key inversion function

$$[\text{''}sk\text{''} \mapsto \text{''}pk\text{''},\ \text{''}pk\text{''} \mapsto \text{''}sk\text{''}]$$

Inverting the public key of role r amounts then to computing

$$([\text{''}sk\text{''} \mapsto \text{''}pk\text{''},\ \text{''}pk\text{''} \mapsto \text{''}sk\text{''}])\ (\!|SApp\ \text{''}pk\text{''}\ (SRole\ \text{''}r\text{''})|\!)_s$$
$$= SApp\ \text{''}sk\text{''}\ (SRole\ \text{''}r\text{''})$$

A symmetric key shared between roles A and B could be modeled as

$$SApp\ \text{''}shared\text{''}\ \langle SRole\ A, SRole\ B\rangle$$

Inverting it with the above key inversion function gives the expected result

$$([\text{''}sk\text{''} \mapsto \text{''}pk\text{''},\ \text{''}pk\text{''} \mapsto \text{''}sk\text{''}])\ (\!|SApp\ \text{''}shared\text{''}\ \langle SRole\ A, SRole\ B\rangle|\!)_s$$
$$= SApp\ \text{''}shared\text{''}\ \langle SRole\ A, SRole\ B\rangle$$

Relations and Functions on Roleterms

There are various relations and functions on roleterms which are needed to describe the behavior of our model. The following three functions decompose roleterms into their constituents:

consts *subtermsS* :: "'a roleterm set ⇒ 'a roleterm set"
inductive "subtermsS M"
intros
Inj [*intro*]: "x ∈ M ⟹ x ∈ subtermsS M"
Fst: "SPair x y ∈ subtermsS M ⟹ x ∈ subtermsS M"
Snd: "SPair x y ∈ subtermsS M ⟹ y ∈ subtermsS M"
Msg: "SEnc x k ∈ subtermsS M ⟹ x ∈ subtermsS M"
Key: "SEnc x k ∈ subtermsS M ⟹ k ∈ subtermsS M"
App: "SApp n arg ∈ subtermsS M ⟹ arg ∈ subtermsS M"

consts *partsS* :: "'a roleterm set ⇒ 'a roleterm set"
inductive "partsS M"
intros
Inj [*intro*]: "x ∈ M ⟹ x ∈ partsS M"
Fst: "SPair x y ∈ partsS M ⟹ x ∈ partsS M"
Snd: "SPair x y ∈ partsS M ⟹ y ∈ partsS M"
Msg: "SEnc x k ∈ partsS M ⟹ x ∈ partsS M"

consts *analzS* :: "['a inverse, 'a roleterm set] ⇒ 'a roleterm set"
inductive "analzS kif M"
intros
Inj: "x ∈ M ⟹ x ∈ analzS kif M"
Fst: "⟨x,y⟩ ∈ analzS kif M ⟹ x ∈ analzS kif M"
Snd: "⟨x,y⟩ ∈ analzS kif M ⟹ y ∈ analzS kif M"
Decrypt: "⟦ SEnc x k ∈ analzS kif M; kif(|k|)$_s$ ∈ synthS(analzS kif M) ⟧
⟹ x ∈ analzS kif M"

They are related by the following subset relation:

analzS kif M ⊆ *partsS M* ⊆ *subtermsS M*

The set *subtermsS M* contains all subterms of the terms in *M*. The set *partsS M* contains all subterms except keys of encryptions and arguments of hash function applications. The set *analz kif M* also excludes messages of encryptions whose inverse key is not inferable from *M*.

consts *synthS* :: "'a roleterm set ⇒ 'a roleterm set"
inductive "synthS M"
intros
Inj [*intro*]: "x ∈ M ⟹ x ∈ synthS M"
Pair [*intro*]: "⟦ x ∈ synthS M; y ∈ synthS M ⟧⟹ ⟨x,y⟩ ∈ synthS M"

Enc [*intro*]: "⟦ x ∈ synthS M; k ∈ synthS M ⟧⟹ SEnc x k ∈ synthS M"
App [*intro*]: "⟦ SFunc n ∈ M; arg ∈ synthS M ⟧⟹ SApp n arg ∈ synthS M"

The function *synthS* describes which terms can be built from a set of terms. The operations employed to build new terms are pairing, encryption, and hashing with known hash functions.

consts *inferS* :: "['a inverse, 'a roleterm set] ⇒ 'a roleterm set"
inductive "inferS kif M"
intros
Inj: "x ∈ M ⟹ x ∈ inferS kif M"
Fst: "⟨x,y⟩ ∈ inferS kif M ⟹ x ∈ inferS kif M"
Snd: "⟨x,y⟩ ∈ inferS kif M ⟹ y ∈ inferS kif M"
Decrypt: "⟦ SEnc x k ∈ inferS kif M; kif(|k|)$_s$ ∈ inferS kif M ⟧
⟹ x ∈ inferS kif M"
Pair: "⟦ x ∈ inferS kif M; y ∈ inferS kif M ⟧
⟹ ⟨x,y⟩ ∈ inferS kif M"
Enc: "⟦ x ∈ inferS kif M; k ∈ inferS kif M ⟧
⟹ SEnc x k ∈ inferS kif M"
App: "⟦ SFunc f ∈ inferS kif M; arg ∈ inferS kif M ⟧
⟹ SApp f arg ∈ inferS kif M"

The function *inferS* describes which terms can be inferred from a set of terms. This is defined as an arbitrary interleaving of decomposing terms like *analzS*, as well as composing terms like *synthS*. Despite the interleaving *inferS* can be factored in terms of *synthS* and *analzS* as *inferS kif M* = *synthS* (*analzS kif M*).

consts
varsOfS :: "'a roleterm ⇒ 'a set"
primrec
"varsOfS (SVar v ty) = {v}"
"varsOfS (SConst c uty) = {}"
"varsOfS (SRole r) = {}"
"varsOfS (SFunc f) = {}"
"varsOfS (SApp n arg) = varsOfS arg"
"varsOfS (SPair x y) = varsOfS x ∪ varsOfS y"
"varsOfS (SEnc x k) = varsOfS x ∪ varsOfS k"

A specialization of *subtermsS* is *varsOfS*, which computes the set of all identifiers of the variables contained in a roleterm. This set is for example used to describe how a successful matching extends the domain of the variable map of an instantiation.

3.2.3 Role Specifications

Role events denote the actions of an agent during the execution of a protocol role. This can either be the sending of a specific message, the receipt of a message matching the specified

pattern, or a local claim that the specified security objective holds. All of these events have a label (the first constructor argument) which is used to make them unique in a protocol specification.

```
datatype 'a roleevent =
    SSend 'a "'a roleterm"
  | SRead 'a "'a roleterm"
  | SClaim 'a 'a string "'a roleterm option"
```

A *role specification* consists of the initial role knowledge and the list of events (actions) this role is expected to execute. The purpose of the initial role knowledge is twofold. First, it used to compute the initial intruder knowledge from the knowledge of all untrusted agents and second, it is used to check, if a role specification can be implemented by the intruder. This is explained further in the section about well-formedness.

```
record 'a rolespec =
  rspecKnow :: "'a roleterm set"
  rspecEvs  :: "'a roleevent list"
```

Example: Initiator and Responder Role of the NSL protocol

We have now defined all constructs needed to define the two roles of the Needham-Schroeder-Lowe protocol (see 2.1 for a MSC). Let us just look at the specification of the initiator role first:

```
constdefs
  nsl_i      :: "string rolespec"
    — The initiator role of the NSL−Protocol
  nsl_i_def  :
 "nsl_i      ≡ (|
    rspecKnow = {SRole ''i'', SRole ''r'', SConst ''ni'' None,
     SApp ''sk'' (SRole ''i''),
     SApp ''pk'' (SRole ''i''), SApp ''pk'' (SRole ''r'') },

    rspecEvs =
    [SSend ''1'' ⟨SRole ''i'', SRole ''r'',
                  SEnc ⟨SRole ''i'', SConst ''ni'' None⟩
                       (SApp ''pk'' (SRole ''r'')) ⟩,
     SRead ''2'' ⟨SRole ''r'', SRole ''i'',
                  SEnc ⟨SConst ''ni'' None, SVar ''V'' ConstT, SRole ''r''⟩
                       (SApp ''pk'' (SRole ''i'')) ⟩,
     SSend ''3'' ⟨SRole ''i'', SRole ''r'',
                  SEnc (SVar ''V'' ConstT) (SApp ''pk'' (SRole ''r'')) ⟩,
     SClaim ''4'' ''i'' ''secret'' (Some (SConst ''ni'' None))
    ]
  |)"
```

The initiators role knowledge consists of the two roles i and r, the nonce ni with no user defined type (i. e., its type is *ConstT*), its secret and public key, and the public key of the responder.

The role events follow the description given in the message sequence chart. The claim event

SClaim ''4'' '' *i* '' '' *secret* '' (*Some* (*SConst* ''*ni*'' *None*))

is interpreted as a claim of the secrecy of *SConst* ''*ni*'' *None*. This claim is local with respect to the role i.

The the responder role is specified as follows. Note the typing of the variables in both role specifications. They should only match nonces and are therefore typed to *ConstT*.

constdefs

```
  nsl_r      :: "string rolespec"
    — The responder role of the NSL−Protocol
  nsl_r_def  :
 "nsl_r      ≡ (|
    rspecKnow = {SRole ''r'', SRole ''i'', SConst ''nr'' None,
     SApp ''sk'' (SRole ''r''),
     SApp ''pk'' (SRole ''i''), SApp ''pk'' (SRole ''r'') },

    rspecEvs =
    [SRead ''1'' ⟨SRole ''i'', SRole ''r'',
                 SEnc ⟨SRole ''i'', SVar ''W'' ConstT⟩
                      (SApp ''pk'' (SRole ''r'')) ⟩,
     SSend ''2'' ⟨SRole ''r'', SRole ''i'',
                 SEnc ⟨SVar ''W'' ConstT, SConst ''nr'' None, SRole ''r''⟩
                      (SApp ''pk'' (SRole ''i'')) ⟩,
     SRead ''3'' ⟨SRole ''i'', SRole ''r'', SEnc (SConst ''nr'' None)
                 (SApp ''pk'' (SRole ''r'')) ⟩,
     SClaim ''4'' ''r'' '' secret '' (Some (SConst ''nr'' None))
    ]
 |)"
```

Well-formed Role Specifications

Not all role specifications are such that the intruder is able to impersonate an untrusted agent.[2] This is only possible, if it is implementable by an agent who has to infer all terms needed during the execution of the protocol from the initial role knowledge plus the variables he has already read. The difficult step is to define when a read is implementable. This is captured by the *readability* predicate *RD*.

[2] A typical example is a role specification where an agent sends a term he does not know and that is not in the initial intruder knowledge.

```
consts
  RD :: " ['a inverse, 'a roleterm set, 'a roleterm ] ⇒ bool"
    — Is a roleterm readable with the given knowledge?
primrec
  "RD kif M (SVar v ty) = True"
  "RD kif M (SConst c uty) = ((SConst c uty) ∈ inferS kif M)"
  "RD kif M (SRole r) = ((SRole r) ∈ inferS kif M)"
  "RD kif M (SFunc f) = ((SFunc f) ∈ inferS kif M)"
  "RD kif M (SApp n arg) = ((SApp n arg) ∈ inferS kif M)"
  "RD kif M (SPair x y) =
     ((RD kif (M ∪ {SVar v ty | v ty. SVar v ty ∈ inferS kif (insert y M) }) x) ∧
      (RD kif (M ∪ {SVar v ty | v ty. SVar v ty ∈ inferS kif (insert x M) }) y))"
  "RD kif M (SEnc x k) = ((SEnc x k) ∈ inferS kif M ∨
                          (kif⦇k⦈s ∈ inferS kif M ∧ RD kif M x))"
```

The predicate *WF_role_aux* captures the implementability conditions for a list of events. It checks, if the roles of the events correspond to the role name of the role specification (given as a parameter), if the sent terms are inferable from the current knowledge, and if the read terms are readable under the current knowledge. A role specification that satisfies this predicate and does not contain any variables in the role knowledge is said to be *implementable*.

```
consts
  WF_role_aux :: " ['a inverse, 'a, 'a roleterm set, 'a roleevent list ] ⇒ bool"
primrec
  "WF_role_aux kif r M [] = True"
  "WF_role_aux kif r M (e#es) = (
     case e of
       (SSend l m) ⇒ (
         (case m of
            SPair rS m' ⇒
              (case m' of
                 SPair rR m'' ⇒ rS = SRole r
               | (_)          ⇒ False)
          | (_)               ⇒ False) ∧
         (m ∈ inferS kif M) ∧
         (WF_role_aux kif r M es)
       )
     | (SRead l m) ⇒(
         (case m of
            SPair rS m' ⇒
              (case m' of
                 SPair rR m'' ⇒ (rR = SRole r) ∧
                                (⟨rS, rR⟩ ∈ inferS kif M) ∧
                                (RD kif M m'')
```

```
                | (_)              ⇒ False)
            | (_)            ⇒ False) ∧
          (WF_role_aux kif r (insert  m M) es)
      )
    | (SClaim l rC n xopt) ⇒ (
        (rC = r) ∧
        (SRole rC ∈ inferS  kif  M) ∧
        (WF_role_aux kif r M es)
      )
  )"
```

Apart from implementability, *well-formedness* also incorporates conditions which make reasoning about protocols easier. These conditions do not concern an agent executing the role, but they are important with respect to the observer perspective we are taking when verifying a security protocol. With respect to role specifications, we need their events to be unique. This implies the existence of a (partial) ordering of the role events consistent with their order in the list of events of the role specification.

constdefs

```
  WF_role                :: "['a inverse, 'a, 'a rolespec] ⇒ bool"
    — Is a role  specification  well−formed?
  WF_role_def            :
  "WF_role kif r spec    ≡
     WF_role_aux kif r (rspecKnow spec) (rspecEvs spec) ∧
     (∀ v ty. SVar v ty ∉ subtermsS (rspecKnow spec)) ∧
     (distinct  (rspecEvs spec))"
```

3.2.4 Protocols

A *protocol* consists of a key inversion function and a map from roles to role specifications.

record *'a proto* =

```
  protoKIF    :: "'a inverse"
  protoRspecs :: "'a ⇀ 'a rolespec"
```

Example: The NSL-Protocol

The role specifications of the Needham-Schroeder-Lowe protocol where already given in 3.2.3. They are assembled with a key inversion function mapping public keys *''pk''* to secret keys *''sk''* and vice versa to build a formal specification of the Needham-Schroeder-Lowe protocol.

