Aalborg University

Compiling Protocol Narrations into Applied Pi Processes

Sam Sepstrup Olesen

June 8, 2015

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Software Engineering
Selma Lagerløfs Vej 300
Telephone +45 9940 9940
+45 9940 9798
http://www.cs.aau.dk

Title:
Compiling Protocol Narrations into Applied Pi Processes

Project period:
10th semester
Spring, 2015

Project group:
des102f15

Author:
Sam Sepstrup Olesen

Supervisor:
Hans Hüttel

Finished: June 8, 2015

Number of pages: 41

Abstract:

The focus of this report is to ease the process of designing and analysing cryptographic protocols.
Based on the work of Briais and Nestmann, we extend their syntax to include an equational theory and define term derivation for arbitrary equational theories, such that we can produce a projection from protocol narrations to processes defined in the applied pi calculus. For reasoning about the complexity of our projection, we prove that the term derivation problem is NP-complete.
We have implemented this projection in our compiler `narcapi`, which is written in OCaml.

Sam Sepstrup Olesen

Preface

This report was written as a master project by Sam Sepstrup Olesen in the spring of 2015, while studying Software at Aalborg University.

The report documents the compilation from protocol narrations to processes in the applied pi calculus. The compilation is implemented in OCaml as the compiler `narcapi`, which is available at `https://github.com/samolesen/narcapi`.

Although, these subjects will be presented in the report to some degree, the reader is expected to be somewhat familiar with process calculi, as well as basic cryptography.

I would like to thank my supervisor, Hans Hüttel, for the constructive feedback and engaging discussions throughout my specialisation.

Contents

Chapter 1

Introduction

When designing or presenting cryptographic protocols, it is common to use informal protocol descriptions, such as *protocol narrations*, to describe the intended execution of a protocol run as a sequence or directed graph of message exchanges between the associated principals[BN07]. An example of a protocol narration can be seen in Table 1.1. Encryption of a message M with a key N denoted by $\{M\}_N$.

$$\begin{array}{l} A \rightsquigarrow S : \{B\}_{k_{AS}} \\ S \rightsquigarrow A : \{k_B\}_{k_{AS}} \\ A \rightsquigarrow B : \{m\}_{k_B} \end{array}$$

Table 1.1: Example of a protocol narration.

This protocol narration describes a simple protocol, in which a principal A can ask a key-server S for a key to communicate with a principal B. However, a few implicit assumptions about ownership of keys are made in the protocol narration. It is first assumed that A and S share a key k_{AS}. It is also assumed that B knows the key k_B that is sent to A by the key-server S. Other important aspects are the actions the principals are supposed to perform on the messages they receive, e.g. decryption and consistency checks are implicit. We obviously expect the principals to attempt decryption of the messages they receive. We also expect that the key-server S only sends the key k_B, if that is the key requested by A.

These implicit concepts of protocol narrations are formalised by Briais and Nestmann in [BN07]. Their paper also defines a translation of protocol narrations to processes in the spi calculus, which have been implemented by Briais in the `spyer` compiler[Bri08]. We believe the automatic generation of explicit protocol descriptions, from the intuitive notation of protocol narrations, eases the work of designing and analysing cryptographic protocols. However, the spi calculus contains a fixed set of cryptographic primitives that the users of the Briais compiler are limited to use. Protocols that depend on cryptographic primitives not included in the spi calculus of Briais, e.g. homomorphic encryption, are therefore not supported by the compiler.

My motivation for working on this project is to ease the process of designing and analysing cryptographic protocols. Last semester, I designed and analysed protocols for election monitoring[Ole15]. Protocol narrations were used for concise descriptions the protocols. Because the dependence on a secret sharing scheme, it was not possible to use `spyer` for automatic translation of the protocol narrations. The protocols were instead manually translated to explicit applied pi processes. I found the manual modelling of the protocols to be a rather cumbersome and error-prone process, that could be interesting to automatise.

The applied pi calculus[AF01] is an extension to the spi calculus, that allows for specification of arbitrary cryptographic primitives through the definition of an equational theory. Symmetric encryption and decryption, of a message x with a key y, can for example be written as the equation $\mathsf{dec}(\mathsf{enc}(x, y), y) = x$.

> Can the translation described in [BN07] be generalised to a translation from protocol narrations to processes in the applied pi calculus?

In addition to the translation from protocol narrations to processes in the spi calculus[BN07], existing work related to explication of protocols include the projection from global session types to local session types[HYC08] and code generation based on protocol narrations[Mod14].

1.1 Outline of the Report

We present the background theory for our work in chapter 2. We examine how we can utilise the background theory in chapter 3, where we also determine the computational complexity of a translation. In chapter 4, we define the compilation to applied pi processes. The compilation is implemented in our compiler `narcapi`. The primary derivations between the formal compilation and the implementation are described in chapter 5. Finally, chapter 6 will present the conclusion of the project.

Chapter 2

Preliminaries

In this chapter, we explore the existing work of protocol narration compilation and process calculi, which we use as a foundation for our work.

2.1 Protocol Narrations

Several implicit concepts allow protocol narrations to be concise. Implicit decomposition of messages by principals allow for implicit projection of elements in tuples, as well as implicit decryption of encrypted messages, on condition that the principal knows the corresponding decryption key. Also, protocol narrations do not explicitly specify which checks a principal should perform when a message has been received. An example is shown in Table 2.1, which is a protocol narration of a simple protocol that allows a principal A to verify whether a principal B knows a shared key k_{AB}.

$$\begin{array}{l} A \rightsquigarrow B : \{n\}_{k_{AB}} \\ B \rightsquigarrow A : \{n+1\}_{k_{AB}} \end{array}$$

Table 2.1: Example of a protocol narration.

In the protocol narration, principal A sends a nonce $n \in \mathbb{N}$, which is encrypted with a shared key k_{AB}, to principal B. By decrypting the nonce n and encrypting the value $n + 1$ with a shared key k_{AB}, B can prove to A that it knows the shared key k_{AB}. The principal A is then implicitly expected to perform a consistency check, whereby A verifies that the message it receives is the encryption of a value x, where $x - 1$ equals the nonce n that was originally sent by A. However, the protocol narration does not describe who knows what, i.e. which information is initially available to the individual principals and whether some of the information is initially available to a potential attacker. It is also important to specify whether the nonce is freshly generated for the protocol run, since it is typically assumed that an attacker is able to replay any sent message, even from previous protocol runs.

These implicit concepts can be formalized by an extended syntax and corresponding semantics for protocol narrations proposed by Briais and Nestmann[BN07]. The implicit decomposition and consistency checks are handled by their semantics, whereas the knowledge information is made explicit through a *declaration header* in their syntax, which is shown in Table 2.2.

$$
\begin{array}{rll}
M, N ::= & a \mid A \mid \{M\}_N \mid (M.N) \mid \mathsf{pub}(M) \mid \mathsf{priv}(M) \mid \mathsf{H}(M) & \text{(messages)} \\
T ::= & A \rightsquigarrow B : M & \text{(exchanges)} \\
L ::= & \epsilon \mid T; L & \text{(narrations)} \\
D ::= & A \textbf{ knows } M \mid A \textbf{ generates } n \mid \textbf{private } k & \text{(declarations)} \\
P ::= & D; P \mid L & \text{(protocol narrations)}
\end{array}
$$

Table 2.2: Formal syntax for protocol narrations.

Messages are based on the classic spi calculus[AG97] constructs. A *message* can therefore be either a name a, a principal A, an encrypted message $\{M\}_N$, a pair $(M.N)$, a public key $\mathsf{pub}(M)$, a private key $\mathsf{priv}(M)$, or a hashed message $\mathsf{H}(M)$, which are all typically available in spi calculus. The *declaration header* consists of three different constructs: The declaration A **knows** M denotes that A initially knows M. The generation of a fresh name n by A for each protocol run is defined by A **generates** n. Names that are not generated for each protocol run will be initially available to an attacker, unless they are declared to be private through the declaration **private** k. Both declarations and exchanges are separated by semicolons.

The previous example in Table 2.1 is described in Table 2.3 using the formal syntax.

$$
\begin{array}{l}
\textbf{private } k_{AB}; \\
A \textbf{ knows } k_{AB}; \\
B \textbf{ knows } k_{AB}; \\
A \textbf{ generates } n; \\
A \rightsquigarrow B : n; \\
B \rightsquigarrow A : \{n\}_{k_{AB}}; \epsilon
\end{array}
$$

Table 2.3: Example of a formal protocol narration.

To make the declaration header less verbose, Briais and Nestmann propose the two macros defined in Table 2.4.

$A_1, \cdots, A_n$ **know** M	expands into	A_1 **knows** $M; \cdots; A_n$ **knows** M
$A_1, \cdots, A_n$ **share** k	expands into	**private** $k; A_1$ **knows** $k; \cdots; A_n$ **knows** k

Table 2.4: Declaration macros.

Based on the work in [BN07], a compiler named `spyer` has been implemented by Briais, which translates formal protocol narrations into corresponding spi calculus processes[Bri08]. The syntax accepted by `spyer` contains some further changes to the syntax in [BN07], such as support for tuples, the use of line breaks as separators, and the omission of the trailing ϵ.

$$\begin{array}{l} A, B \ \textbf{know} \ (A.B) \\ A, B \ \textbf{share} \ k_{AB} \\ A \ \textbf{generates} \ n \\ A \rightsquigarrow B : n \\ B \rightsquigarrow A : \{n\}_{k_{AB}} \end{array}$$

Table 2.5: Example of a formal protocol narration in the `spyer` syntax.

The example in Table 2.1 is described using the syntax of `spyer` in Table 2.5. Note that in Table 2.3 it is implicitly assumed that principals initially know each other, whereas `spyer` requires this information to be explicitly defined in the declaration header.

2.2 Checks-on-Reception

When translating, [BN07] and `spyer` divide an exchange $A \rightsquigarrow B : M$ into three components:

1. A constructing and sending M.
2. B receiving a message that could be M.
3. B verifying that the received message has the expected properties of M.

It is clear that the interesting aspect of translating protocol narrations is which checks a recipient of a message is expected to perform. The extent of these checks heavily depends on what knowledge the principal has previously acquired, which we call the principal's *knowledge set* K. We use the following notation for a piece of knowledge $k \in K$ throughout the report:

$$\textit{expectation} \bullet \textit{representation}$$

The *expectation* is the intended message as described in the protocol narration. The *representation* defines how the principal is supposed to express the *expectation*, i.e. when receiving a message M, the principal would store the message in an unused variable a, resulting in the addition of $M \bullet a$ to the principal's knowledge set. You can therefore say that the *representation* is supposed to evaluate to the *expectation*. We denote a test for equality of the evaluation of two representations a and b as $[a = b]$. We omit the definition of evaluation as it is straightforward, however, note that a representation will evaluate to $\perp$ when an unsuitable operation is attempted, e.g. decryption $\mathsf{D}_b(a)$ with a wrong key b, decryption $\mathsf{D}_b(a)$ when a is not an encrypted value, or projection $\pi_n(a)$ of the nth element when a is not a pair. If either of the representations in an equality test evaluate to $\perp$, the test will return `false`.

Given this notion of knowledge, one can imagine scenarios where checks should be generated:

- If you know multiple representations for the same expected value, e.g. $m \bullet a$ and $m \bullet b$, you would expect the representations to evaluate to the same value, i.e. $[a = b]$.
- When receiving a pair $(m.n) \bullet a$, you would expect to be able to derive $m \bullet \pi_1(a)$ and $n \bullet \pi_2(a)$, such that $[a = (\pi_1(a).\pi_2(a))]$.
- When coming into possession of a value $m \bullet a$ and its hashed value $\mathsf{H}(m) \bullet b$, you would expect that $[\mathsf{H}(a) = b]$.
- When coming into possession of an encrypted message $\{m\}_k \bullet a$ and its corresponding key $k \bullet b$, you would expect to be able to derive $m \bullet \mathsf{D}_b\,(a)$, where $[a = \{\mathsf{D}_b\,(a)\}_b]$.

By applying a sequence of actions for every message exchange, Briais and Nestmann define the generation of *checks-on-reception* as follows:

1. When a principal receives a message expected to be m, it is added to the principal's knowledge set K, producing $K' = K \cup \{\, m \bullet a \,\}$ where the representation $a \notin \mathsf{vars}(K)$. For the knowledge set K', the set of knowledge that can be derived is described through analysis $\mathcal{A}(K')$, which is defined in Table 2.7. The associated synthesis $\mathcal{S}(K)$ is defined in Table 2.8 and is the set of knowledge that can be constructed using K. The analysis and synthesis are the key components of the checks-on-reception method.
2. The checks to be generated are denoted by $\Phi(\mathcal{A}(K'))$ and are defined using the consistency formula in Table 2.6, which generates checks for every expectation in $\mathcal{A}(K')$ that can be synthesised using $\mathcal{A}(K')$. $\mathsf{inv}(E, F)$ is used to test whether a public key and a private key are a part of the same key pair. Note that for every expectation E the check $[E = E]$ will be generated. Briais and Nestmann refer to these as *well-formed tests*, which will fail iff E evaluates to $\perp$. The well-formed tests verify whether decryption and projection succeed. However, it should be noted that these well-formed tests require that the cryptographic primitives used in real world scenarios provide data integrity, i.e. produce errors for wrong input.

$$\begin{aligned}\Phi(K) \overset{\text{def}}{=} \;& \bigwedge\nolimits_{M \bullet E \in K \;\wedge\; M \bullet F \in \mathcal{S}(K)} [E = F] \\ & \wedge \bigwedge\nolimits_{M \bullet E \in K \;\wedge\; M^{-1} \bullet F \in \mathcal{S}(K)} \mathsf{inv}(E, F)\end{aligned}$$

Table 2.6: Consistency formula.

3. Finally, Briais and Nestmann propose several measures to minimise the principals' knowledge sets and avoid repeating checks. While making the resulting spi calculus processes simpler, these measures are not essential for the correctness of the translation.

Analysis $\mathcal{A}(K)$, given a knowledge set K, is the smallest set of $\bigcup_{n\in\mathbb{N}}\mathcal{A}_n(K)$ satisfying the following rules:

$$\textsc{ana-ini}\,\frac{M\bullet E\in K}{M\bullet E\in\mathcal{A}_0(K)}$$

$$\textsc{ana-fst}\,\frac{(M.N)\bullet E\in\mathcal{A}_n(K)}{M\bullet\pi_1(E)\in\mathcal{A}_{n+1}(K)}\qquad\textsc{ana-snd}\,\frac{(M.N)\bullet E\in\mathcal{A}_n(K)}{N\bullet\pi_2(E)\in\mathcal{A}_{n+1}(K)}$$

$$\textsc{ana-dec}\,\frac{\{M\}_N\bullet E\in\mathcal{A}_n(K)\quad N^{-1}\bullet F\in\mathcal{S}(\mathcal{A}_n(K))}{M\bullet\mathsf{D}_F\,(E)\in\mathcal{A}_{n+1}(K)}$$

$$\textsc{ana-dec-rec}\,\frac{\{M\}_N\bullet E\in\mathcal{A}_n(K)\quad N^{-1}\bullet F\notin\mathcal{S}(\mathcal{A}_n(K))}{\{M\}_N\bullet E\in\mathcal{A}_{n+1}(K)}$$

$$\textsc{ana-nam-rec}\,\frac{M\bullet E\in\mathcal{A}_n(K)\quad M\in\mathbf{N}\cup\mathbf{A}}{M\bullet E\in\mathcal{A}_{n+1}(K)}$$

$$\textsc{ana-pub}\,\frac{\mathsf{pub}(M)\bullet E\in\mathcal{A}_n(K)}{\mathsf{pub}(M)\bullet E\in\mathcal{A}_{n+1}(K)}\qquad\textsc{ana-priv}\,\frac{\mathsf{priv}(M)\bullet E\in\mathcal{A}_n(K)}{\mathsf{priv}(M)\bullet E\in\mathcal{A}_{n+1}(K)}$$

$$\textsc{ana-hash}\,\frac{\mathsf{H}(M)\bullet E\in\mathcal{A}_n(K)}{\mathsf{H}(M)\bullet E\in\mathcal{A}_{n+1}(K)}$$

Table 2.7: Analysis.