constdefs
nsl_proto :: "string proto"
— The NSL−Protocol
nsl_proto_def :
"nsl_proto ≡ (| protoKIF = ["pk" ↦ "sk", "sk" ↦ "pk"],
protoRspecs = ["i" ↦ nsl_i, "r" ↦ nsl_r] |)"

Decomposing Protocols

The following decomposition functions are used when describing the properties of a protocol.

constdefs
protoRoleEvs :: "['a proto, 'a] ⇒ 'a roleevent set"
— All events of a specific role in a protocol
protoRoleEvs_def :
"protoRoleEvs P r ≡ ⋃(λ spec. set (rspecEvs spec)) ‘ o2s (protoRspecs P r)"

constdefs
protoEvs :: "'a proto ⇒ 'a roleevent set"
— All events occurring in a protocol
protoEvs_def : "protoEvs P ≡
⋃(λ r. set (rspecEvs (the (protoRspecs P r)))) ‘ dom (protoRspecs P)"

constdefs
msgs :: "'a proto ⇒ 'a roleterm set"
— All messages sent in a protocol
msgs_def : "msgs P ≡ {m | l m. SSend l m ∈ protoEvs P}"

Well-formed Protocols

A protocol which consists only of well-formed roles and whose events are unique is *well-formed*. Most properties can obviously only be proved about well-formed protocols.

constdefs
WF :: "'a proto ⇒ bool"
— Is a protocol well−formed?
WF_def :
"WF P ≡ (∀ r ∈ dom (protoRspecs P).
case protoRspecs P r **of**
Some spec ⇒WF_role (protoKIF P) r spec
| None ⇒ True) ∧
(∀ r ∈ dom (protoRspecs P). ∀r' ∈ dom (protoRspecs P).
r ≠ r' ⟶ protoRoleEvs P r ∩ protoRoleEvs P r' = {})"

The function *role* computes the protocol role whose specification the event belongs to. Using the axiom of choice this can be defined for all protocols. However proofs involving the concrete value of this function are only possible if the events of the protocol are unique, i. e., the protocol is well-formed.

constdefs
role :: "['a proto, 'a roleevent] ⇒ 'a option"
role_def :
"role P ev ≡ **if** (∃ r. ev ∈ protoRoleEvs P r)
then Some (SOME r. ev ∈ protoRoleEvs P r)
else None"

Protocol Ordering Relations

The *event ordering* induced by a protocol plays an important role in protocol verification. It captures the fact that an agent executes events in the order specified by the role specification.[3]

constdefs
evOrd :: "'a proto ⇒ ('a roleevent × 'a roleevent) set"
evOrd_def :
"evOrd P ≡ ⋃{ beforeOrd (rspecEvs spec) | spec. spec ∈ ran (protoRspecs P)}"

The *communication relation* captures the relation between a send event and its intended read event. This relation is described in the protocol specification by giving equal labels to corresponding send and read events.

constdefs
comRel :: "('a roleevent × 'a roleevent) set"
comRel_def : "comRel ≡ { (SSend l ms, SRead l mr) | l ms mr. True}"

The *protocol ordering* is the transitive closure of the event ordering and the communication relation. It captures all dependencies of a protocols events. Informally, this dependencies correspond to the transitive closure of all arrows in the protocols message sequence chart.

constdefs
protoOrd :: "'a proto ⇒ ('a roleevent × 'a roleevent) set"
protoOrd_def : "protoOrd P ≡ (evOrd P ∪ comRel)$^+$"

[3]The ordering *beforeOrd* is part of the HOL list theory extension developed for this formalization. Further information can be found in A.1

3.3 Execution Model

We model the execution of a protocol as a set of traces. These traces represent all possible interleavings of actions agents can take (creating new runs, sending and receiving messages, and claiming that some property holds) and state changes that the network (partially) controlled by the intruder can achieve.

The *network* and the *intruder* are modeled as a single component, which we will refer to under either name depending on the context. *Agents* are simple protocol automata executing protocol roles step by step using the send and receive buffer of the network for *asynchronous communication*. Such an execution of a specific role by a specific agent is called a *run*. Each run has a unique *run identifier*, which is used to make constants generated in this run unique and thus guarantee their freshness. Therefore, the dynamic message terms sent and received during the execution differ from the static message terms used in a protocol role specification. This dynamic message terms are generated by *instantiating* the static terms from the role specification with the *instantiation* of the corresponding run.

The *system state* consists of a set of runs representing the state of the participating agents, a send and a receive buffer, and the intruder knowledge. In our formalization, it is represented using the following record.

record *'a state* =
  stF :: "'a run set" — state of all agents
  stBS :: "'a runterm multiset" — send buffer
  stBR :: "'a runterm multiset" — receive buffer
  stM :: "'a runterm set" — intruder knowledge

In the initial system state, the set of runs and the network buffers are empty and the intruder knowledge is computed from the role knowledge of the untrusted agents. A single execution step is called a *transition*. Each transition consists of a *label* stating what has happened and the *state* of the system after this transition. Thus a *trace* is just a list of transitions.

The concrete function mapping a protocol to the set of all possible traces is parametrized over the environment used to execute the protocol. This *execution environment* specifies which typeflaw system is used and and what network (intruder) is present. Due to this parametrization, we are able to investigate formally the relation between different typeflaw systems and intruders.

3.3.1 Dynamic Message Terms

In 3.2.2 we described the static message terms, which we call *roleterms*. These terms are *instantiated* to dynamic message terms, which are called *runterms*, during the execution of a protocol. Runterms can be seen as an extended version of roleterms: Agents are added. Variables, roles, and constants are made unique with respect to individual runs of the protocol by adding the run identifier as a constructor argument. The intruder gets a

special set of *intruder constants*, which he can use wherever a normal agent would use a constant.

The concrete construction is done as follows: The set of agents is partitioned into a set of *trusted* agents and a set of *untrusted* agents. Untrusted agents are compromised by the intruder. This is achieved by including their initial role knowledge, and as such, also all keys they know, into the initial intruder knowledge.

datatype *'a agent* =
  Trusted 'a
| *Untrusted 'a*

Apart from the set of agents, we also need a set of run identifiers, whose elements are used to uniquely identify the individual runs of the protocol during execution. Thus it is important that this set is infinite, because otherwise our model would only include a finite number of runs. Apart from being infinite, there are no other constraints on this set. Therefore, we are using its elements to note properties of the run which would otherwise be inaccessible during proofs about a protocol execution. Currently, the run identifier of a run tells us what agent is executing which role of the protocol. The *ridId* field is used to differ between different runs of the same agent executing the same role.

record *'a runid* =
  ridAgent :: "'a agent"
  ridRole :: *'a*
  ridId :: *nat*

Like roleterms, runterms are constructed as a inductive datatype. All constructors are prefixed with an 'D' for dynamic. The syntax for pairing, which is *not associative*, is setup such that $\{\!|A, B, NA|\!\}$ corresponds to (*DPair A* (*DPair B NA*)).

datatype *'a runterm* =
  DVar 'a "'a runid" "'a type" — symbolic variables
| *DConst 'a* "'a runid" "'a option"
| *DIntrConst 'a* "'a option" ("DIConst") — intruder constants
| *DRole 'a* "'a runid" — symbolic roles
| *DAgent* "'a agent"
| *DFunc 'a*
| *DApp 'a* "'a runterm"
| *DPair* "'a runterm" "'a runterm"
| *DEnc* "'a runterm" "'a runterm"

Instantiations

When *instantiating* a roleterm to a runterm the following three actions are taken:

- roles are substituted with concrete agents

- variables are substituted with their values
- constants are extended with the run identifier to make them unique

The type of variable substitutions is a map from identifiers to pairs of runterms and the type of the variable. This implies that we cannot have variables with different types.

types *'a dvarsubst* = "'a ⇀ ('a runterm × 'a type)"

An *instantiation* consists of a run identifier, a map from roles to agents, and a variable substitution.

datatype *'a inst* =
Inst "'a runid" "'a ⇀ 'a agent" "'a dvarsubst"

These fields can be accessed using the following selector functions.[4]

consts
instRID :: "'a inst ⇒ 'a runid"
instRMap :: "'a inst ⇒ ('a ⇀ 'a agent)"
instVMap :: "'a inst ⇒ 'a dvarsubst"

An instantiation *inst* is applied to a roleterm x using the function *instantiate*. This is written as $\langle inst \rangle(x)$. In order to make *instantiate* total, the following actions are also taken upon instantiation:

- variables that are not assigned a value by *inst* are instantiated to *symbolic variables*
- roles that are not assigned an agent by *inst* are instantiated to *symbolic roles*

Both symbolic roles and symbolic variables are marked with the run identifier of *inst* to make them unique throughout the execution of a protocol.

consts
instantiate :: "['a inst, 'a roleterm] ⇒ 'a runterm" ("⟨_⟩'(_')")
primrec
"instantiate inst (SVar v ty) =
(**case** instVMap inst v **of**
Some dt ⇒ fst dt
| None ⇒ DVar v (instRID inst) ty)"
"instantiate inst (SConst c uty) = DConst c (instRID inst) uty"
"instantiate inst (SRole r) =
(**case** instRMap inst r **of**
Some agent ⇒ DAgent agent
| None ⇒ DRole r (instRID inst))"
"instantiate inst (SFunc f) = DFunc f"
"instantiate inst (SApp f arg) =
DApp f (instantiate inst arg)"
"instantiate inst (SPair x y) =

[4] See A.1 for their definitions

```
    DPair (instantiate inst x) (instantiate inst y)"
  "instantiate inst (SEnc x k) =
    DEnc (instantiate inst x) (instantiate inst k)"
```

Example: The First Message of the Needham-Schroeder-Lowe protocol Assuming that the trusted agent A starts a communication with the compromised agent E using the Needham-Schroeder-Lowe protocol, then the instantiation used when sending the first message would perform the following steps:

- replace the protocol roles with the assigned agents
- make the nonce ni unique by added the run identifier rid

The variable map of the instantiation is *empty*, because it is the first message and therefore no variable has yet been assigned a value during a read event. In Isabelle notation the described instantiation would look as follows.

```
⟨Inst rid [''i''↦Trusted ''A'', ''r''↦Untrusted ''E''] empty⟩(
   ⟨SRole ''i'', SRole ''r'',
    SEnc ⟨SRole ''i'', SConst ''ni'' None⟩ (SApp ''pk'' (SRole ''r'')) ⟩
 ) =
 {|DAgent (Trusted ''A''), DAgent (Untrusted ''E''),
  DEnc {|DAgent (Trusted ''A''), DConst ''ni'' rid None|}
       (DApp ''pk'' (DAgent (Untrusted ''E'')))|}
```

The Type of a Runterm

Every runterm is assigned a unique type. It is computed as follows.

```
consts
  getType :: "'a runterm ⇒ 'a type"
    — Compute the type of a runterm
primrec
  "getType (DVar v rid ty) = ty"
  "getType (DConst c rid uty) = (case uty of
                                   Some ty ⇒UserT ty
                                 | None    ⇒ ConstT)"
  "getType (DIConst c uty) = (case uty of
                                Some ty ⇒UserT ty
                              | None    ⇒ ConstT)"
  "getType (DRole r rid) = AgentT"
  "getType (DAgent a) = AgentT"
  "getType (DFunc f) = FuncT"
  "getType (DApp f arg) = AppT (getType arg)"
  "getType (DPair x y) = PairT (getType x) (getType y)"
  "getType (DEnc x k) = EncT (getType x) (getType k)"
```

Using this definition, variable substitutions can be checked for well-typedness. However, this property is currently not used in our formalization. It is only formalized to show the correspondence to the original description of the OSSP.

constdefs

```
  welltyped      :: " ['a conformance, 'a dvarsubst] ⇒ bool"
    — Is a substitution well−typed?
  welltyped_def :
 "welltyped conf vMap ≡ ∀ v ∈ dom vMap. case vMap v of
     Some objTy ⇒(getType (fst objTy), snd objTy) ∈ conf
   | None       ⇒ True"
```

Matching

When a read event *SRead l pat* is executed, the incoming message *obj* is matched against the pattern *pat*. This matching extends the given instantiation *inst* to an instantiation *inst'* such that $\langle inst' \rangle(obj) = pat$. The function computing this matching is *matchSD*, whose name means 'match a static term to a dynamic term'. It is a partial function parametrized over the type conformance relation.

consts

```
  matchSD     :: " ['a conformance, 'a inst , 'a roleterm, 'a runterm]
                  ⇒ 'a inst option"
  — Match a runterm against a roleterm pattern under a given instantiation
primrec
   "matchSD conf inst (SVar v ty) obj = (
     case (instVMap inst v) of
       Some vInstTy ⇒
         if (fst vInstTy = obj ∧ snd vInstTy = ty)
         then Some inst
         else None
     | None       ⇒
         if    ((getType obj, ty) ∈ conf)
         then (Some (Inst (instRID inst) (instRMap inst)
                          ((instVMap inst)(v ↦(obj,ty)))))
         else None)"
  "matchSD conf inst (SConst c uty) obj = (
     if (DConst c (instRID inst) uty = obj) then (Some inst) else None)"
  "matchSD conf inst (SRole r) obj = (
     case (instRMap inst r) of
       Some a ⇒if (DAgent a = obj) then (Some inst) else None
     | None  ⇒ if (DRole r (instRID inst) = obj) then (Some inst) else None)"
  "matchSD conf inst (SFunc f) obj = (
     if (DFunc f = obj) then (Some inst) else None)"
  "matchSD conf inst (SApp f arg) obj = (
```

```
    case obj of
      (DApp f' arg') ⇒
        if  f = f'  then (matchSD conf inst arg arg') else  None
    | (_)              ⇒ None)"
"matchSD conf inst (SPair x y) obj = (
    case obj of
      (DPair x' y') ⇒ (case (matchSD conf inst x x') of
                          Some inst' ⇒ matchSD conf inst' y y'
                        | None       ⇒ None)
    | (_)           ⇒ None)"
"matchSD conf inst (SEnc x k) obj = (
    case obj of
      (DEnc x' k') ⇒ (case (matchSD conf inst x x') of
                        Some inst' ⇒ matchSD conf inst' k k'
                      | None       ⇒ None)
    | (_)          ⇒ None)"
```

If it holds that *Some inst' = matchSD conf inst pat obj*, i. e., we *successfully* computed a new instantiation *inst'* by matching (controlled by the type conformance relation *conf*) a message *obj* to a pattern *pat* partially instantiated by *inst*, then the following properties hold:

— the object is equal to the instantiated pattern
$\langle inst' \rangle (pat) = obj$

— the run identifier and role map are left unchanged
$instRID\ inst' = instRID\ inst$
$instRMap\ inst' = instRMap\ inst$

— the variable map is *extended* with the variables in the pattern
$instVMap\ inst \subseteq_m instVMap\ inst'$
$dom\ (instVMap\ inst') = dom\ (instVMap\ inst) \cup (varsOfS\ pat)$

— the result is well-typed, if the argument was well-typed
$welltyped\ \ conf\ (instVMap\ inst) \longrightarrow welltyped\ conf\ (instVMap\ inst')$

Functions over runterms

All functions that are defined over roleterms are defined analogously over runterms. Therefore only their type signature is given here. For their definitions see A.1 and for a description of their intent see 3.2.2.

consts *invertKey* :: " ['a inverse , 'a runterm] ⇒ 'a runterm" ("_(|_|)")

The syntax for key inversion on runterms is $kif(|key|)$. Instead of marking the functions on runterms with a 'D' for dynamic, the 'S' of the static versions is just dropped. The reason being just notational brevity.

consts
subterms :: "'a runterm set ⇒ 'a runterm set"
parts :: "'a runterm set ⇒ 'a runterm set"
analz :: "['a inverse, 'a runterm set] ⇒ 'a runterm set"

synth :: "'a runterm set ⇒ 'a runterm set"
infer :: "['a inverse, 'a runterm set] ⇒ 'a runterm set"

varsOf :: "'a runterm ⇒'a set"

3.3.2 Agents

The execution model consists of two subcomponents. One modeling the execution of protocol roles by agents and the other modeling the network combined with the intruder. Central to the agents execution model is the notion of a *run*. A run corresponds to the execution of a specific role by a specific agent. When a run is created, it consists of a copy of the role events of the corresponding role specification (*runEvs*) and an instantiation (*runInst*) which contains the run identifier, a role map assigning each role in the protocol an agent, and an empty variable map. This instantiation is used to keep track of the terms assigned to individual variables during the execution of a read event. The events of the run denote the remaining events that have to be executed to complete this role.

record *'a run* =
runInst :: "'a inst"
runEvs :: "'a roleevent list"

The above definition of a run captures everything that has to be known about the state of an agent to simulate its execution of a protocol role. However when analyzing the system, some properties depend on the past events an agent has already executed. The function denoting these past events is called *runPast*. Its definition exploits the fact that the events in a protocol are unique and thus the past can be computed by taking all events at the beginning of the role specification up to the current event of the run.

constdefs
runPast :: "['a proto, 'a run] ⇒ 'a roleevent list"
— The past events of a run
runPast_def :
"runPast P run ≡ (**case** (runEvs run) **of**
([]) ⇒ (λ specEvs. specEvs)
| (ev#evs) ⇒(λ specEvs. takeWhile (λ x. x≠ ev) specEvs))
(rspecEvs (the (protoRspecs P (ridRole (instRID (runInst run))))))"

There exists an infinite set *runsOf* from which new runs are taken from upon creation during the execution of a protocol. This set contains all runs whose run events are a copy of some role specification of the protocol and whose instantiation maps all roles of the protocol to some agent and has an empty variable map.

constdefs
runsOf :: "'a proto ⇒ 'a run set"
— All possible runs of a protocol
runsOf_def :
"runsOf P ≡
{ (|runInst = (Inst ((|ridAgent=the(rMap R), ridRole=R, ridId=id|)) rMap empty),
runEvs = (rspecEvs (the (protoRspecs P R)))
|)
| id rMap R.
R ∈ dom (protoRspecs P) ∧
dom rMap = dom (protoRspecs P)
}"

When creating a new run, only those in *runsOf* which have a run identifier that has not yet been used are valid candidates. The projection of the current set of runs to the set of used run identifiers is captured by the function *runIDs*.

constdefs
runIDs :: "'a run set ⇒ 'a runid set"
— The run identifiers of a set of runs
runIDs_def : "runIDs F ≡ (λ run. instRID (runInst run)) ‘ F"

3.3.3 Network and Intruder

The network is modeled as a relation over system states with the property that the set of runs is left unchanged. This relation captures all possible state transitions a network could possibly achieve. However, this relation is dependent on the key inversion function to be used (e. g., the fake rule of the Dolev-Yao intruder). This dependency and the condition on the set of runs is captured by a rule. A *rule* is a relation which is parametrized over the key inversion function to be used and leaves the set of runs unchanged.

types *'a state_trans* = "('a state × 'a state) set"

typedef *'a rule* = "{R::'a inverse ⇒ 'a state_trans .
∀ kif . ∀ (st , st') ∈ R kif. stF st = stF st'}"

In principle, the network, which is (partially) controlled by the intruder, could now be modeled directly as a single rule. However, because all networks share common state transitions, we model a *network* as a set of rules.

types *'a network* = "'a rule set"

The actual state transition relation to be used when defining the set of all traces of a protocols execution is computed by taking the transitive closure of the union of the relations provided by the individual rules. This is captured by the function *rule_closure*.

constdefs

rule_closure :: "['a inverse, 'a rule set] ⇒ 'a state_trans"
rule_closure_def : "rule_closure kif R ≡ (⋃ {Rep_rule r kif | r. r ∈ R})⁺"

Concrete rules are defined as functions mapping a key inversion function to its corresponding state transition relation. In our case, all of these relations are total and their side conditions are captured in the condition on the elements of the set. Due to the type definition of a rule, this functions must be mapped to rules by means of the *Abs_rule* injection.

constdefs

rule_transmit :: "'a rule"
rule_transmit_def :
"rule_transmit ≡ Abs_rule (λ kif.
{ (st, st(|stBS:=stBS st − {#m#}, stBR:=stBR st + {#m#}|)) | st m.
m :# stBS st
})"

rule_take :: "'a rule"
rule_take_def :
"rule_take ≡ Abs_rule (λ kif.
{ (st, st(|stM:=insert m (stM st), stBS:=stBS st − {#m#}|)) | st m.
m :# stBS st
})"

rule_fake :: "'a rule"
rule_fake_def :
"rule_fake ≡ Abs_rule (λ kif.
{ (st, st(|stBR:=stBR st + {#m#}|)) | st m.
m ∈ infer kif (stM st)
})"

rule_eavesdrop :: "'a rule"
rule_eavesdrop_def :
"rule_eavesdrop ≡ Abs_rule (λ kif.
{(st, st(|stM:=insert m (stM st),
stBS:=stBS st − {#m#}, stBR:=stBR st + {#m#}|)) | st m.
m :# stBS st
})"

rule_jam :: "'a rule"
rule_jam_def :
"rule_jam ≡ Abs_rule (λ kif.

```
{(st, st(|stBS:=stBS st − {#m#}|)) | st m. m :# stBS st})"
```

Networks capturing the presence of no intruder, a passive intruder which is only eavesdropping, a wireless intruder (i. e., a intruder which can either jam or eavesdrop, but not both at the same time), and the well-known Dolev-Yao intruder are now defined using the above rules. The lemma suffixes are used to identify lemmas holding only for a specific intruder.

constdefs

```
  no_intruder      :: "'a network"
— Lemma−suffix: NI
  no_intruder_def  : "no_intruder ≡ {rule_transmit}"

  passive_intruder      :: "'a network"
— Lemma−suffix: PI
  passive_intruder_def  :
"passive_intruder ≡ {rule_eavesdrop}"

  wireless_intruder      :: "'a network"
— Lemma−suffix: WI
  wireless_intruder_def  :
"wireless_intruder ≡ {rule_eavesdrop, rule_jam, rule_fake}"

  dolev_yao      :: "'a network"
— Lemma−suffix: DY
  dolev_yao_def  : "dolev_yao ≡ {rule_take, rule_fake}"
```

3.3.4 Protocol Traces

Before defining the operational semantics of security protocols, we need to define what a transition is, how we represent the execution environment, and how the initial intruder knowledge is computed.

Transitions

A trace is a sequence of transitions. It represents a possible interleaving of the actions agents and the network can take. A transition consists of a transition label and the system state after this transition. There are four types of transitions: The *Init* transition happens at the very beginning of every trace. It is used to inject the initial system state. The *Create* transition states, that a new run has been created. The *RunEvent* transition denotes an action an agent has taken, i. e., a role event that has happened for an agent in a specific run under the given instantation. The *Network* transition states, that the network has made a state transition. Note that the *Create*, *RunEvent*, and *Network* transition labels have associated abbreviations. They help making proof states less verbose.

types *'a runevent* = "'a inst × 'a roleevent"

datatype *'a trans_label* =
  Init
| *Create* "'a run" ("Cre")
| *RunEvent* "'a runevent" ("REv")
| *Network* ("Net")

For notational brevity (i. e., because Isabelle/HOL sets up a different syntax for inductive datatypes than for records), we define *transitions* as an inductive datatype.

datatype *'a transition* = *TR* "'a trans_label" "'a state"

In order to access label and state of a transition without a case construct, the following two selectors are defined.

constdefs
  trST :: "'a transition ⇒ 'a state"
  — Access the state of a transition
  trST_def : "trST tr ≡ **case** tr **of** TR l st ⇒ st"

  trL :: "'a transition ⇒ 'a trans_label"
  — Access the label of a transition
  trL_def : "trL tr ≡ **case** tr **of** TR l st ⇒ l"

Environments

In order to parametrize the execution of a protocol over different message type systems and intruder models, we introduce *environments*. This is currently a record consisting of the type conformance relation and the network to be used during execution. In the future, we would like to extend environments with further parameters like for example the initial network state or bounds on the number or types of runs. This parametrization allows us to formally compare and make use of different execution models of security protocols.

record *'a environment* =
  envConf :: "'a conformance"
  envNet :: "'a network"

Initial Intruder Knowledge

The intruder knowledge is stored in the field *stM* of a system state and is extended during the execution of the protocol. The initial state of the system must explicitly specify the intruder knowledge. This initial knowledge can be computed explicitly due to the inclusion of the role knowledge in the protocol definitions. It contains all the intruder constants and

the instantiated role knowledge without terms containing constants of every role where the role itself is instantiated to an untrusted agent.

constdefs
initM :: "'a proto ⇒ 'a runterm set"
— The initial intruder knowledge
initM_def :
"initM p ≡ {DIntrConst ic uty | ic uty. True} ∪
⋃{ instantiate (Inst rid roleMap empty) ‘
{rt ∈ ⋃rspecKnow ‘ o2s (protoRspecs p R).
(∀ c uty. SConst c uty ∉ subtermsS {rt})
}
| rid roleMap R. (∃ agent. Some (Untrusted agent) = roleMap R) ∧
(dom roleMap = dom (protoRspecs p))}"

Example: The Needham-Schroeder-Lowe protocol Here the initial intruder knowledge has the following form:

"initM nsl_proto = {DIntrConst ic uty | ic uty. True} ∪
range DAgent ∪
{DApp ''pk'' (DAgent a) | a. True} ∪
{DApp ''sk'' (DAgent (Untrusted a)) | a. True}"

The intruder knows all the intruder constants, the names and public keys of all agents, and the private keys of all *compromised* agents.

Operational Semantics

Executing a protocol under a given execution environment means mapping it to its corresponding set of traces. They are modeled as finite lists of transitions.

types '*a trace* = "'a transition list"

The set of traces of a protocols execution is defined inductively and parametrized over an environment and the protocol to be executed. Traces grow at the head of the list. Thus the introduction rules can use pattern matching (*TR lb s*)#*t* ∈ *traces env P* to access the last transitions state *s*, which represents the current system state. Using this system state, the conditions for a specific transition to happen can be specified. The new extended trace is then added with the corresponding label and an updated system state to the set of traces. Note that in the case of a *Network* transition, the conditions on the system states are internalized in the definition of the networks rules.

consts
traces :: "['a environment, 'a proto] ⇒ 'a trace set"
inductive "traces env P"

intros

Init:
"[TR Init (⦇stM=initM P, stBS={#}, stBR={#}, stF={}⦈)]
∈ traces env P"

Create:
"⟦ (TR lb s)#ts ∈ traces env P;
run ∈ runsOf P;
instRID (runInst run) ∉ runIDs (stF s) ⟧
⟹
(TR (Create run) (s⦇stF := insert run (stF s)⦈))#((TR lb s)#ts)
∈ traces env P"

Send:
"⟦ (TR lb s)#ts ∈ traces env P;
run ∈ stF s; run = ⦇runInst=inst, runEvs=(SSend l m)#evs⦈ ⟧
⟹
(TR (RunEvent (inst, SSend l m))
(s⦇stBS := stBS s + {# ⟨inst⟩(m) #},
stF := insert (⦇runInst=inst, runEvs=evs⦈) (stF s − {run})⦈)
)#((TR lb s)#ts)
∈ traces env P"

Read:
"⟦ (TR lb s)#ts ∈ traces env P;
m :# stBR s;
run ∈ stF s; run = ⦇runInst=inst, runEvs=(SRead l pat)#evs⦈;
Some inst' = matchSD (envConf env) inst pat m ⟧
⟹
(TR (RunEvent (inst, SRead l pat))
(s⦇stBR := stBR s − {#m#},
stF := insert (⦇runInst=inst', runEvs=evs⦈) (stF s − {run})⦈)
)#((TR lb s)#ts)
∈ traces env P"

Claim:
"⟦ (TR lb s)#ts ∈ traces env P;
run ∈ stF s; run = ⦇runInst=inst, runEvs=(SClaim l rC claim mopt)#evs⦈ ⟧
⟹
(TR (RunEvent (inst, SClaim l rC claim mopt))
(s⦇stF := insert (⦇runInst=inst, runEvs=evs⦈) (stF s − {run})⦈)
)#((TR lb s)#ts)
∈ traces env P"

Net:

```
"⟦ (TR lb s)#ts ∈ traces env P;
   (s, s') ∈ rule_closure (protoKIF P) (envNet env) ⟧
 ⟹
   (TR Net s')#((TR lb s)#ts) ∈ traces env P"
```

Different Typeflaw Models

The typeflaw model can be changed by changing the conformance relation of the execution environment. The conformance relations defined in 3.2.1 satisfy the following subset relation.

$$noTypeFlaws \subseteq basicTypeFlaws \subseteq fullTypeFlaws$$

Therefore, we define a type system T_{strict} as being *more strict* than a type system $T_{lenient}$, if the conformance relation of T_{strict} is a subset of the conformance relation of $T_{lenient}$. This definition is supported by the fact that all traces existing under T_{strict} also exist under $T_{lenient}$ in our model, which is a direct consequence of the fact that all terms matching under the strict conformance relation also match under the lenient conformance relation.

lemma *stricter_tracesD*:

```
"⟦ t ∈ traces (env(|envConf:=strict|)) P; strict ⊆ lenient ⟧ ⟹
 t ∈ traces (env(|envConf:=lenient|)) P"
```

The expected consequence stating that secrecy under a type flaw system implies secrecy under all more strict systems is shown in 3.4.1.