Synthesis $\mathcal{S}(K)$, given a knowledge set K, is defined as the smallest set satisfying the following rules:

$$\textsc{syn-pair}\,\frac{M\bullet E\in\mathcal{S}(K)\quad N\bullet F\in\mathcal{S}(K)}{(M.N)\bullet(E.F)\in\mathcal{S}(K)}$$

$$\textsc{syn-enc}\,\frac{M\bullet E\in\mathcal{S}(K)\quad N\bullet F\in\mathcal{S}(K)}{\{M\}_N\bullet\{E\}_F\in\mathcal{S}(K)}\qquad\textsc{syn-hash}\,\frac{M\bullet E\in\mathcal{S}(K)}{\mathsf{H}(M)\bullet\mathsf{H}(M)\in\mathcal{S}(K)}$$

$$\textsc{syn-priv}\,\frac{M\bullet E\in\mathcal{S}(K)}{\mathsf{priv}(M)\bullet\mathsf{priv}(M)\in\mathcal{S}(K)}\qquad\textsc{syn-pub}\,\frac{M\bullet E\in\mathcal{S}(K)}{\mathsf{pub}(M)\bullet\mathsf{pub}(M)\in\mathcal{S}(K)}$$

Table 2.8: Synthesis.

2.3 Applied Pi Calculus

The applied pi calculus[AF01], by Abadi and Fournet, provides a formal syntax and semantics for describing and examining cryptographic protocols. It is a generalisation of the spi calculus[AG97] and extends the π-calculus with function symbols and term equivalences, that allow for specification of arbitrary cryptographic primitives.

The syntax of the applied pi calculus is defined in Table 2.9.

$M, N ::=$		Terms
	a	Name
	$\mid x$	Variable
	$\mid f(M_1, \cdots, M_n)$	Function application
$P, Q ::=$		Processes
	$\mathbf{0}$	Null process
	$\mid P \mid Q$	Parallel composition
	$\mid\, !P$	Replication
	$\mid \nu a.P$	Name restriction
	$\mid$ if $M = N$ then P else Q	Conditional
	$\mid u(x).P$	Message input
	$\mid \overline{u}\langle M\rangle.P$	Message output
$A, B, C ::=$		Extended processes
	P	Plain process
	$\mid A \mid B$	Parallel composition
	$\mid \nu u.P$	Restriction
	$\mid \{^M/_x\}$	Active substitution

Table 2.9: The syntax of the applied pi calculus.

The extension to π-calculus is realised by *terms*, which consist of *function applications*, *names*, and *variables* representing terms. Each term M must comply with a *signature* Σ, which we denote by writing $\Sigma \vdash M$, where for each contained function application $f(N_1, \cdots, N_n)$ in M, we have that the function symbol f of arity n is in Σ. We let both u and v range over names and variables. The set of *processes* is almost identical to those of the π-calculus, but note that channels can be both names and variables. Further, equality of terms is defined through an equational theory Θ, which is a set of pairs (M, N) forming the corresponding *term equivalences* $M = N$. We have that $\forall(M, N) \in \Theta : \Sigma \vdash M \wedge \Sigma \vdash N$. The set of *extended processes* consists of the *plain process*, *parallel composition*, *restriction*, and *active substitution* $\{^M/_x\}$, which denotes the replacement of the *variable* x with the *term* M.

The equational theory Θ describes the behaviour of the functions in Σ, and allows for modelling of cryptographic primitives. As an example, non-deterministic symmetric encryption and decryption can be modelled as follows, for a message x, key y, and nonce z:

$$\mathsf{dec}(\mathsf{enc}(x, y, z)\,, y) = x$$

Any extended process can be mapped to a frame $\phi(A)$, which consists only of restrictions and active substitutions, by replacing every plain process with the null process $\mathsf{0}$. A frame $\phi(A)$ represents the static information that is exposed to a *context* of the extended process A.

A *context* is an extended process with a hole $[\cdot]$. An *evaluation context* is a context, whose hole is not under a replication, a conditional, a message output, or a message input, i.e the context $\overline{c}\langle x\rangle.P \mid c(x).[\cdot]$ is a context, but not an evaluation context as the hole is under a message input. When A in place of the hole in a context is closed, we say that the context closes A.

If X is a syntactic entity in the applied calculus, e.g. a process, we denote its set of free names $\mathrm{fn}(X)$, bound names $\mathrm{bn}(X)$, free variables $\mathrm{fv}(X)$, and bound variables $\mathrm{bv}(X)$.

Structural congruence $\equiv$ is defined as the least equivalence satisfying the axioms in Table 2.10, and is closed by application of any evaluation context.

Internal message transfers and conditional branching is represented by reduction, whose semantics is shown in Table 2.11. Reduction is closed by structural congruence and application of any evaluation context.

The labelled semantics extends the rules of structural congruence and reduction, with the rules shown in Table 2.12. The transitions are of the form: $A \xrightarrow{\alpha} A'$, i.e. an extended process A becomes the extended process A', after performing the action α. An action α can either be an input action $a(M)$, a bound output action $\nu u.\overline{a}\langle u\rangle$, or a free output action $\overline{a}\langle u\rangle$. Note that $A\{^{M}/_{x}\}$ denotes the substitution of x with M in the process A.

α-EQUIV	$A \equiv B$	if A and B are α-equivalent.
PAR-**0**	$A \equiv A \mid \mathbf{0}$	
PAR-A	$A \mid (B \mid C) \equiv (A \mid B) \mid C$	
PAR-C	$A \mid B \equiv B \mid A$	
REPL	$!P \equiv P \mid !P$	
NEW-**0**	$\nu n.\mathbf{0} \equiv \mathbf{0}$	
NEW-C	$\nu u.\nu v.A \equiv \nu v.\nu u.A$	
PAR-A	$A \mid \nu u.B \equiv \nu u.(A \mid B)$	when $u \notin \mathrm{fv}(A) \cup \mathrm{fn}(A)$
ALIAS	$\nu x.\{^{M}/_{x}\} \equiv \mathbf{0}$	
SUBST	$\{^{M}/_{x}\} \mid A \equiv \{^{M}/_{x}\} \mid A\{^{M}/_{x}\}$	
REWRITE	$\{^{M}/_{x}\} \equiv \{^{N}/_{x}\}$	when $\Sigma \vdash M = N$

Table 2.10: Axioms of structural congruence.

COMM $\overline{a}\langle x\rangle.P \mid a(x).Q \rightarrow P \mid Q$ THEN if $M = M$ then P else $Q \rightarrow P$

$$\text{ELSE}\frac{\mathrm{fv}(M) = \mathrm{fv}(N) = \emptyset \qquad (M, N) \notin \Theta}{\text{if } M = N \text{ then } P \text{ else } Q \rightarrow Q}$$

Table 2.11: Internal reduction.

$$\textsc{Input}\quad a(x).P \xrightarrow{a(M)} P\{^M/_x\} \qquad \textsc{Output}\quad \overline{a}\langle u\rangle.P \xrightarrow{\overline{a}\langle u\rangle} P$$

$$\textsc{Open}\frac{A \xrightarrow{\overline{a}\langle u\rangle} A' \qquad u \neq a}{\nu u.A \xrightarrow{\nu u.\overline{a}\langle u\rangle} A'} \qquad \textsc{Scope}\frac{A \xrightarrow{\alpha} A' \qquad u \text{ does not occur in } \alpha}{\nu u.A \xrightarrow{\alpha} \nu u.A'}$$

$$\textsc{Par}\frac{A \xrightarrow{\alpha} A' \qquad \mathrm{bv}(\alpha) \cap \mathrm{fv}(B) = \mathrm{bn}(\alpha) \cap \mathrm{fn}(B) = \emptyset}{A|B \xrightarrow{\alpha} A'|B}$$

$$\textsc{Struct}\frac{A \equiv B \qquad B \xrightarrow{\alpha} B' \qquad B' \equiv A'}{A \xrightarrow{\alpha} A'}$$

Table 2.12: The labelled semantics of the applied pi calculus.

Chapter 3

Analysis

Since we compile to applied pi processes, an equational theory must be supplied with the protocol narrations, as it is required by the applied pi calculus. In this chapter, we examine the obstacles of adding such equational theory to protocol narrations.

3.1 Compilation

It is clear that the main result in [BN07] is Briais and Nestmann's method for generation of checks-on-reception. We can utilise this method to define a compilation of protocol narrations with an attached equational theory, but because the analysis and synthesis in section 2.2 is defined for a fixed equational theory, we have to create new definitions of analysis and synthesis that allow for derivation of terms using arbitrary equational theories.

3.2 Complexity

While the addition of an equational theory for deriving terms is useful for modelling arbitrary cryptographic primitives, they can be the source of high computational complexity. We now show that the problem of deriving terms using an equational theory and a set of terms is NP-complete.

Definition 1 (Term Derivation Problem)**.** Given a signature Σ, an equational theory Θ, a set of terms T, and a term t, can a term be synthesised from Σ and T that equals t in Θ?

Theorem 1. *The term derivation problem is NP-Complete.*

Proof. We argue that the term derivation problem is in NP, since given a certificate C, which represents a sequence of terms, such that $C = M \to \cdots \to N$, it can be verified in polynomial time that $N = t$ in Θ and that every step in the sequence is constructed of function applications in Σ and terms in T.

We now show that the NP-Complete problem, SAT, is polynomial-time reducible to the term derivation problem. Given a boolean formula X, we define the input for term derivation as the signature Σ_{SAT}, the equational theory Θ_{SAT}, the set of terms $\{\, \mathsf{a}(m) \,\}$, and the term m.

$$\Sigma_{SAT} \overset{\text{def}}{=} \left\{ \ \mathsf{tt} \mapsto 0, \mathsf{ff} \mapsto 0, \mathsf{and} \mapsto 2, \mathsf{or} \mapsto 2, \mathsf{neg} \mapsto 1, \mathsf{sat} \mapsto 2, \mathsf{a} \mapsto 1, \mathsf{b} \mapsto 1 \right\}$$

$$\Theta_{SAT} \overset{\text{def}}{=} \left\{ \begin{array}{ccc} \mathsf{and}(\mathsf{tt},\mathsf{tt}) = \mathsf{tt} & \mathsf{or}(\mathsf{tt},\mathsf{tt}) = \mathsf{tt} & \\ \mathsf{and}(\mathsf{tt},\mathsf{ff}) = \mathsf{ff} & \mathsf{or}(\mathsf{tt},\mathsf{ff}) = \mathsf{tt} & \mathsf{neg}(\mathsf{tt}) = \mathsf{ff} \\ \mathsf{and}(\mathsf{ff},\mathsf{tt}) = \mathsf{ff} & \mathsf{or}(\mathsf{ff},\mathsf{tt}) = \mathsf{tt} & \mathsf{neg}(\mathsf{ff}) = \mathsf{tt} \\ \mathsf{and}(\mathsf{ff},\mathsf{ff}) = \mathsf{ff} & \mathsf{or}(\mathsf{ff},\mathsf{ff}) = \mathsf{ff} & \\ \multicolumn{3}{c}{\mathsf{sat}(X\!\upharpoonright, \mathsf{a}(x)) = \mathsf{sat}(X\!\upharpoonright, \mathsf{b}(x))} \\ \multicolumn{3}{c}{\mathsf{sat}(\mathsf{tt}, \mathsf{b}(x)) = x} \end{array} \right\}$$

$X\!\upharpoonright$ denotes the trivial projection of X such that the boolean logic is substituted with the equivalent and, or, and neg functions in Σ_{SAT}, and we have that $x \notin \mathsf{vars}(X\!\upharpoonright)$. Is is clear from the equations for sat that a term equal to m can be synthesised, iff X is satisfiable. Since $X\!\upharpoonright$ can be trivially constructed in polynomial-time, we have that SAT is polynomial-time reducible to the term derivation problem.

We have shown that the term derivation problem is in NP and that SAT is polynomial-time reducible to it. We therefore conclude that the term derivation problem is NP-complete. □

Even though the term derivation problem is shown to be NP-complete, we argue that the practical use of a compiler is not affected significantly, as the complexity of the equational theories will often be low enough to allow for rapid computation.

Chapter 4

Design

In this chapter, we first define a syntax for equational protocol narrations. We then define analysis, synthesis, and generation of checks. Using these definitions, a compilation from equational protocol narrations to applied pi processes is presented.

4.1 Syntax of Equational Protocol Narrations

The syntax for equational protocol narration is shown in Table 4.1. We simply add equational theory to Briais and Nestmann's syntax in Table 2.2.

$$\begin{array}{rlr}
M, N & ::= a \mid A \mid f(\,\tilde{M}\,) & \text{(terms)} \\
E & ::= A \rightsquigarrow B : M \mid E; E & \text{(exchanges)} \\
D & ::= A \ \textbf{knows}\ M \mid A \ \textbf{generates}\ n \mid \textbf{private}\ k \mid D; D & \text{(declarations)} \\
T & ::= M = N \mid f/i \mid T; T & \text{(equational theory)} \\
P & ::= T; D; E; \epsilon & \text{(equational protocol narrations)}
\end{array}$$

Table 4.1: Generalised syntax for equational protocol narrations.

We use the notation $f(\tilde{M})$ to denote a function application of the function f with a sequence of arguments $\tilde{M}$, such that:

$$f(M_1, M_2, \cdots, M_n) = f(\tilde{M})$$

For an equational protocol narration P, the signature Σ of P is the smallest set of functions where $f \mapsto i \in \Sigma$ for any function declaration f/i in P. Likewise, the equational theory Θ of P is the smallest set of term equivalence pairs, where $(M, N) \in \Theta$ for any term equivalence $M = N$ in P.

4.2 Knowledge Deduction

In this section, we define analysis and synthesis for an equational theory Θ and a knowledge set K. We first introduce the formulae for synthesis and analysis, followed by their dependent formulae in order of appearance.

We consider the equational theory Θ to be a rewrite system, containing left-to-right oriented rewrite rules. Any set of equations can be converted to an equivalent rewrite system[DJ90]. We use a notation similar to the conventional set-builder notation for applying a list of parameters $\tilde{x}$ to a function f, such that:

$$f(x_1, x_2, \cdots, x_n) = f(\tilde{x}) = f(x \mid x \in \tilde{x})$$

Note that we let the membership operator $\in$ preserve each element's position in the sequence. Somewhat unconventionally we let a substitution be a *partial* mapping $\sigma : \mathbf{V} \to \mathbf{M}$ for variables $\mathbf{V}$ to terms $\mathbf{M}$. For a substitution $\sigma = \{ {}^{t_1}/_{x_1}, {}^{t_2}/_{x_2}, \cdots, {}^{t_n}/_{x_n} \}$, we thus only include the explicitly defined mappings ${}^{t_i}/_{x_i}$ where $1 \leq i \leq n$. The set of variables, with a defined mapping in σ, is called the domain $\mathsf{dom}(\sigma)$, such that $\mathsf{dom}(\sigma) = \{ x \mid {}^{t}/_{x} \in \sigma \}$. Note that the application $t\sigma$ of a substitution σ to a term t results in $\perp$, when no mapping exists in σ for some variable in t, i.e. $\mathsf{vars}(t) \not\subseteq \mathsf{dom}(\sigma)$. As an example of a failing substitution application we have that $\mathsf{f}(a,b)\{ {}^{\mathsf{h}(k)}/_{a} \} = \perp$. A substitution with containing mappings becomes $\perp$, e.g. $\{ {}^{b}/_{a}, {}^{c}/_{a} \} = \perp$. A set containing $\perp$ becomes $\perp$.