3.3.5 The Dolev-Yao Intruder

Many properties of a protocols execution depend on the concrete behavior of the intruder. This means that they have to be proved individually for each intruder. However, common sense tells us that it should be possible to exploit the relations between different intruders in these proofs. One such relation often used informally in literature is that some intruders are "weaker" than others. Thus assuming the strongest possible intruder should give the strongest guarantees with respect to security protocol verification. This is the reason, why a Dolev-Yao type intruder, which is considered to be the strongest possible intruder, is almost always assumed in literature. [14]

In this section, we will give a formal definition of what it means for an intruder to be weaker than another one. The most simple definition would be that all traces existing under the weaker intruder also exist under the stronger intruder. However, this definition does not capture the case where a stronger intruder *knows more* than the weaker one, i. e., his intruder knowledge is a superset of the intruder knowledge of the weak intruder. Thus the simple definition has to be extended as follows: A weak intruder is *weaker* than a strong intruder, if all traces existing under the weak intruder also exist under the strong intruder

with the exception that the intruder knowledge of the strong intruder is only required to be a superset, instead of being equal to the knowledge of the weak intruder.

Formalizing this definition requires us to introduce two new predicates: The first one, written as $sw \equiv_{\subseteq M} ss$, captures the equality over system states where the intruder knowledge of the "strong state" *ss* is only required to be a superset of the "weak state" *sw*. One can also view it as an equality with bounded intruder knowledge.

constdefs
boundedM_eq :: "['a state, 'a state] ⇒ bool" (**infixl** "≡ ⊆M" 60)
— Everything is equal except for stM which is bounded
boundedM_eq_def :
"(sw ≡ ⊆M ss) ≡ ((stM sw ⊆ stM ss) ∧ (stBS sw = stBS ss) ∧
(stBR sw = stBR ss) ∧ (stF sw = stF ss))"

The second predicate extends this equality with bounded intruder knowledge to traces.

consts
boundedM :: "['a trace, 'a trace] ⇒ bool"
primrec
"boundedM [] ts = (ts = [])"
"boundedM (t#tw) ts = (**case** ts **of**
(t'#ts') ⇒ (trL t = trL t') ∧ (trST t ≡ ⊆M trST t') ∧
boundedM tw ts'
| [] ⇒ False)"

In the informal definition given before, we where comparing the traces of a protocol once executed under a weak intruder and once under a strong intruder. However, it is possible to define the predicate *weaker*, written as $weak \lesssim_w strong$, only with respect to the *rule_closure* of the two intruders, because only the *Net* introduction rule of *traces* changes, when changing the intruder.

Thus an intruder I_{weak} is *weaker* than an intruder I_{strong}, if for all steps (sw, sw') that I_{weak} can take there exists for all states ss bounding the intruder knowledge of sw a step (ss, ss') that I_{strong} can take and ss' bounds the intruder knowledge of sw'.[5]

constdefs
weaker :: "['a rule set, 'a rule set] ⇒ bool" ("_ ≲w _")
weaker_def : "weaker weak strong ≡ (
∀ kif. ∀ (sw, sw') ∈ rule_closure kif weak.
(∀ ss. (sw ≡ ⊆M ss) ⟶
(∃ ss'. (sw' ≡ ⊆M ss') ∧ (ss, ss') ∈ rule_closure kif strong)))"

This definition is *transitive* as expected, but *not reflexive*. A simple counter-example is a relation that is not monotonic in the intruder knowledge. However, for the intruders defined in our formalization, it is reflexive and the following hierarchy of intruders exists.

[5] Note that in this case "bounding the intruder knowledge" actually means "being equal with bounded intruder knowledge".

$$no\_intruder \lesssim_w passive\_intruder \lesssim_w wireless\_intruder \lesssim_w dolev\_yao$$

The main consequence of the above definition is the following lemma, which states that for all traces stemming from an execution of the protocol under the weaker intruder, there exists a trace which stems from an execution of the protocol under the strong intruder and which is equal with bounded intruder knowledge to the weak trace.

lemma *weaker_tracesD*:
"⟦ tw ∈ traces (env(|envNet:=weak|)) P;
weak $\lesssim_w$ strong ⟧ ⟹
∃ ts. boundedM tw ts ∧ ts ∈ traces (env(|envNet:=strong|)) P"

This lemma is used for example in 3.4.1 to prove the expected result that secrecy under a strong intruder implies secrecy under a weaker intruder. It can also be used to give a general transfer condition on when a property over traces remains invariant, if the intruder is being replaced with a weaker one.

lemma *transfer_to_weaker*:
assumes *weaker*: "weak $\lesssim_w$strong"
and *strong*: "∀ t ∈ traces (env(|envNet:=strong|)) P. Q t"
and *transfer*: "∀ tw ts. (boundedM tw ts ∧ Q ts) ⟶ Q tw"
shows "∀ t ∈ traces (env(|envNet:=weak|)) P. Q t"

3.3.6 Static Over-Approximation of the Intruder Knowledge

When proving security properties, we often need prove that certain terms *cannot* be inferred by the intruder. This is can be done by specifying an appropriate invariant and proving it by induction over the set of traces of a protocol. However there exists a simpler method for terms which cannot be inferred by the intruder, because they are not in the initial intruder knowledge and are never sent in an accessible position.[6] This can be done by defining a suitable over-approximation to the intruder knowledge.

A useful over-approximation to the intruder knowledge is *staticM*. We call it a static over-approximation, because it is not dependent on the concrete execution (and thus the execution environment), but uses only the information given by the protocol. Its main assumption is that all keys are known to the intruder. This assumption is captured by using $synth \circ parts$ instead of *infer* as an inference relation.[7] The introduction rule *InitM*, includes the initial intruder knowledge and the rule *Synth*, captures the inferences achievable by the intruder. The introduction rules *Inst*, captures the sending of a message. This message is instantiated with an instantiation whose variable map is constrained by the terms in *staticM*.

[6] An example for such a term is the private key of a trusted agent in the Needham-Schroeder-Lowe protocol.
[7] Remember that $infer\ \ kif\ \ M = synth\ (analz\ kif\ M)$ and $analz\ kif\ M \subseteq parts\ M$

consts *staticM* :: "'a proto ⇒ 'a runterm set"
inductive "staticM P"
intros
InitM: "x ∈ initM P ⟹ x ∈ staticM P"
Inst: "x ∈ {⟨Inst rid rMap vMap⟩(m) | rid rMap vMap m.
m ∈ msgs P ∧ fst ‘ ran vMap ⊆staticM P}
⟹ x ∈ staticM P"
Synth: "x ∈ synth (parts (staticM P)) ⟹ x ∈ staticM P"

This approximation is invariant under *parts*, *analz*, and *synth*. This implies that it is also invariant under *infer*.

lemma *parts_staticM* [*simp*]: "parts (staticM P) = staticM P"
lemma *analz_staticM* [*simp*]: "analz kif (staticM P) = staticM P"
lemma *synth_staticM* [*simp*]: "synth (staticM P) = staticM P"

Its main use is the following bound on the intruder knowledge of the Dolev-Yao intruder.

lemma *traces_static_bound_infer_M_DY*:
"⟦ tr ∈ set t; t ∈ traces (env(|envNet := dolev_yao|)) P; WF P ⟧
⟹ infer (protoKIF P) (stM (trST tr)) ⊆ staticM P"

Example: Private keys of trusted agents in Needham-Schroeder-Lowe protocol

The above over-approximation is used to prove the secrecy of private keys of trusted agents in the Needham-Schroeder-Lowe protocol.

lemma *trusted_sk_notin_staticM*:
"DApp ''sk'' (DAgent (Trusted a)) ∉ staticM nsl_proto"

The actual induction invariant used to prove this lemma is stronger. Due to the possibility of synthesizing hash function applications, if the hash function and its argument are known, we also have to prove that *DFunc* ''*pk*'' is secret.

lemma *trusted_sk_notin_staticM_lemma*:
"x ∈ staticM nsl_proto ⟹
({DApp ''sk'' (DAgent (Trusted a)), DFunc ''sk''} ∩ parts {x} = {})"

However, the proof itself is still much simpler than doing a full induction over the traces of the Needham-Schroeder-Lowe protocol.

3.4 Security Properties

The precise modeling of the execution of security protocols allows us to specify various security properties *independent* of the protocol under investigation. The connection between the protocol and a specific security property is made by the *claim events* of a role specification. In the traces, they are reflected as transitions labelled with this claim event. These transitions serve as witnesses for the claims.

Our initial focus for this formalization is on secrecy, because it is relatively simple and can be formalized as a single predicate. However, integrating authentication like it is described in [11] is planned as future work.

3.4.1 Secrecy

The predicate *SECRET env proto* (*SClaim l ''secret''* (*Some t*)) states that the given claim occurs in the protocol and for every occurrence of this claim in a trace of the protocol execution under the environment *env*, it takes this occurrences instantiation *inst* and checks that at no point in the trace $\langle inst \rangle(t)$ is in the intruder knowledge.

```
constdefs
  SECRET  :: "['a environment, 'a proto, 'a roleevent] ⇒ bool"
  SECRET_def : "SECRET env proto claim ≡ case claim of
    (SClaim l rC name optS) ⇒
     (claim ∈ protoEvs proto) ∧
     (case optS of
        Some S ⇒(
          ∀ t ∈ traces env proto.
           (∀ tr ∈ set t. ∀ rid rMap vMap st.
              ((TR (RunEvent (Inst rid rMap vMap, claim)) st = tr) ∧
               (ran rMap ⊆range Trusted))
               ⟶
              (∀ tr' ∈ set t.
                ⟨Inst rid rMap vMap⟩(S) ∉ infer (protoKIF proto) (stM (trST tr'))
              )
           ))
      | None ⇒False)
  | (_) ⇒ False"
```

Based on this definition of secrecy, it is possible to give a formal proof that proving secrecy under a strong intruder also implies secrecy under a weaker intruder.

```
lemma SECRET_transfer_intruder:
  "⟦ weak ≲w strong;
     SECRET (env(|envNet:=strong|)) P claim ⟧⟹
  SECRET (env(|envNet:=weak|)) P claim"
```

Furthermore, it is also possible to show that secrecy under a message type system implies secrecy under all stricter message type systems.

lemma *SECRET_transfer_typeflaws*:
```
"⟦ strict ⊆ lenient;
   SECRET (env(|envConf:=lenient|)) P claim ⟧⟹
SECRET (env(|envConf:=strict|)) P claim"
```

A secrecy proof of a term can also be reduced to a proof that this term is not contained in the static approximation to the intruder knowledge, we defined in 3.3.6. This is for example useful, if one wants to prove that the private key of an agent is secret, because it is never sent.

lemma *SECRET_by_staticM*:
```
"⟦ WF P;
   SClaim l rC name (Some rt) ∈ protoEvs P;
   ∀ t ∈ traces (env(|envNet := dolev_yao|)) P. ∀rid rMap vMap st.
    ((ran rMap ⊆range Trusted) ∧
     (TR (REv (Inst rid rMap vMap, SClaim l rC name (Some rt))) st ∈ set t))
     ⟶
    (⟨Inst rid rMap vMap⟩(rt) ∉ staticM P)
 ⟧ ⟹
 SECRET (env(|envNet:=dolev_yao|)) P (SClaim l rC name (Some rt))"
```

3.5 Protocol Verification

Verifying a security protocol means showing that it achieves its security objectives. In our model, this means that we show for every claim event contained in the protocol specification that it holds with respect to the chosen execution environment. Because of the independence between a protocol and the concrete predicate for a specific claim, we can define a single predicate *ALL_CLAIMS env P* which holds, if the given protocol P is correct under the given execution environment *env*.

3.5.1 Proving Protocol Correctness

The predicate *TRUECLAIM* is a technical construct which is used to connect all the different claim predicates into a single predicate working on all role events. For a send or a read event nothing has to be checked for correctness and thus the predicate holds. For a known claim event the respective predicate has to be verified and it is thus referenced. For an unknown claim event, the predicate does not hold, thus avoiding that a protocol verification succeeds although not all claims have been checked.

constdefs
TRUECLAIM :: "[string environment, string proto, string roleevent] ⇒ bool"
TRUECLAIM_def : "TRUECLAIM env proto ev ≡ **case** ev **of**
SClaim l rC name optS ⇒
if (name = ''secret'') **then** SECRET env proto ev
else False
| (_) ⇒ True"

The correctness of a protocol P under an execution environment *env* is then defined such that $TRUECLAIM\ env\ P$ holds for all role events *ev* of the protocol P.

constdefs
ALL_CLAIMS :: "[string environment, string proto] ⇒ bool"
ALL_CLAIMS_def : "ALL_CLAIMS env proto ≡
∀ r ∈ dom (protoRspecs proto).
∀ ev ∈ set (rspecEvs (the (protoRspecs proto r))).
TRUECLAIM env proto ev"

3.5.2 Showing the Existence of an Attack

Although our ultimate goal is protocol verification, it is important to verify the consistency of our model by exhibiting known attacks on specific protocols. To show that an attack on a protocol exists, one has to exhibit an attack trace. This can be done in two ways: Either, one uses the introduction rules of the inductive set of traces directly or one writes an executable trace which can then be translated to a 'normal' trace. The second method was developed, because our traces are quite verbose. It was used to prove the existence of the classic man-in-the-middle attack on the Needham-Schroeder protocol.

Executable Traces

The idea behind executable traces is that we want to provide a non-redundant and unambiguous format to specify a trace. This trace can then be computed by *simulating* the execution of the system guided by the executable trace. This simulation is performed using Isabelles rewriting facilities. In order to be able to specify all steps deterministically, some finite sets used during the execution are represented as lists.