Definition 2 (Synthesis)**.** The *synthesis* formula $\mathcal{S}(t, K)$ returns the smallest set satisfying the definition in Table 4.2. The result of the formula is the set of a principal's representations for a term t given the principal's knowledge K.

$$\mathcal{S}(t, K) \ni \begin{cases} r & \text{if } e \bullet r \in K : e = t \\ f(s \mid n \in \tilde{n} \land s \in \mathcal{S}(n, K)) & \text{if } t = f(\tilde{n}) \text{ and } \forall n \in \tilde{n} : \mathcal{S}(n, K) \neq \emptyset \\ \mathcal{S}(e_r\sigma, K) & \text{if } (e_l, e_r) \in \Theta \land \sigma = \mathcal{U}(e_l, \emptyset, t) \end{cases}$$

Table 4.2: Synthesis.

Case 1 The trivial case, where a representation for the term t can be found in the knowledge K.

Case 2 If t is a function application $f(\tilde{n})$, the second case allows for recursive synthesis of the arguments $\tilde{n}$, where the function f is applied on the resulting representations, e.g. given the knowledge set $\{ m \bullet a \}$, it is clear that the hashed term $\mathsf{hash}(m)$ can be represented by $\mathsf{hash}(a)$.

Example. For the equational theory $\Theta = \{ \mathsf{f}(x) = \mathsf{g}(x) \}$, knowledge set $K = \{ \mathsf{h}(\mathsf{f}(m)) \bullet a, \mathsf{g}(m) \bullet b \}$, and term $t = \mathsf{h}(\mathsf{f}(m))$, the synthesis $\mathcal{S}(t, K)$ is the set of possible representations of t. Through the first case in the formula, it is clear that t can be represented by a, because the representation of a equals t. Using the second case, we can attempt the synthesis $\mathcal{S}(\mathsf{f}(m), K)$ of t's subterm. We can then follow the third case, which through the equation in Θ allows us to find the representation b for $\mathsf{g}(m)$ using the first case. Finally, when stepping out of the recursion, the second case requires the function h to be applied on b, resulting in the representation $\mathsf{h}(b)$ for t. We therefore have that $\mathcal{S}(t, K) = \{ a, \mathsf{h}(b) \}$.

Definition 3 (Analysis)**.** The *analysis* formula $\mathcal{A}(K)$ returns a set of knowledge $K \cup K'$, where K' is the set of knowledge that can be derived from K. $\mathcal{A}_0(K)$ derives the variants of all terms. An iteration of $\mathcal{A}_{n+1}(K)$ derives knowledge $e \bullet r$, where the expectation e is the application of a substitution of an equation's left-hand side to its corresponding right-hand side, for each possible representation r. The analysis is defined recursively, for a finite n such that $\mathcal{A}_n(K) = \mathcal{A}_{n+1}(K)$.

$$\mathcal{A}_0(K) \overset{\text{def}}{=} \left\{ \mathcal{V}(e) \bullet r \;\middle|\; e \bullet r \in K \right\}$$

$$\mathcal{A}_{n+1}(K) \overset{\text{def}}{=} \mathcal{A}_n(K) \cup \left\{ e_r\sigma \bullet r \;\middle|\; \bigwedge \begin{array}{c} (e_l, e_r) \in \Theta \\ \sigma \in \mathcal{F}(e_l, e_r, \mathcal{A}_n(K)) \\ r \in \mathcal{S}(e_l\sigma, \mathcal{A}_n(K)) \end{array} \right\}$$

$$\mathcal{A}(K) \overset{\text{def}}{=} \bigcup_{n\in\mathbb{N}} \mathcal{A}_n(K)$$

Table 4.3: Analysis.

Example. For the equational theory representing symmetric encryption $\Theta = \{\, \mathsf{f}(\mathsf{g}(x,y)\,,y) = x \,\}$ and knowledge set $K = \{\, \mathsf{g}(m,k) \bullet a, k \bullet b \,\}$, the analysis $\mathcal{A}(K)$ is the set of knowledge that can be derived from K using Θ. Because the terms in K have no variants, we have that $\mathcal{A}_0(K) = K$. In the second iteration, the set *find substitutions* $\mathcal{F}(\mathsf{f}(\mathsf{g}(x,y)\,,y)\,,x,\mathcal{A}_0(K))$ includes the substitution $\sigma = \{\, {}^{m}/_{x}, {}^{k}/_{y} \,\}$. We have therefore deduced the term $x\sigma = m$. A representation of the term is then found using synthesis of $\mathsf{f}(\mathsf{g}(x,y)\,,y)\,\sigma = \mathsf{f}(\mathsf{g}(m,k)\,,k)$, which yields the representation $\mathsf{f}(a,b)$. We therefore have that $\mathcal{A}_1(K) = K \cup \{\, m \bullet \mathsf{f}(a,b) \,\}$, which is also the result of $\mathcal{A}(K)$, because no new knowledge can be derived in further iterations.

Definition 4 (Term unification)**.** *Term unification* $\mathcal{U}(t_1, \sigma, t_2)$ is defined by the formula in Table 4.4, which unifies a term t_1 with t_2, while satisfying an existing substitution σ by attempting to extend σ with a minimal substitution σ', such that $t_1(\sigma \cup \sigma') = t_2$. When the terms fail to unify, term unification results in $\perp$.

$$\mathcal{U}(t_1, \sigma, t_2) \overset{\text{def}}{=} \begin{cases} \sigma \cup \{\, {}^{t_2}/_{t_1} \,\} & \text{if } t_1 \in \mathbf{V} \wedge (t_1 \notin \mathsf{dom}(\sigma) \vee t_1\sigma = t_2) \\ \bigcup_{1 \,\le\, i \,\le\, |\tilde{m}|} \mathcal{U}(m_i, \sigma, n_i) & \begin{array}{l} \text{if } t_1 = f(\tilde{m}) \wedge t_2 = f(\tilde{n}) \\ \text{where } m_i \in \tilde{m} \text{ and } n_i \in \tilde{n} \end{array} \\ \perp & \text{otherwise} \end{cases}$$

Table 4.4: Term unification.

Case 1 The first case covers the condition, where t_1 is a variable and σ does not contain a substitution conflicting with ${}^{t_2}/_{t_1}$.

Case 2 When both t_1 and t_2 are instances of the same function application, the second case is applied where the term unification function is called pairwise on t_1 and t_2's arguments. Recall that a substitution with conflicting mappings for the same variable becomes invalid, i.e. $\perp$.

Case 3 When the first and second case are inapplicable, the result is $\perp$.

Definition 5 (Term variants)**.** The formula *term variants* $\mathcal{V}(e)$ returns the smallest set satisfying the definition in Table 4.5. Given a term e, a set of variants of the term is returned. The variants are based on the equational theory. The formula ensures that variants of a term's subterms are derived.

$$\mathcal{V}(e) \ni \begin{cases} e & \\ f(\mathcal{V}(n) \mid n \in \tilde{n}) & \text{if } e = f(\tilde{n}) \\ f(\mathcal{V}(e_r\sigma) \mid n \in \tilde{n} \wedge (e_l, e_r) \in \Theta \wedge \sigma = \mathcal{U}(e_l, \emptyset, n)) & \text{if } e = f(\tilde{n}) \end{cases}$$

Table 4.5: Term variants.

Case 1 The trivial case simply includes e in the set.

Case 2 The formula is called recursively with the arguments of e, thereby allowing for variants of subterms.

Case 3 The third case behaves like the second case, but the arguments of e are exchanged with the right-hand side of an equation.

Example. For the equation theory $\Theta = \{\, \mathsf{f}(x, y) = \mathsf{f}(y, x) \,\}$, and the piece of knowledge $e = \mathsf{h}(\mathsf{f}(m, n)) \bullet a$, the function adds the variant $\mathsf{h}(\mathsf{f}(n, m)) \bullet a$ to the knowledge set by returning $\mathcal{V}(e) = \{\, \mathsf{h}(\mathsf{f}(m, n)) \bullet a, \mathsf{h}(\mathsf{f}(n, m)) \bullet a \,\}$.

Definition 6 (Find substitutions)**.** The formula *find substitutions* $\mathcal{F}(e_l, e_r, K)$ returns a set of substitutions, where knowledge in K can derive a substitution for the right-hand side of an equation e_r by fulfilling the left-hand side e_l.

$$\mathcal{F}(e_l, e_r, K) \overset{\text{def}}{=} \bigcup \mathcal{M}(\{\, e_l \,\}, \sigma, K) \quad \text{where } \sigma \in \{\, \mathcal{T}(k, e_l, e_r) \mid k \bullet \_ \in K \,\}$$

Table 4.6: Find substitutions.

Definition 7 (Term fit)**.** The formula *term fit* $\mathcal{T}(t, e_l, e_r)$, shown in Table 4.7, returns a set of substitutions, where a substitution is the result of the unification of the left-hand side of an equation e_l to t or the unification of e_l's subterms to t. When e_l unifies with t, the unification from e_l's subterms is not included in the resulting set. Informally the formula returns the outermost possible unification(s) from e_l in t. Furthermore, for each substitution σ in the result, it should be the case that $\mathsf{vars}(e_r) \subseteq \mathsf{dom}(\sigma)$.

$$\mathcal{T}(t, e_l, e_r) \overset{\text{def}}{=} \begin{cases} \emptyset & \text{if } e_l \in \mathbf{V} \\ \{\, \mathcal{U}(e_l, \emptyset, t) \,\} & \text{if } e_l \notin \mathbf{V} \wedge \mathsf{vars}(\mathcal{U}(e_l, \emptyset, t)) \supseteq \mathsf{vars}(e_r) \\ \{\, \mathcal{T}(t, x, e_r) \mid x \in \tilde{n} \,\} & \text{otherwise, where } e_l = f(\tilde{n}) \end{cases}$$

Table 4.7: Term fit.

Case 1 The empty set $\emptyset$ is returned, when e_l is a variable, since no new knowledge can be derived from a variable.

Case 2 The second case returns a singleton of the unification of e_l to a function application t, if the substitution replaces all variables in e_r.

Case 3 When those cases are inapplicable, the formula is called recursively for each argument of the function application t.

Example. Given $t = \mathsf{g}(a, b)$ and the equation $\mathsf{f}(\mathsf{g}(x, y), \mathsf{g}(x, y)) = x$, it is clear that the term t "fits" in the two subterms of the equations left-hand side, producing the set of substitutions: $\{\, \{\, {}^{a}/_{x},\ {}^{b}/_{y} \,\}, \{\, {}^{b}/_{x},\ {}^{a}/_{y} \,\} \,\}$. The reason is that the third case is satisfied, which returns $\{\, \mathcal{T}(t, \mathsf{g}(x, y), x), \mathcal{T}(t, \mathsf{g}(x, y), x) \,\}$, where the two instances of the term fit function can be unified.

Definition 8 (Fulfil substitution)**.** The *fulfil substitution* formula $\mathcal{M}(T, \sigma, K)$ in Table 4.8 will ensure that the terms in T, that have variables in the domain of σ, can be synthesised. Furthermore, the substitution will be extended if necessary.

$$\mathcal{M}(T, \sigma, K) \overset{\text{def}}{=} \begin{cases} \{\, \sigma \,\} & \text{if } \forall t \in T : t \in \mathbf{V} \wedge t \notin \mathsf{dom}(\sigma) \\ \mathcal{M}((T \cup \tilde{n}) \setminus \{\, f(\tilde{n}) \,\}, \sigma, K) & \text{if } f(\tilde{n}) \in T \text{ and the result} \neq \bot \\ \bigcup \mathcal{M}(T \setminus \{\, t \,\}, \sigma', K) & \begin{array}{l}\text{otherwise, where} \\ t \in T \wedge \sigma' \in \mathcal{K}(t, \sigma, K)\end{array} \end{cases}$$

Table 4.8: Fulfill substitution.

Case 1 The trivial case, where all terms (if any) in T are variables and not in the domain of σ. They can therefore be ignored and a singleton set of σ can be returned.

Case 2 The second case calls the formula recursively with t replaced with its arguments in T. Note that the result must not be $\bot$.

Case 3 When the first and the second case are inapplicable, the result is the knowledge substitutions for the term t with substitution σ.

Example. Given the equation $\mathsf{f}(\mathsf{g}(x, y), \mathsf{g}(y, z)) = x$, a substitution $\{\, {}^{m}/_{x},\ {}^{k}/_{y} \,\}$, and a knowledge set $\{\, \mathsf{g}(m, k) \bullet a,\ \mathsf{g}(k, n) \bullet b \,\}$, the substitution will be extended with ${}^{n}/_{z}$.

Definition 9 (Knowledge substitutions)**.** The *knowledge substitutions* formula $\mathcal{K}(t, \sigma, K)$ in Table 4.9 returns a set of substitutions, each extending σ with a substitution σ', such that $t(\sigma \cup \sigma')$ can be synthesised using the knowledge K.

$$\mathcal{K}(t, \sigma, K) \stackrel{\text{def}}{=} \begin{cases} \{\, \sigma \,\} & \text{if } \mathcal{S}(t\sigma, K) \neq \emptyset \\ \{\, \mathcal{U}(t, \sigma, k) \;\big|\; k \bullet \_ \in K \wedge \mathcal{U}(t, \sigma, k) \neq \bot \,\} & \text{otherwise} \end{cases}$$

Table 4.9: Knowledge substitutions.

Case 1 A singleton of σ is returned, if $t\sigma$ can be synthesised by the principal.

Case 2 When the condition in the first case is false, a set of unification substitutions, satisfying σ from t to the expected value of all pieces of knowledge, is returned.

4.3 Generation of Checks

After deducing terms T, consistency checks are generated for any term t where it is expected that t can be constructed using the remaining terms $T \setminus t$. To formalise this behaviour, we define the set of consistency checks.

Definition 10 (Consistency checks)**.** Given a set of knowledge K, its set of consistency checks is described by $\Phi(\mathcal{A}(K))$ using the formula in Table 4.10. For every piece of knowledge $M \bullet E \in K$ a check is generated for all possible representations that can be synthesised for M. This includes the well-formed test $[E = E]$.

$$\Phi(K) \stackrel{\text{def}}{=} \bigwedge\nolimits_{M \bullet E \in K \,\wedge\, F \in \mathcal{S}(M,K)} [E = F]$$

Table 4.10: Consistency checks.

While not necessary for the correctness of the translation, the knowledge set can be reduced with no loss of information after checks have been generated, by which repetition of some checks is avoided.

Definition 11 (Irreducibles)**.** The formula for the irreducible knowledge set $\mathcal{I}(K)$ of a knowledge set K is defined in Table 4.11. It recursively removes a piece of knowledge k that can be synthesized by the remaining knowledge set $K \setminus \{\, k \,\}$ until no such k exists.

$$\mathcal{I}(K) \stackrel{\text{def}}{=} \begin{cases} \mathcal{I}(K \setminus \{\, e \bullet r \,\}) & \text{if } e \bullet r \in K \wedge \mathcal{S}(e, K \setminus \{\, e \bullet r \,\}) \neq \emptyset \\ K & \text{otherwise} \end{cases}$$

Table 4.11: Irreducibles.