Therefore we define a new datatype representing the system state during execution. This *executable state* uses a list to represent the finite sets of active runs.

record $'a\ xstate$ =
xstM :: "'a runterm set"
xstBS :: "'a runterm multiset"
xstBR :: "'a runterm multiset"
xstF :: "'a run list"

Also the role events are replaced by *executable events*. The only change is that the read event not only specifies the pattern which is used to read a message, but also the message itself.

datatype 'a *xevent* =
  XSend 'a "'a roleterm"
| *XRead* 'a "'a roleterm" "'a runterm"
| *XClaim* 'a 'a 'a "'a roleterm option"

Furthermore, transition labels are extended to incorporate the above changes. A create transition specifies now the role which is instantiated and the instantiation used. A run event specifies the run identifier and the executable event that is used. The network event specifies a network trace which represents the steps the intruder has taken.

datatype 'a *xtrans_label* =
  XInit
| *XCreate* 'a "'a inst" ("XCre")
| *XRunEvent* "'a runid" "'a xevent" ("XREv")
| *XNetwork* "'a xnet_trace" ("XNet")

Thus, an *executable trace* is a list of executable transition labels. Note the difference to a 'normal' trace which consists of a list of transition labels together with the resulting state. So by using executable traces to denote a trace, we can get rid of the list of states and we are also able to use a bit less verbose transition labels.

types 'a *xtrace* = "'a xtrans_label list"

What we need in the end is not an executable trace but a 'normal' trace. This normal trace can be computed by *simulating* the executable trace. There are quite a few technical definitions which are needed for this purpose. However, for the user it is enough to know about the partial function *simulate conf xnet P xtrace* which maps the executable trace *xtrace* to a 'normal' trace of the protocol P executed under the message type conformance relation *conf* and the intruder *xnet*. The domain of *simulate* are all executable traces that have a corresponding 'normal' trace.

consts
  simulate :: "['a conformance, 'a xnetwork, 'a proto]
    ⇒ 'a xtrace ⇀ 'a trace"
— Simulate an executable trace to get a normal trace

The main result concerning *simulate* is the following lemma. It states that, if we simulated an executable trace successfully, then the result is a trace of the protocol. As a side condition it has that the intruder of the execution environment must be at least as strong as the executable intruder used to simulate the executable trace.

lemma *traces_simulateI*:
  "⟦ Some t = simulate (envConf env) xnet P xt;
    weakerXIntr xnet (envNet env) ⟧
  ⟹ t ∈ traces env P"

Example: The Man-in-the-middle Attack on the Needham-Schroeder protocol

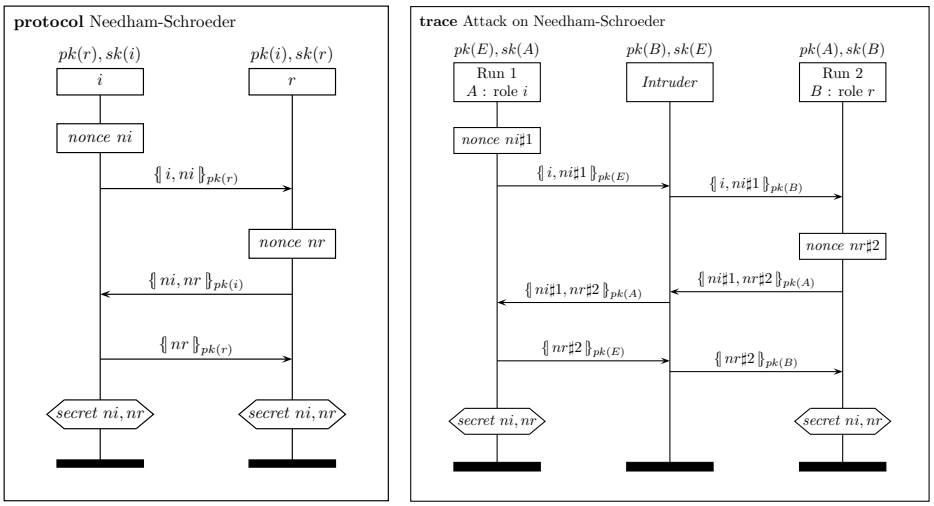

Figure 3.2: Nonce-secrecy for the Needham-Schroeder protocol and an attack on it.

The Needham-Schroeder protocol is the flawed predecessor of the Needham-Schroeder-Lowe protocol, whose formal definition can be found in 3.2.4. The only difference is in the second message in both roles, where Lowe added the role identifier of the responder to identify the sent nonce.

constdefs

```
ns_i      :: "string rolespec"     — The initiator role of the NS−Protocol
ns_i_def  : "ns_i      ≡ (|
   ...
   SRead "2" ⟨SRole "r", SRole "i",
              SEnc ⟨SConst "ni" None, SVar "V" VarT⟩ (SApp "pk" (SRole "i"))⟩
   ... |)"

ns_r      :: "string rolespec"     — The responder role of the NS−Protocol
ns_r_def  : "ns_r      ≡ (|
   ...
   SSend "2" ⟨SRole "r", SRole "i",
              SEnc ⟨SVar "W" VarT, SConst "nr" None⟩ (SApp "pk" (SRole "i"))⟩
   ... |)"
```

constdefs

```
ns_proto      :: "string proto"     — The NS−Protocol
ns_proto_def  : "ns_proto  ≡ (| protoKIF  = ["pk" ↦ "sk", "sk" ↦ "pk"],
                                protoRspecs = ["i" ↦ ns_i, "r" ↦ ns_r]      |)"
```

The central step in the proof that the secrecy claim for the responder nonce does *not* hold, is providing the concrete attack trace. This is done using the simulation lemma together with the attack trace specified as an executable trace.

```
lemma ns_proto_NOT_SECRET_DY:
  "¬ (SECRET ((|envConf=noTypeFlaws, envNet=dolev_yao|)) ns_proto
              (SClaim ''4'' ''r'' ''secret'' (Some (SConst ''nr'' None))))"
  ...
  apply(rule_tac xt="
  let ridA = (|ridAgent=Trusted ''A'', ridRole=''i'', ridId=0|);
      ridB = (|ridAgent=Trusted ''B'', ridRole=''r'', ridId=0|)
  in
  [
    XCre ''i'' (Inst ridA [''i'' ↦ Trusted ''A'', ''r'' ↦ Untrusted ''E''] empty),
    XREv ridA (XSend ''1''
      (⟨ SRole ''i'', SRole ''r'',
         SEnc ⟨SRole ''i'', SConst ''ni'' None⟩
           (SApp ''pk'' (SRole ''r'')) ⟩)
    ),
    XNet [
      (''take'', {|DAgent (Trusted ''A''), DAgent (Untrusted ''E''),
                   DEnc {|DAgent (Trusted ''A''), DConst ''ni'' ridA None|}
                     (DApp ''pk'' (DAgent (Untrusted ''E'')))|}),

      (''fake'', {|DAgent (Trusted ''A''), DAgent (Trusted ''B''),
                   DEnc {|DAgent (Trusted ''A''), DConst ''ni'' ridA None|}
                     (DApp ''pk'' (DAgent (Trusted ''B'')))|})

    ],
    XCre ''r'' (Inst ridB [''i'' ↦ Trusted ''A'', ''r'' ↦ Trusted ''B''] empty),
    XREv ridB (XRead ''1''
      (⟨SRole ''i'', SRole ''r'',
        SEnc ⟨SRole ''i'', SVar ''W'' VarT⟩ (SApp ''pk'' (SRole ''r'')) ⟩)
      ({|DAgent (Trusted ''A''), DAgent (Trusted ''B''),
                   DEnc {|DAgent (Trusted ''A''), DConst ''ni'' ridA None|}
                     (DApp ''pk'' (DAgent (Trusted ''B'')))|})
    ),
    XREv ridB (XSend ''2''
      (⟨SRole ''r'', SRole ''i'',
        SEnc ⟨SVar ''W'' VarT, SConst ''nr'' None⟩
             (SApp ''pk'' (SRole ''i'')) ⟩)
    ),
    XNet [
      (''take'', {|DAgent (Trusted ''B''), DAgent (Trusted ''A''),
                   DEnc {|DConst ''ni'' ridA None, DConst ''nr'' ridB None|}
                     (DApp ''pk'' (DAgent (Trusted ''A'')))|}),

      (''fake'', {|DAgent (Untrusted ''E''), DAgent (Trusted ''A''),
                   DEnc {|DConst ''ni'' ridA None, DConst ''nr'' ridB None|}
                     (DApp ''pk'' (DAgent (Trusted ''A'')))|})
    ],
    XREv ridA (XRead ''2''
      (⟨SRole ''r'', SRole ''i'',
        SEnc ⟨SConst ''ni'' None, SVar ''V'' VarT⟩ (SApp ''pk'' (SRole ''i'')) ⟩)
      ({|DAgent (Untrusted ''E''), DAgent (Trusted ''A''),
        DEnc {|DConst ''ni'' ridA None, DConst ''nr'' ridB None|}
             (DApp ''pk'' (DAgent (Trusted ''A'')))|})
    ),
    XREv ridA (XSend ''3''
```

```
    (⟨SRole ''i '', SRole ''r '',
      SEnc (SVar ''V'' VarT) (SApp ''pk'' (SRole ''r '')) ⟩)
  ),
  XNet [
    (''take '', {|DAgent (Trusted ''A''), DAgent (Untrusted ''E''),
                  DEnc (DConst ''nr'' ridB None)
                       (DApp ''pk'' (DAgent (Untrusted ''E'')))|}),

    (''fake '', {|DAgent (Trusted ''A''), DAgent (Trusted ''B''),
                  DEnc (DConst ''nr'' ridB None)
                       (DApp ''pk'' (DAgent (Trusted ''B'')))|})
  ],
  XREv ridB (XRead ''3''
    (⟨SRole ''i '', SRole ''r '',
      SEnc (SConst ''nr'' None) (SApp ''pk'' (SRole ''r '')) ⟩)
    ({|DAgent (Trusted ''A''), DAgent (Trusted ''B''),
      DEnc (DConst ''nr'' ridB None)
           (DApp ''pk'' (DAgent (Trusted ''B'')))|})
  ),
  XREv ridB (XClaim ''4'' ''r''  '' secret ''
              (Some (SConst ''nr'' None))
  )
]'' in traces_simulateI )
...
done
```

4 Related Work

4.1 Operational Semantics of Security Protocols

The formalization of the OSSP presented in 3 is based on the operational semantics of security protocols proposed in [11]. Our formalization is best seen as a *refinement* of this original description *removing all ambiguity*. We also simplified some constructions and provide several extensions both with respect to new constructions and with respect to proved properties of the system. Furthermore, several flaws could be identified in different theorems accompanying the original description.

4.1.1 Simplifications and Extensions

There are several small technical details that have been changed: The constraint on the type *roleknowledge* has been moved to the place of its only use, i. e., to the well-formedness constraints. The uniqueness assumptions the original description makes have also been made explicit and added to the well-formedness constraints. Also the intruder knowledge is not accessed by subscripting the trace, because this introduces more structure than what is needed for a partial ordering. The result being that a lot of boundary conditions for the indices have to be managed, which greatly reduces the possibilities of automation using Isabelles reasoning tools.[1]

Typing Model and Construction of Message Terms

The original description constructs message terms by partitioning a set of identifiers into different classes (e. g., roles, variables, functions). This construction is not suitable for a formalization, because it implies threading the partitioning information through all functions depending on the class of an identifier. Therefore, we are using a standard inductive datatype for our message terms. Furthermore, we also give an explicit way to specify keying systems and how to invert a key. This was left unspecified in the original description.

Another construction that was not suitable for a formalization was the very general typing model proposed in the original description. It was given as an arbitrary partition of the set of message terms. However the generality provided by this model is not needed. Therefore

[1] We were first using indexed access to traces. Some of the resulting theorems and the difficulties in their proofs can still be found in the theory files.

we decided to implement a more simple type system which essentially just records the constructor applications a term uses. We also extended the message term datatypes such that the type of a term can be computed without any additional knowledge to avoid threading type information through all functions.

Additional Functions on Message Terms

The original description knows only the two functions *subterms* and *infer*. However, those two functions were neither sufficient to formulate all lemmas we needed nor did they provide induction schemes powerful enough to proof the lemmas we could formulate. This is the reason why we introduced the operators *synth*, *analz*, and *parts*. They are adapted versions of the operators Paulson used in [21].

Another function we refined is the *match* function. For the original description, a specification of its properties was sufficient. However, we wanted have a constructive formulation and therefore we provide a primitive recursive definition of matching which has all the properties of the original description.

Static Over-Approximation of the Intruder Knowledge

In 3.3.6, we proposed a static over-approximation to the intruder knowledge. It gives a simple proof strategy for facts stating the intruder does not know a message, because it was never sent in an accessible position. It can be used as a replacement for the flawed lemma 3.25 proposed in the original description.

Meta-Theorems about Typeflaw Systems and Intruder Strengths

The original description assumed that verifying a protocol under a full type flaw system and a Dolev-Yao intruder will provide the strongest guarantees. We gave a formal definition on what it means for an intruder to be weaker than another one and formally showed that the above assumption is correct.

4.1.2 Problems Found

There have not been any fundamental problems. As a whole the original description corresponds to a high degree to the formalization developed in this thesis. However, we found several mistakes in different statements made.

Example 2.13 The precondition should read $\{m, k^{-1}\} \nvdash k$

Definition 2.16 (Readable Predicate) In the original definition, symmetric pairs are *always* readable, even if their subterms are not individually readable. Thus the intruder will not be able to read such pairs although an agent is. Our definition (see 3.2.3) rules this behavior out by only extending the knowledge with variables that can be inferred from the other half of a pair.

Lemma 3.25 This lemma essentially tries to give an over-approximation of the Intruder Knowledge. However, it is both flawed and applied wrongly in the paragraph after theorem 3.29. The private key of a trusted agent *is* a subterm of some sent message. Also is it possible to fulfill the premises by choosing the term t to be a variable not sent, but nevertheless instantiated to some sent term and thus known to the intruder at some point. Another counter-example is choosing the term t to be a pair of some term only known to the intruder and some term that will be sent. The static intruder knowledge approximation we proposed in this thesis does not have these problems.

Lemma 3.27 is flawed. Assume the intruder knows the key k, but he cannot construct the message m. Then as soon as the messages containing the rest of the message m have been sent, the intruder can construct and send $\{\!|\, m \,|\!\}_k$.

Proof of Theorem 3.29 This proof contains several wrong steps. Most of them stem from an imprecise dealing with *inferrability*.

All of these mistakes stem either from a wrong intuition about instantiations or a wrong intuition about inferrability. This makes clear where the most difficult parts of security protocol verification lie. It also emphasizes the need for machine checked proofs, which rule out these type of mistakes.