4.4 Compilation to Applied Pi

By utialising the previous definitions, we can now define the compilation of an equational protocol narration P. We first define the projection of a principal in P to its local description in the form of an applied pi process.

Definition 12 (Principal Projection)**.** Given a signature Σ and an equational theory Θ, the projection $\llbracket P \rrbracket_A^{(\upsilon,\kappa)}$ of a principal A in a equational protocol narration P into a corresponding applied pi calculus process is defined in Table 4.12, where υ is a set of used variables and κ is a set of principals' knowledge sets represented as pairs $A \times M \bullet N$. The knowledge of a principal A can be projected from κ using $\kappa(A)$, where $\kappa(A) = \left\{\, e \bullet r \;\middle|\; (A, e \bullet r) \in \kappa \,\right\}$.

$$
\begin{array}{rcl}
\llbracket \epsilon \rrbracket_A^{(\upsilon,\kappa)} & \overset{\text{def}}{=} & \mathbf{0} \\
\llbracket f/i; P \rrbracket_A^{(\upsilon,\kappa)} & \overset{\text{def}}{=} & \llbracket P \rrbracket_A^{(\upsilon,\kappa)} \quad \text{if } \nexists (f \mapsto j) \in \Sigma : i \neq j \\
\llbracket M = N; P \rrbracket_A^{(\upsilon,\kappa)} & \overset{\text{def}}{=} & \llbracket P \rrbracket_A^{(\upsilon,\kappa)} \quad \text{if } \Sigma \vdash M \wedge \Sigma \vdash N \\
\llbracket A' \textbf{ knows } M; P \rrbracket_A^{(\upsilon,\kappa)} & \overset{\text{def}}{=} & \llbracket P \rrbracket_A^{(\upsilon \cup \mathsf{vars}(M), \kappa \cup \{A', M \bullet M\})} \quad \text{if } \Sigma \vdash M \\
\llbracket A' \textbf{ generates } n; P \rrbracket_A^{(\upsilon,\kappa)} & \overset{\text{def}}{=} & \begin{cases} \nu n. \llbracket P \rrbracket_A^{(\upsilon \cup \{n\}, \kappa \cup \{A, n \bullet n\})} & \text{if } A = A' \wedge n \notin \upsilon \\ \llbracket P \rrbracket_A^{(\upsilon \cup \{n\}, \kappa \cup \{A', n \bullet n\})} & \text{if } A \neq A' \wedge n \notin \upsilon \end{cases} \\
\llbracket \textbf{private } k; P \rrbracket_A^{(\upsilon,\kappa)} & \overset{\text{def}}{=} & \llbracket P \rrbracket_A^{(\upsilon,\kappa)} \\
\llbracket A' \rightsquigarrow B : M; P \rrbracket_A^{(\upsilon,\kappa)} & \overset{\text{def}}{=} & \begin{cases} \overline{B}\,\langle M \rangle . \llbracket P \rrbracket_A^{(\upsilon,\kappa)} & \text{if } A = A' \\ A(x).\text{if } \phi \text{ then } \llbracket P \rrbracket_A^{(\upsilon \cup \{x\}, \kappa')} \text{ else } \mathbf{0} & \text{if } A = B \\ \llbracket P \rrbracket_A^{(\upsilon,\kappa)} & \text{otherwise} \end{cases} \\
 & & \text{where } A' \neq B \wedge \mathcal{S}(M, \kappa(A')) \neq \emptyset \wedge x \notin \upsilon \\
 & & \text{and } K = \mathcal{A}(\kappa(A) \cup \{M \bullet x\}) \\
 & & \text{and } \phi = \Phi(K) \\
 & & \text{and } \kappa' = (\kappa \setminus \kappa(A)) \cup \left\{\, A, k \;\middle|\; k \in \mathcal{I}(K) \,\right\}
\end{array}
$$

Table 4.12: The projection of principals to processes.

By placing the projections of the principals of an equational protocol narration P in a parallel composition, we end up with an applied pi process that represents the intended execution of P.

Definition 13 (Compilation)**.** The translation $\mathcal{T}[\![P]\!]$ of a equational protocol narration P into an applied pi calculus process is defined in Table 4.13, where $(\nu n)_{n\in I}$ denotes n-ary restriction and $\prod_{n\in I}$ denotes n-ary parallel composition. $Ag(P)$ is the set of principals acting in the narration P. $R(P)$ is the set of restricted names excluding generated names, as defined in Table 4.14.

$$\mathcal{T}[\![P]\!] \overset{\text{def}}{=} (\nu n)_{n\in R(P)} \quad \prod_{A\in Ag(P)} ! [\![P]\!]_A^{(R(P),\emptyset)}$$

Table 4.13: The translation into applied pi calculus.

Unlike the translation to spi processes in [BN07], we have the projections of the principals under replication, in order to simulate an arbitrary number of protocol runs, which gives us a clear separation of generated and non-generated restricted names.

$$\begin{aligned}
R(\epsilon) &\overset{\text{def}}{=} \emptyset \\
R(f/i;P) &\overset{\text{def}}{=} R(P) \\
R(M = N;P) &\overset{\text{def}}{=} R(P) \\
R(A\ \textbf{knows}\ M;P) &\overset{\text{def}}{=} R(P) \\
R(A\ \textbf{generates}\ n;P) &\overset{\text{def}}{=} R(P) \quad \text{if } n \notin R(P) \\
R(\textbf{private}\ k;P) &\overset{\text{def}}{=} \{\, k \,\} \cup R(P) \\
R(A \rightsquigarrow B : M;P) &\overset{\text{def}}{=} R(P)
\end{aligned}$$

Table 4.14: The set of restricted names in a narration.

Chapter 5

Implementation

We have implemented the compilation in our compiler `narcapi`. The compiler is written in OCaml[INR] and is available at `https://github.com/samolesen/narcapi`. In this chapter, the derivations between the formal compilation and the implemented compilation are presented.

5.1 Syntax

For convenience, we extend the formal syntax in Table 4.1 with some syntactical sugar, resulting in the syntax shown in Table 5.1.

$$
\begin{aligned}
M, N &::= a \mid A \mid (\,\tilde{M}\,) \mid f(\,\tilde{M}\,) && \text{(terms)}\\
E &::= A \rightsquigarrow B : M \mid E; E && \text{(exchanges)}\\
D &::= \tilde{A}\ \textbf{knows}\ M \mid A\ \textbf{generates}\ n \mid \textbf{private}\ M \mid \tilde{A}\ \textbf{share}\ M \mid D; D && \text{(declarations)}\\
T &::= M = N \mid f/i \mid T; T && \text{(equational theory)}\\
P &::= T; D; E; \epsilon && \text{(equational protocol narrations)}
\end{aligned}
$$

Table 5.1: Implemented syntax for equational protocol narrations.

The implemented syntax supports the macros in Table 2.4, with the further addition that the declarations **private** and **share** support terms instead of just names, such that all the names in the terms become initially inaccessible to an attacker. A function f with arity 0, i.e. a constant function, is written with parentheses $f()$ to catch some human errors, where the intended constant function f is interpreted as a variable due to a missing function declaration $f/0$. While tuples can be expressed through functions as, e.g. nested ordered pairs, we explicitly support tuples for convenience, which requires special handling for projection. For readability, we sometimes use line breaks as separators and omit the trailing ϵ.