4.2 Paulsons Inductive Approach

In [21] Paulson has proposed to use a trace based semantics for security protocols. His approach - commonly referred to as "Paulsons Inductive Approach" - was extended by Bella in [5]. The Isabelle theories developed in Bellas extension are now part of the standard distribution of Isabelle. As the basis for the comparison between Paulsons Inductive Approach and our formalization, we will use the theories distributed with Isabelle 2005 and the textual description found in Bellas PhD thesis [5].

Paulson represents a protocol P as an inductively defined set T_P of traces. These traces can be interpreted as a possible history of events. The introduction rules for T_P comprise both the behavior of the intruder and the behavior of agents executing the protocol P. The agents are stateless and thus as soon as sending a message is possible (i. e., the premises of its introduction rule are satisfied), it will be sent indefinitely many times. Therefore Paulsons approach is an *over-approximation* with respect to the stateful execution of a protocol being executed in the real world.

The following example of the Needham-Schroeder-Lowe protocol [2] illustrates these statements:

```
consts nsl_public :: "event list set"
inductive nsl_public
  intros
        (* Initial trace is empty*)
   Nil: " [] ∈ nsl_public"

        (*The spy MAY say anything he CAN say. We do not expect him to
          invent new nonces here, but he can also use NS1. Common to
          all similar protocols.*)
  Fake: "⟦evsf ∈ nsl_public; X ∈ synth (analz (spies evsf))⟧
         ⟹ Says Spy B X # evsf ∈ nsl_public"

        (*Alice initiates a protocol run, sending a nonce to Bob*)
   NS1: "⟦evs1 ∈ nsl_public; Nonce NA ∉ used evs1⟧
         ⟹ Says A B (Crypt (pubEK B) {|Nonce NA, Agent A|})
              # evs1 ∈ nsl_public"

        (*Bob responds to Alice's message with a further nonce*)
   NS2: "⟦evs2 ∈ nsl_public; Nonce NB ∉ used evs2;
          Says A' B (Crypt (pubEK B) {|Nonce NA, Agent A|}) ∈ set evs2⟧
         ⟹ Says B A (Crypt (pubEK A) {|Nonce NA, Nonce NB, Agent B|})
              # evs2 ∈ nsl_public"

        (*Alice proves her existence by sending NB back to Bob.*)
   NS3: "⟦evs3 ∈ nsl_public;
          Says A B (Crypt (pubEK B) {|Nonce NA, Agent A|}) ∈ set evs3;
          Says B' A (Crypt (pubEK A) {|Nonce NA, Nonce NB, Agent B|})
            ∈ set evs3⟧
         ⟹ Says A B (Crypt (pubEK B) (Nonce NB)) # evs3 ∈ nsl_public"
```

Over-Approximation The only state used in this definition are the events having already happened in the trace. Thus as soon as *Says A' B* (*Crypt* (*pubEK B*){|*Nonce NA*, *Agent A*|}) is in a trace, the agent B will send indefinitely many answers. This over-approximates the real protocol, where B would answer exactly once and then wait for the third protocol message completing the protocol.

Intruder Strength The behavior of the intruder is given in the *Fake* rule. He can fake all messages he can infer (*synth* ∘ *analz*) from what he can spy from the trace (i. e., all sent messages and all notes taken by compromised agents). The definitions of *synth* and *analz* are such that this intruder is a Dolev-Yao type intruder.

[2] See 2.1 for an MSC of the Needham-Schroeder-Lowe protocol

Compared to our current formalization, Paulsons approach leads to simpler proofs. The three main reasons for this fact are the *simpler term structure*, the *stateless agents*, and the *direct definition of the protocol together with the intruder* he is using. We will describe the implications of this simplifications in the following sections.

4.2.1 Term Structure

The excerpt from the theory `Message.thy` shows how messages are built in Paulsons model. Keys are modeled as natural numbers. Agents are either the server, a friend identified by a natural number, or the spy. Messages are built as an inductive datatype.

types
key = *nat*

datatype —We allow any number of friendly agents
agent = *Server* | *Friend nat* | *Spy*

datatype
msg = *Agent agent* —Agent names
| *Number nat* —Ordinary integers, timestamps, ...
| *Nonce nat* —Unguessable nonces
| *Key key* —Crypto keys
| *Hash msg* —Hashing
| *MPair msg msg* —Compound messages
| *Crypt key msg* —Encryption, public− or shared−key

The above definitions state that encryptions can only use natural numbers as keys. Informally stated, this implies that the set of keys is "flat". The structure needed to get *provably different* public and private keys, long term shared keys between any two agents, and session keys computed from a hash value of previously sent message components is enforced by additional axioms. The theory `Public.thy` states the following two:

axioms
— No private key equals any public key
— (essential to ensure that private keys are private !)
privateKey_neq_publicKey [*iff*]: "privateKey b A ≠ publicKey c A'"

— All shared keys are symmetric
sym_shrK [*iff*]: "shrK X ∈ symKeys"

The theory `TLS.thy` contains these two further axioms:

axioms
— sessionK makes symmetric keys
isSym_sessionK: "sessionK nonces ∈ symKeys"

— sessionK never clashes with a long−term symmetric key
sessionK_neq_shrK [*iff*]: "sessionK nonces ≠ shrK A"

Introducing these axioms shortcuts the construction of an appropriate representation of keys, but risks introducing *inconsistencies*. Therefore without a formal soundness proof, the results obtained in Paulsons theory cannot be fully trusted.

Another problem of this construction is that the intruder must be strengthened depending on the operations used in the protocol. Whereas in the Needham-Schroeder-Lowe protocol the *Fake* rule was sufficient to model the intruder, the TLS protocol requires the following *additional* rule:

SpyKeys: —The spy may apply PRF and sessionK to available nonces
"⟦ evsSK ∈ tls;
{Nonce NA, Nonce NB, Nonce M} ⊆analz (spies evsSK) ⟧
⟹ Notes Spy {|Nonce (PRF(M,NA,NB)),
Key (sessionK((NA,NB,M),role)) |}# evsSK ∈ tls"

This implies that the specification of the intruder is *dependent* on the protocol under investigation.

4.2.2 Stateless Agents

Using stateless agents simplifies the structure of the inductive set of traces representing a protocols execution. This results in a simpler induction rule and therefore in easier proofs. However as we already explained this is an over-approximation with respect to the real execution of the protocol. As Moedersheim points out in [19] this over-approximation is sound for secrecy and for authentication, if it is formulated using witness events. However, it is an over-approximation and therefore not all properties of the original protocol can be proved. Typical examples are strong authentication properties like for example injective agreement. They cannot be verified, because a replay attack is always possible in Paulsons model.

4.2.3 No Formal Representation of a Protocol

In Paulsons approach, the intruder and the theorems stating the protocols correctness depend on the structure of the protocol. This is no problem as long as one just wants to investigate a *single protocol* in a *single environment*.

However this model is not sufficient if one is interested in *general properties* of security protocols. Formalizing results about compositionality, the effect of different execution environments, or higher-level proof strategies like trace patterns or a protocol logics is *not possible*. The reason is that a *mathematical representation* of security protocols and a semantic of their execution is missing.

4.2.4 Comparison

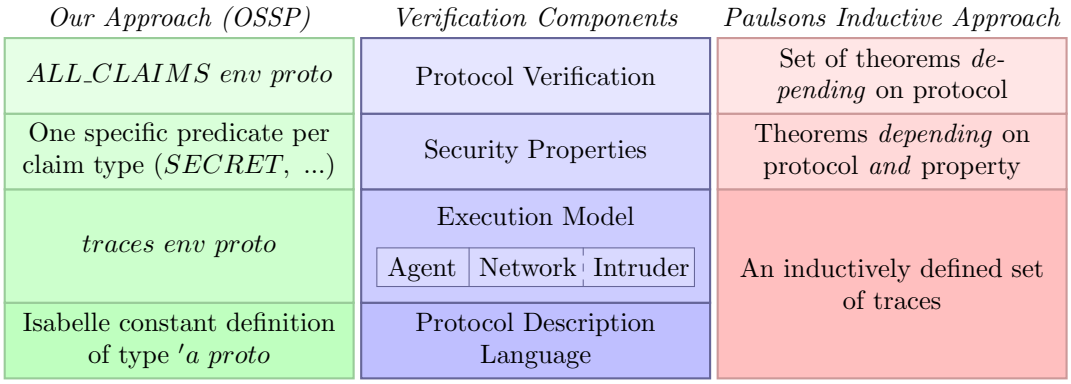

Figure 4.1: Comparison of verification components

With respect to Paulsons approach, our formalization comprises both an *extension* as well as a *refinement*. The term structure was extended such that there is no need to introduce additional axioms. Our theory was developed as a conservative extension of HOL, which guarantees its soundness with respect to HOL. Paulsons model was refined such that the definition of the intruder and the security objectives are independent of the protocol under investigation. This was achieved by defining a mathematical representation of a security protocol together with a semantic for its execution. Our model is stateful and therefore more security objectives can be formulated. A stateless model like Paulsons could also be formulated on top of our representation of security protocols. This is a possible means to simplify proofs in our model and remains future work.

4.3 Protocol Composition Logic

The Protocol Composition Logic (PCL) was proposed by Datta, Derek, Mitchell, and Roy in [13]. It is a logic for proving security properties of network protocols that use public and symmetric key cryptography. It is designed around a process calculus with actions for possible protocol steps including generating new random numbers, sending and receiving messages, and performing decryption and digital signature verification. The proof system consists of axioms about individual protocol actions and inference rules that yield assertions about protocols composed of multiple steps.

PCL has two main advantages: First, proofs in PCL are concise and second, the logic can be used to describe the invariants an execution environment must fulfill to guarantee a protocols security objectives. This allows for compositional proofs for protocols composed of individual sub-protocols. Verifying the sub-protocols independently and putting them together afterwards while checking the required invariants reduces the verification effort considerably.

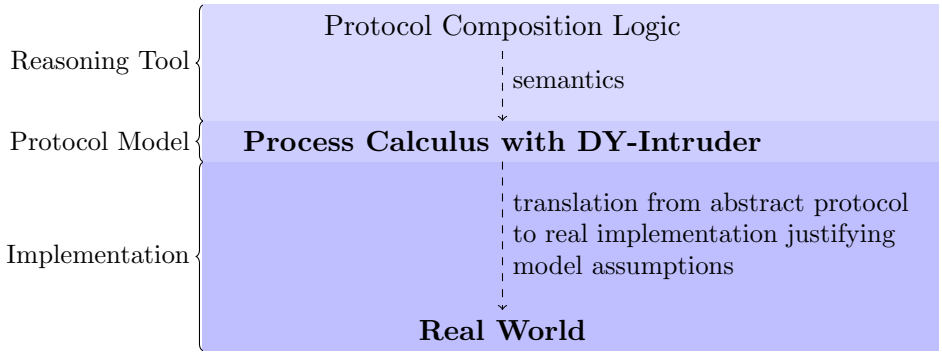

Figure 4.2: Relation between PCL and the real world.

When applying formal methods, it is important to make the assumptions under which a verification holds explicit.[3] In the case of a logic, this is done by providing a semantics for the logic. In figure 4.2 we see that for a protocol logic, we have to provide a semantic interpretation function of its well-formed formulas with respect to a protocol model based on justified assumptions. If this semantics is not sound, then it would be possible to proof properties an execution of the protocol does not have. Thus providing a sound semantics for PCL is of utmost importance to guarantee its sound use as a reasoning tool for protocol verification. However, because of the issues explained below, we don't believe that the currently given semantics and its soundness proof [13, appendix B] are correct and guarantee this sound use.

One problem we noted is that in [13, subsection 3.2] the interpretation of the modal formula $\mathcal{Q}, R \vDash \phi_1[P]_A\phi_2$ is defined as

> $\mathcal{Q}, R \vDash \phi_1[P]_A\phi_2$ holds, if $R = R_0R_1R_2$, for some R_0, R_1, and R_2, and either P does not match $R_1|_A$ or P matches $R_1|_A$ and $\mathcal{Q}, R_0 \vDash \sigma\phi_1$ implies $\mathcal{Q}, R_0R_1 \vDash \sigma\phi_2$, where σ is the substitution matching P to $R_1|_A$.

This interpretation can be trivially satisfied for a non-empty sequence of actions P by choosing R_1 as the empty run []. Therefore, the obviously false formula $\top[\texttt{send}(x)]_A\bot$ is evaluated to true. Together with axiom **G3**, it is possible to derive any formula, which renders the system inconsistent.

A further problem is *variable binding* in PCL formulas. All variables assigned inside the actions P of the modal formula $\phi_1[P]_A\phi_2$ are bound in the formulas ϕ_1 and ϕ_2. The axioms of type **AR1** are used to deal with bound variables stemming from matching, signature verification, and decryption. However, there are no such axioms for the actions `new` and `receive`, although they also introduce new variable bindings. This is possibly the cause of the following error:

[13] - Table 5 - Step (13) Here, the axiom $\mathbf{AA1}_{\texttt{new}}$ is applied as $\top[\texttt{new}(x)]_A New(A, m)$, instead of using the correct version $\top[\texttt{new}(x)]_A New(A, x)$ with the bound variable x. The non-bound version used here would imply that in every run that the variable x

[3] See 5.1.1 for a discussion of the modeling assumptions we are making in our formalization.

is assigned a fresh value in, we will have some variable y that is assigned the value m and m is fresh. This is obviously not true and possibly misses a replay attack.

It is also surprising that in the two example proofs in [13, section 5] only once (table 6 step 21 using **AR2**) reasoning about a bound variable is necessary, although the protocol role we are reasoning about contains three binding occurrences of variables.[4] In table 6, the received message is just substituted at all places where its variable would occur, although there is no inference rule justifying this step for the `receive` action. It is possible that one misses important case distinctions with respect to the origin of a message, when substituting the content of a received message without careful consideration.

Because of the above problems, we would not recommend using PCL in its current state for concrete protocol verifications. Otherwise, one risks verifying a *flawed* protocol to be correct, as it was possible in the case of the Needham-Schroeder protocol [17] and the BAN logic [7, 9]. There are two areas where PCL should be improved: First, a precise formal semantics is needed together with a full soundness proof to guarantee that PCL is based on a sensible protocol model. Second, the user of PCL should be supported more strongly when reasoning about bound variables. The best solution would be to provide mechanized tool support.[5]

One way to solve the above problems would be to to define a semantic interpretation function of PCL on top of our formalization. This should be possible, because the process calculus model PCL is currently based on is not very different from our trace based model. The result would be a conservative embedding of PCL in Isabelle/HOL, which implies soundness of PCL with respect to the OSSP by construction.

[4] not counting the variables needed in the implicit matching.

[5] The preliminary tool support provided in the form of Isabelle/HOL theories containing an axiomatized version of PCL downloadable from the authors website at `http://www.andrew.cmu.edu/user/danupam/logic-derivation.html` does not solve this problem. These theories do not allow to reason about substitution.

5 Conclusions

5.1 Discussion

5.1.1 Relation to the Real World

When applying formal methods, the problem relevant part of the real world is selected and modeled. The decision on what is relevant and what is not lies with the person modeling the problem. It is also her responsibility to justify the abstractions she is making. If these justifications are correct, then we can safely assume the *closed world assumption*. It states that a solution in the abstract world represents a solution in the real world.

The main implication of the closed world assumption for our model, is that there is no other communication between agents than the one specified by the protocol under investigation. This is certainly not true in general, because on todays networks several protocols are run in parallel. However by using a tagging scheme like it is proposed in [1], it is safe not to consider interaction between different protocols. We also do not consider interaction of the type that is used in social engineering, because it is not a problem of protocol verification. On the level of our formalization, we also do not consider possible errors like buffer overflows or other implementation problems, because the problem of getting a correct implementation of an abstract security protocol can be separated from the problem of verifying this abstract protocol.

Like many other approaches we are doing black-box security protocol verification in a Dolev-Yao type model [14]. We assume that *cryptography is perfect*. This means that an encryption can only be decrypted, if the inverse key is known. We also assume that messages can be modeled as terms of a free algebra. This *free algebra assumption* means that two terms are equal, if and only if they are syntactically equal. This implies that we cannot currently model algebraic operations like *xor* or *Diffie-Hellman exponentiation*. It also implies that the bit strings representing a real world message must contain enough information to reconstruct the term structure.

These two assumptions have a cryptographic justification as the work of Backes, Pfitzmann, and Waidner shows [3]. Their abstract cypto-library and its related proofs have been formalized in Isabelle/HOL by Sprenger in [22]. Connecting our work with his formalization, will allow us to formally justify these two assumptions. This is planned as future work.

5.1.2 Design Decisions

All constructs described in the original description had to be mapped to Isabelle/HOL constructs during this formalization. This mapping is not unique and therefore several design discussions had to be made. They have not been discussed in the description of our formalization, because we wanted the reader to have a clear view onto the *current* formalization.

Most of these decisions concern the structure of traces. Traces are *growing at the front*, because the first element has to be extracted often and this is supported much better at the front than at the tail. Traces do also contain the full system state after *every transition*. A less verbose representation would be to represent traces as a pair of a list of transitions and the last system state. However this requires the transition labels to uniquely identify the trace, because otherwise not all intermediate states could be reconstructed. This is not the case with our current labels and therefore we decided to include the state after every transition. This has also the advantage that notational brevity is gained when accessing intermediate states.

In the original description, traces have been accessed by indexing its elements. On our first attempt to proving general properties of traces, we also used this approach. However it results in a lot of unnecessary side conditions, because the structure of the integers is much richer than what is needed for the partial ordering we are interested in. Therefore we now use bounded quantification over the elements of the list, i. e., $\forall\ x \in set\ \ t.\ \ P\ x$. Equality over concatenated lists i. e., $\forall\ tp\ \ ts.\ \ t = ts\ @\ tp \longrightarrow Q\ tp\ ts$, is used to decompose a trace t into a trace prefix tp and a trace suffix ts. This approach gives us better automation than the indexed approach.

Another difference between the original description and our formalization is the modeling of the network controlled by the intruder. Whereas in the original description intruders are modeled in an operational fashion, we are modeling them using a denotational semantics. The reason is that we wanted the comparison of traces stemming from different intruders to be as simple as possible. For this comparison only the total effect of the intruders actions is interesting and therefore modeling the intruder with an operational semantics would have required us to compact the trace in some manner. However we did not see any relevant properties depending on the precise operation of the intruder and therefore we implemented this compaction directly by using a denotational semantics for the intruder.

5.1.3 Issues to be Resolved

Well-Formedness and Implementability The original description contained only a well-formedness predicate. Its intent was also partially distributed trough out the text of the description. As we already mentioned in 3.2.3 this predicate should be partitioned into *well-formedness* ensuring basic properties of the protocol definition and *implementability* ensuring that a protocol could be implemented by agents having to infer all needed terms. The main use of implementability is guaranteeing that the

intruder is able to impersonate an untrusted agent. The concrete definitions of these two predicates have yet to be investigated.

Set of Runs The function *runsOf* could possibly generate runs with instantiations whose role map does not assign an agent to all roles used in the role specification of a well-formed protocol. This only happens for protocols whose role specifications contains roles that the protocol does not possess an associated role specification for. It remains to decide, if this is a well-formed protocol at all.

Initial Intruder Knowledge Currently all terms of the initial role knowledge of an untrusted agent containing a constant at some position are excluded from the initial intruder knowledge. If this knowledge contains a pair *SPair* (*SConst c*) (*SFunc key*) then this behavior results in a too weak knowledge of the intruder, disallowing him possibly to impersonate an untrusted agent. This problem will be dealt with upon the definition of implementability.

Security Objectives The predicates checking the validity of a protocols claim should be as restrictive as possible to avoid wrong protocol verifications. The predicate *SECRET* does currently not check the name of the claim it is given. Also when giving it a pair, it checks if *any* of its components is secret. However, we would expect it to check for the secrecy of all of its components. This will be changed in the future.

Well-Typedness We modeled substitutions such that they can be checked for well-typedness. However, we do not see any useful applications of this predicate. Therefore we possibly should remove it, because the current structure of the substitutions unnecessarily complicates the proofs.

analz The current formulation of *analz* is lacking important computational rules. Reasoning about *analz kif* (*insert x M*) is extremely tedious. The main problem is that x cannot be moved out of *analz*, because it could be needed as a key. This is the reason why we did not complete the proof of the nonce-secrecy of the Needham-Schroeder-Lowe protocol in the given time constraint. However, we have found a generalized version of *analz* which possesses the required rules. It is called *analz2* and currently defined in the theory `nsl_proto.thy`.

consts
analz2 :: "['a inverse, 'a runterm set, 'a runterm set] ⇒ 'a runterm set"
inductive "analz2 kif K M"
intros
Inj: "x ∈ M ⟹ x ∈ analz2 kif K M"
Fst: "{|x,y|} ∈ analz2 kif K M ⟹ x ∈ analz2 kif K M"
Snd: "{|x,y|} ∈ analz2 kif K M ⟹ y ∈ analz2 kif K M"
Decrypt: "⟦ DEnc x k ∈ analz2 kif K M;
kif(|k|) ∈ synth (K ∪ analz2 kif K M) ⟧
⟹ x ∈ analz2 kif K M"

This function is parametrized over the key inversion function, a set of keying material to be used when decrypting, and the knowledge that should be analyzed. The difference with respect to *analz* is the additional set of keying material.[1] This set enables us to prove the following computational rules:

```
(∗ This rule exists analogously for all other literals ∗)
"analz2 kif K (insert (DRole r rid) M) =
   insert (DRole r rid) (analz2 kif (insert (DRole r rid) K) M)"

"analz2 kif K (insert {|x,y|} M) =
   insert {|x,y|} (analz2 kif K (insert y (insert x M)))"

"analz2 kif K (insert (DEnc m k) M) =
   insert (DEnc m k) (
     if (kif(|k|) ∈ synth (K ∪ analz2 kif (insert (DEnc m k) K) M))
     then (analz2 kif (insert (DEnc m k) K) (insert m M))
     else (analz2 kif (insert (DEnc m k) K) M)
   )"
```

We plan to adapt our formalization to use this generalized version. This will enable us to complete the proof of the nonce-secrecy of the Needham-Schroeder-Lowe protocol.

5.2 Contributions

Our main contribution is a strictly formal protocol model in the form of a formalization of the operational semantics of security protocols proposed in [11]. This formalization clarifies and extends the original description in several points.

- It contains a *simpler type system* which made it possible to give an *explicit definition of matching*.
- We gave a precise statement on what it means for a type system to be more lenient and what *influence the type system has on the execution* of a protocol.
- A better understanding of the *infer* relation has been provided by splitting it into appropriate downward (*analz*) and upward closures (*synth*).
- We defined a *static approximation to the intruder knowledge* which helps establishing facts about terms which are never sent in an accessible position.
- We proposed a notion of what it means for an intruder to be weaker than another and showed formally that *secrecy for a strong intruder implies secrecy for a weak intruder.*
- We checked that the man-in-the-middle attack on the Needham-Schroeder protocol exists in our model.

Our formalization is the first unambiguous description of a semantics of security protocols which contains a precise execution model. Having such a precise description will enable a

[1] Obviously *analz* is a special case of *analz2*, because $analz\ kif\ M = analz2\ kif\ \{\}\ M$.

deeper understanding of the nature of security protocols. Furthermore, having an *unambiguous* description also makes communicating results much easier. This formalization is therefore best seen as a *solid basis* for further developments in security protocol verification. Its main benefits lie in *enabling* all the possible extensions we list in the next section.

5.3 Future Work

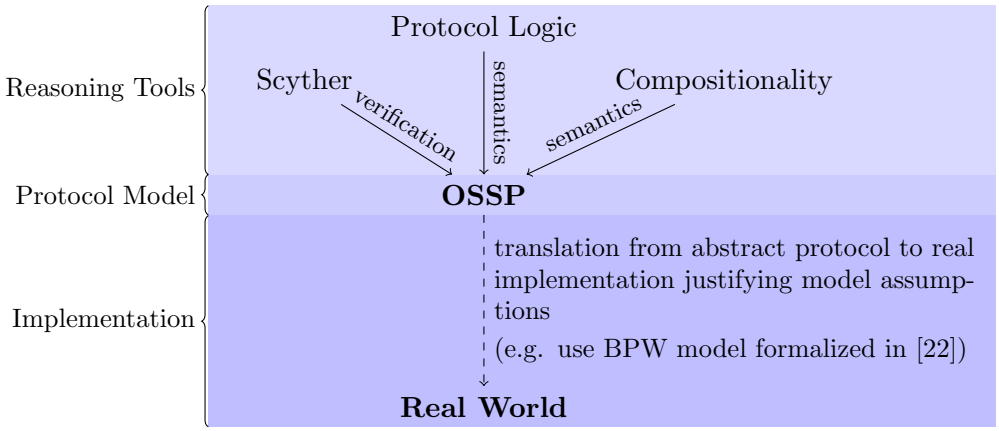

Figure 5.1: A strictly formal protocol verification and construction technique.

In order to check our protocol model, one goal of this thesis was also to verify the nonce secrecy of the Needham-Schroeder-Lowe protocol. However, due to time constraints, we did not achieve this goal. Finishing this proof together with resolving the issues mentioned in 5.1.3 will make this protocol model a sound mechanized basis for security protocol verification.

Our ultimate goal is to develop a sound and strictly formal protocol *verification* and *construction* technique on top of this basis. In order to achieve this, there are three main lines of future work: First, we want to broaden the scope of our method. This includes extending the protocol modeling language such that more real world protocols can be modeled in our framework and providing a translation from an abstract protocol to a real world implementation justifying the model assumptions. Second, we would like to introduce stronger reasoning tools to enable a more efficient verification. Third, we would like to provide automation by integrating automated decision procedures like Scyther. The details of these directions will be discussed in the following sections.

5.3.1 Broadening the Scope

Extending the Protocol Description Language

The set of protocols we can currently model with our protocol description language is restricted by the lack of algebraic operations *xor* or *Diffie-Hellman exponentiation* as well as

by the lack of control structures. We would like to extend our term structure to incorporate these components as well.

Implementation of Protocols

What we ultimately want is a protocol which is executable in the real world and achieves the desired security objectives. This goal can obviously never be fully achieved, because the question, if a real world protocol is correct, is not well-posed. This question can be made well-posed by providing an abstract model of a protocols execution together with a protocol description language which allows us to model the real world protocol as precise as possible. If we now also defined a translation from this protocol description language to a real world implementation, we would be able to produce real world protocols with a high probability of being correct. A first step towards this goal would be to connect our work with the formalization Sprenger has done in [22]. Defining such a translation and showing that this translation justifies the assumptions of our model is one line of future work.

Adding more Security Properties

Currently, only secrecy is defined in our formalization. We would like to add the various forms of authentication as they are proposed in the original description. Furthermore, we would like to investigate how other properties like non-repudiation or shared key establishment can be formalized in our model.

Extending the Parametrization of the Execution Model

Model checkers often bound the number of runs they are considering when checking a protocol. For some protocols it is possible to show that this approach is sound. By parameterizing our execution model also over the set of allowed runs, we can investigate these statements formally. This would allow us to use bounded model checkers as sound proof procedures in our formalization.

Another benefit of this parametrization of our execution model would be the integration of the results of [10]. They state that to check secrecy, it suffices to consider all runs of one honest and one dishonest agent. A more general result they also state is that for a security objective ϕ whose formula involves k agent variables, $k + 1$ agents will suffice. This results enables a reduction in the search space of concrete proofs of a security objective.

Explore Additional Relations Between Different Execution Models

In 3.3.5, we defined a notion of when an intruder is weaker than another intruder. The definition given there is based on a direct relation between the traces generated under the two different intruders. However, the really important relation between the two intruders

is their strength in breaking different security objectives. The idea is that an intruder is *weaker with respect to security objective* ϕ than another intruder, if ϕ under the weak intruder implies ϕ under the strong intruder. As this relation is more general than the one we have currently, having these relations would allow us to chose from a greater set of intruder models when integrating automated protocol verification algorithms. A possible candidate for inclusion is the Machiavelli intruder proposed in [23].

5.3.2 Providing more Abstraction

Compositionality

Real world protocols are almost always too big to be verified directly. However, they are often constructed from various sub-protocols which are amenable to direct verification. To construct a correctness proof for the composed protocol we need to be able to reason about various compositions (e. g., parallel or sequential execution) of protocols. A first step towards the goal of compositional verification could be to formalize the work done in [1]. As the flawed lemmas from section 3 of the original description of the OSSP have shown, it is likely that such a formalization would highlight problems in the proofs. Solving these problems would result in mechanically verifiable conditions on the composability of security protocols.

Protocol Logic

In [13] Mitchell e.a., proposed PCL, a protocol logic which allows to reasons about security protocols without explicit reasoning about the intruder. We would like to investigate the existence and strength of such a logic whose semantics is based on our model. This would combine the strength of reasoning PCL seems to provide with the strong soundness guarantees that a conservative embedding of a logic in HOL gives.

Security Property Specification

Currently a protocol designer can only verify security properties that have already been specified. The set of verifiable properties can only be extended by providing a new constant definition specifying the required property in terms of HOL. This is certainly nothing every protocol designer can do. By embedding a temporal logic like for example Past LTL on top of our traces, we could make specifying and possibly also proving security properties easier.

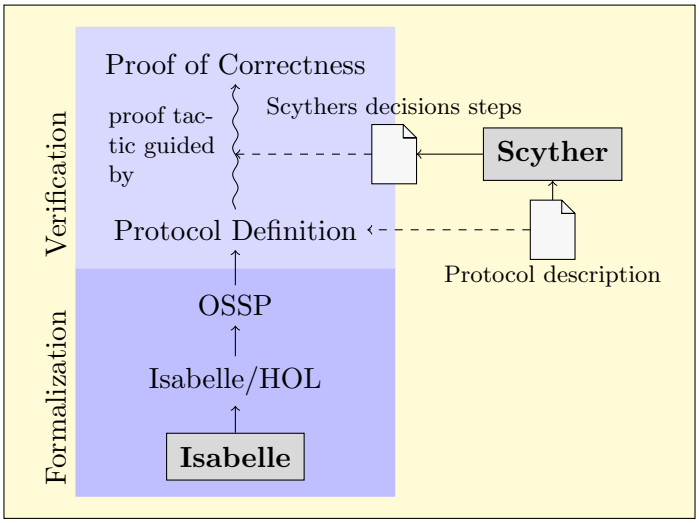

Figure 5.2: Integration of Scyther into Isabelle/HOL

5.3.3 Automation

Scyther [11] is a state-of-the-art tool for the automatic verification of security protocols. It is based on the same OSSP as this formalization is. We plan to integrate Scyther such that we get full automation for a majority of protocols, i. e., all protocols Scyther can do unbounded verification for. The structure of this integration can be seen in figure 5.2. The main step towards this integration will be the formalization of trace patterns and the theorems capturing the decision steps of Scythers verification algorithm. (See B.1 for a rough estimate of the needed effort.) These theorems can then be "plugged" together according to the proof steps Scyther has taken in a concrete protocol verification. This automation together with formalized results about compositionality will greatly increase the size of protocols that can be mechanically verified.

A Additional Definitions

A.1 Relations and Functions on Runterms

See 3.2.2 for an explanation of their intent.

consts *varsOf* :: "'a runterm ⇒ 'a set"
primrec
"varsOf (DVar v rid ty) = {v}"
"varsOf (DConst c rid uty) = {}"
"varsOf (DIntrConst c uty) = {}"
"varsOf (DRole r rid) = {}"
"varsOf (DAgent a) = {}"
"varsOf (DFunc f) = {}"
"varsOf (DApp f arg) = varsOf arg"
"varsOf (DPair x y) = varsOf x ∪ varsOf y"
"varsOf (DEnc x k) = varsOf x ∪ varsOf k"

consts *subterms* :: "'a runterm set ⇒ 'a runterm set"
inductive "subterms M"
intros
Inj [*intro*]: "x ∈ M ⟹ x ∈ subterms M"
Fst: "DPair x y ∈ subterms M ⟹ x ∈ subterms M"
Snd: "DPair x y ∈ subterms M ⟹ y ∈ subterms M"
Msg: "DEnc x k ∈ subterms M ⟹ x ∈ subterms M"
Key: "DEnc x k ∈ subterms M ⟹ k ∈ subterms M"
App: "DApp n arg ∈ subterms M ⟹ arg ∈ subterms M"

consts *parts* :: "'a runterm set ⇒ 'a runterm set"
inductive "parts M"
intros
Inj [*intro*]: "x ∈ M ⟹ x ∈ parts M"
Fst: "DPair x y ∈ parts M ⟹ x ∈ parts M"
Snd: "DPair x y ∈ parts M ⟹ y ∈ parts M"
Msg: "DEnc x k ∈ parts M ⟹ x ∈ parts M"

consts *synth* :: "'a runterm set ⇒ 'a runterm set"
inductive "synth M"
intros

Inj [*intro*]: "x ∈ M ⟹ x ∈ synth M"
Pair [*intro*]: "⟦ x ∈ synth M; y ∈ synth M ⟧⟹ {|x,y|} ∈ synth M"
Enc [*intro*]: "⟦ x ∈ synth M; k ∈ synth M ⟧⟹ DEnc x k ∈ synth M"
App [*intro*]: "⟦ DFunc n ∈ M; arg ∈ synth M ⟧⟹ DApp n arg ∈ synth M"

constdefs
invertKey :: "['a inverse, 'a runterm] ⇒ 'a runterm" ("_(|_|)")
— Compute the inverse of a key under a given keying scheme
invertKey_def :
"invertKey kif key ≡ **case** key **of**
(DApp f arg) ⇒(**case** kif f **of**
Some f' ⇒ DApp f' arg
| None ⇒ DApp f arg)
| (_) ⇒ key"

constdefs
keysFor :: "['a inverse, 'a runterm set] => 'a runterm set"
—Keys useful to decrypt elements of a message set
"keysFor kif M ≡ invertKey kif ` {k. ∃x. DEnc x k ∈ M}"

consts *infer* :: "['a inverse, 'a runterm set] ⇒ 'a runterm set"
inductive "infer kif M"
intros
Inj [*intro*,*simp*] : "x ∈ M ⟹ x ∈ infer kif M"
Fst [*intro*]: "{|x,y|} ∈ infer kif M ⟹ x ∈ infer kif M"
Snd [*intro*]: "{|x,y|} ∈ infer kif M ⟹ y ∈ infer kif M"
Decrypt [*dest*]:
"⟦ DEnc x k ∈ infer kif M; kif(|k|) ∈ infer kif M ⟧⟹ x ∈ infer kif M"
Pair [*intro*]: "⟦ x ∈ infer kif M; y ∈ infer kif M ⟧⟹ {|x,y|} ∈ infer kif M"
Enc [*intro*]: "⟦ x ∈ infer kif M; k ∈ infer kif M ⟧⟹ DEnc x k ∈ infer kif M"
App [*intro*]:
"⟦ DFunc f ∈ infer kif M; arg ∈ infer kif M ⟧⟹ DApp f arg ∈ infer kif M"

consts
analz :: "['a inverse, 'a runterm set] ⇒ 'a runterm set"
inductive "analz kif M"
intros
Inj: "x ∈ M ⟹ x ∈ analz kif M"
Fst: "{|x,y|} ∈ analz kif M ⟹ x ∈ analz kif M"
Snd: "{|x,y|} ∈ analz kif M ⟹ y ∈ analz kif M"
Decrypt: "⟦ DEnc x k ∈ analz kif M; kif(|k|) ∈ synth(analz kif M) ⟧
⟹ x ∈ analz kif M"

Accessing the Fields of an Instantiation

constdefs
instRID :: "'a inst ⇒ 'a runid"
instRID_def : "instRID inst ≡ **case** inst **of** (Inst rid roleMap varMap) ⇒rid"

instRMap :: "'a inst ⇒ ('a ⇀ 'a agent)"
instRMap_def :
"instRMap inst ≡ **case** inst **of** (Inst rid roleMap varMap) ⇒roleMap"

instVMap :: "'a inst ⇒ 'a dvarsubst"
instVMap_def :
"instVMap inst ≡ **case** inst **of** (Inst rid roleMap varMap) ⇒varMap"

HOL List Extension

consts
before :: "'a list ⇒ 'a ⇒ 'a ⇒ bool"
primrec
"before [] a b = False"
"before (x#xs) a b = (**if** (x=a) **then** (b ∈ set xs) **else** (before xs a b))"

constdefs
beforeOrd :: "'a list ⇒ ('a × 'a) set"
beforeOrd_def : "beforeOrd xs ≡ {(x,y) | x y. before xs x y}"

B Theory Dependencies

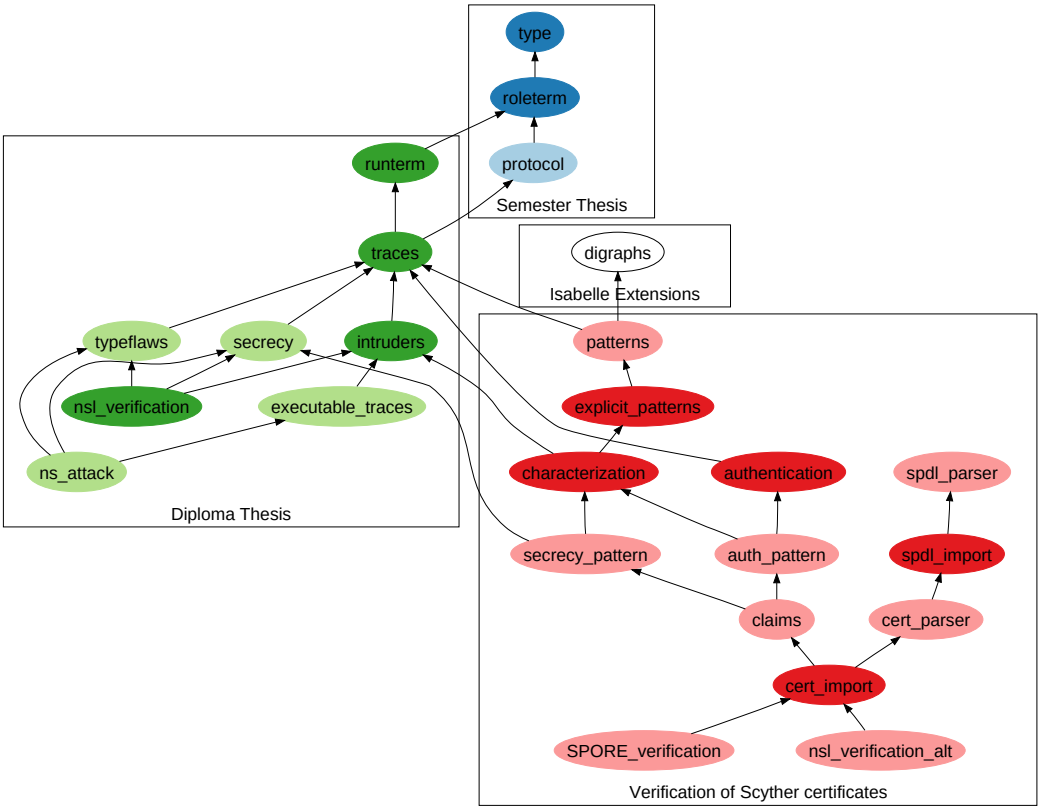

Figure B.1: Theory dependencies

Figure B.1 shows the theory dependencies that exist in our formalization. It also gives a rough estimate of the effort needed to integrate Scyther.

Bibliography

[1] S. Andova, C.J.F. Cremers, K. Gjøsteen, S. Mauw, S.F. Mjølsnes, and S. Radomirović. Sufficient conditions for composing security protocols. *Information and Computation*, 2007. To appear.

[2] P.B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Academic Press, Orlando, May 1986. ISBN 0120585367. URL `http://gtps.math.cmu.edu/andrews.html`.

[3] M. Backes, B. Pfitzmann, and M. Waidner. A universally composable cryptographic library, 2003. URL `http://citeseer.ist.psu.edu/backes03universally.html`.

[4] D. Basin, S. Mödersheim, and L. Viganò. OFMC: A symbolic model checker for security protocols. *International Journal of Information Security*, 4(3):181–208, 2005. URL `http://www.springerlink.com/index/10.1007/s10207-004-0055-7`.

[5] G. Bella. *Inductive Verification of Cryptographic Protocols*. PhD thesis, University of Cambridge, March 2000. URL `http://citeseer.ist.psu.edu/article/bella00inductive.html`.

[6] B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.

[7] A. Bleeker and L. Meertens. A semantics for BAN logic. In *Proc. DIMACS Workshop on Design and Formal Verification of Security Protocols*, 1997.

[8] Achim D. Brucker and Burkhart Wolff. The HOL-OCL book. Technical Report 525, ETH Zurich, 2006. URL `http://www.brucker.ch/bibliography/abstract/brucker.ea-hol-ocl-book-2006`.

[9] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, 1990.

[10] H. Comon-Lundh and V. Cortier. Security properties: Two agents are sufficient. URL `http://citeseer.ist.psu.edu/article/comon-lundh04security.html`.

[11] C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Ph.D. dissertation, Eindhoven University of Technology, 2006.

[12] C.J.F. Cremers. The Scyther tool: Automatic verification of security protocols, 2007. URL `http://people.inf.ethz.ch/cremersc/scyther/index.html`.

[13] A. Datta, A. Derek, J. C. Mitchell, and A. Roy. Protocol composition logic (pcl). *Electron. Notes Theor. Comput. Sci.*, 172:311–358, 2007. ISSN 1571-0661. doi: http://dx.doi.org/10.1016/j.entcs.2007.02.012. URL `http://www.andrew.cmu.edu/user/danupam/ddmr-pcl06.pdf`.

[14] D. Dolev and A.C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(12):198–208, March 1983.

[15] J. Heather, G. Lowe, and S. Schneider. How to prevent type flaw attacks on security protocols. *Journal of Computer Security*, 11(2):217–244, 2003.

[16] ITU-TS. *Recommendation Z.120: Message Sequence Chart (MSC)*. ITU-TS, Geneva, 1999.

[17] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proc. 2nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer, 1996.

[18] S. Meier. Formalizing an operational semantics of security protocols; Part I: Static Protocol Descriptions. ETH Semester Thesis, 2007. URL `http://n.ethz.ch/student/meiersi/projects/foosp`.

[19] Sebastian Mödersheim. On the relationships between models in protocol verification (extended version). Technical Report 512, Infsec, ETH Zuerich, 03 2006.

[20] T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002. URL `http://www4.in.tum.de/~nipkow/LNCS2283/`.

[21] L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998. URL `http://citeseer.ist.psu.edu/paulson00inductive.html`.

[22] C. Sprenger, M. Backes, D. Basin, B. Pfitzmann, and M. Waidner. Cryptographically sound theorem proving. In *CSFW '06: Proceedings of the 19th IEEE Workshop on Computer Security Foundations*, pages 153–166, Washington, DC, USA, 2006. IEEE Computer Society. doi: 10.1109/CSFW.2006.10. URL `http://portal.acm.org/citation.cfm?id=1155674`.

[23] P. Syverson, C. Meadows, and I. Cervesato. Dolev-yao is no better than machiavelli. URL `http://citeseer.ist.psu.edu/syverson00dolevyao.html`.