5.2 Data Representation

As a part of narcapi, we have implemented a parser for the syntax in Table 5.1 using ocamllex and ocamlyacc[Ler14]. The result of the parser is a structure of the type narration, which is our internal representation of an equational protocol narration. The type narration is defined in Listing 5.1 together with its dependencies.

```
type term =
  | Variable of string
  | Tuple of tuple
  | Function of string * tuple
and tuple = term list

type equation = Equation of term * term

type exchange = Exchange of string * string * term

type access_level =
  | Public
  | Private

type signature_table = (string, int) Hashtbl.t
type name_table = (string, access_level) Hashtbl.t
type knowledge_table = (term, term) Hashtbl.t
type agent_table = (string, knowledge_table) Hashtbl.t

type narration =
  {
    signatures : signature_table;
    equations : equation list;
    names : name_table;
    agents : agent_table;
    exchanges : exchange list
  }
```

Listing 5.1: Abstract Syntax Tree.

Note that while OCaml is primarily a functional language, we use the mutable hash table Hashtbl from OCaml's standard library[Ler14] to represent mappings for both performance and convenience. The types int, string, and 'a list, are some of OCaml's internal types[Ler14].

5.3 Deviations from the Formal Translation

While most of the formulae in chapter 3 are consistent with an implemented counterpart in narcapi, some tricks are utilised only in the implementation. The primary deviation being that checks are generated as a part of the analysis.

The `analyse` function

The `analyse` function shown in Listing 5.2 performs both term derivation and check generation. The names of parametrised expressions have been highlighted for readability.

```
 1 let analyse knowledge equations =
 2   let rec analyse knowledge equations checks =
 3     let tuple_lets = divide_tuples knowledge in
 4     let prune_checks = prune knowledge equations in
 5     let old_size = Hashtbl.length knowledge in
 6     let new_checks = tuple_lets @ checks @ prune_checks
         @ analyse_step knowledge equations in
 7     let new_size = Hashtbl.length knowledge in
 8     if old_size = new_size then new_checks
 9     else analyse knowledge equations new_checks in
10   analyse knowledge equations []
```

Listing 5.2: Implementation of the `analyse` function.

`analyse` Given a knowledge set `knowledge` and an equational theory `equations`, the `analyse` function returns a list of checks and tuple projections. The `analyse` function builds the returned list recursively through the function of the same name in line two, which takes an extra parameter `checks` for the accumulated result.

`divide_tuples` Tuple projections of a tuple t are possible in applied pi processes through pattern-matching bindings of the following form:

$$\text{let } (x_1, \cdots, x_n) = t \text{ in } P$$

In conventional applied pi calculus theory, this process becomes the null process, when t is not a tuple of length n, thereby acting as a well-formed test for tuple projection. The tuple projections that can be generated for a knowledge set are computed by the function `divide_tuples`, which is called in the third line of the `analyse` function.

`prune` The `prune` function is the implementation of the *irreducibles* formula $\mathcal{I}(K)$, which removes terms t from the knowledge set K that can be synthesised by the remaining terms $K \setminus \{ t \}$, with the addition that a check $[r = r']$ is generated for each removed term $e \bullet r$ such that $r' \in \mathcal{S}(e, K \setminus \{ e \bullet r \})$.

`analyse_step` The formula $\mathcal{A}_{n+1}(K)$, corresponds to the `analyse_step` function, which is shown in Listing 5.3 along with an accompanying description. It returns a well-formed test $[r = r]$ for any deduced piece of knowledge $e \bullet r$ that already exists in K.

`@` operator The built-in infix binary operator `@`[Ler14] performs list concatenation.

The recursive `analyse` function terminates when the size of the knowledge set is the same before and after attempted knowledge deduction using the `analyse_step` function. The structure of the `analyse` function limits the generation of unnecessary well-formed tests by only generating well-formed tests for tuples in `divide_tuples` and pieces of knowledge that are not removed by the `prune`. We show this property in the two following examples:

Example (Symmetric Deterministic Encryption). We define the decryption dec of a deterministic encrypted enc message x with key y scheme as follows:

$$\mathsf{dec}(\mathsf{enc}(x, y), y) = x$$

Given the knowledge set $\{\, \mathsf{enc}(m, k) \bullet a, k \bullet b \,\}$, an of the recursive `analyse` function increases the knowledge set to $\{\, \mathsf{enc}(m, k) \bullet a, k \bullet b, m \bullet \mathsf{dec}(a, b) \,\}$ and `analyse` is called with the new knowledge set. The `prune` function removes $\mathsf{enc}(m, k) \bullet a$, because $\mathsf{enc}(m, k)$ it can be represented by $\mathsf{enc}(\mathsf{dec}(a, b), b)$ resulting in the check $[a = \mathsf{enc}(\mathsf{dec}(a, b), b)]$. This check is then returned because no new knowledge can be deduced by `analyse_step`. No well-formed test is generated.

Example (Symmetric Non-deterministic Encryption). The equation for the symmetric deterministic encryption scheme becomes non-deterministic by adding a nonce z:

$$\mathsf{dec}(\mathsf{enc}(x, y, z), y) = x$$

Given the knowledge set $\{\, \mathsf{enc}(m, k, n) \bullet a, k \bullet b \,\}$, an iteration of the `analyse` function increases the knowledge set to $\{\, \mathsf{enc}(m, k, n) \bullet a, k \bullet b, m \bullet \mathsf{dec}(a, b) \,\}$ and `analyse` is called with the new knowledge set. As the nonce n is not in the knowledge set, the ciphertext a is not pruned, unlike in the previous example. Because $m \bullet \mathsf{dec}(a, b)$ can again be deduced from the knowledge set by `analyse_step`, the well-formed test $\mathsf{dec}(a, b) = \mathsf{dec}(a, b)$ is generated, which is then returned since no new knowledge is added to the knowledge set.

The `analyse_step` function

The `analyse_step` function corresponds to the formula $\mathcal{A}_{n+1}(K)$, but generates a well-formed test $[r = r]$ for any $e \bullet r \in \mathcal{A}_n(K) \cap \mathcal{A}_{n+1}(K)$. The implementation of `analyse_step` is shown in Listing 5.3.

`analyse_step` Given a knowledge set `knowledge` and an equational theory `equations`, the `analyse_step` function performs an iteration of knowledge derivation.

`derive_from_equation` The `derive_from_equation` function performs knowledge derivation for an equation in the equational theory.

`find_substitutions` Corresponding to the formula $\mathcal{F}(e_l, e_r, K)$ in Table 4.6, the `find_substitutions` function returns possible substitutions for an equation's right-hand side, where its left-hand side can be synthesised.

`create_representations` Given a substitution and and equation, the function `create_representations` creates possible representations for the right-hand side of the equation.

```
let analyse_step knowledge equations =
  let derive_from_equation res eq =
    let substitutions =
      find_substitutions eq knowledge equations in
    let create_representations res sub =
      let Equation (eq_left, eq_right) = eq in
      let exp = apply_substitution eq_right sub in
      let representations = synthesize knowledge
        equations (apply_substitution eq_left sub) in
      List.fold_left
        (add_check_or_knowledge knowledge equations exp)
        res representations in
    List.fold_left
      create_representations res substitutions in
  List.fold_left derive_from_equation [] equations
```

Listing 5.3: Implementation of the `analyse_step` function.

`apply_substitution` The application $t\sigma$ of a substitution σ to a term t, is implemented as the function `apply_substitution`.

`synthesize` The `synthesize` function is the implementation of the synthesis formula $\mathcal{S}(t, K)$, which generates representation for a term t given a knowledge set K.

`add_check_or_knowledge` Given a knowledge set K and piece of knowledge $e \bullet r$, the `add_check_or_knowledge` function returns $K \cup \{ e \bullet r \}$ when e cannot be synthesised from K, otherwise a well-formed test is generated.

`List.fold_left` Given a function f, accumulator a, and a list of elements l, the built-in function `List.fold_left`[Ler14] will call the function f with a and the first element in l producing a'. The function f is then called with a' and the next element in l, etc. When f has been called for each element in l, the accumulated value is returned.

5.4 Examples

To showcase the `narcapi` compiler, we include the compilation of two different protocols: the Wide Mouth Frog protocol[BAN90] and the Diffie-Hellman key exchange[DH76].

The Wide Mouth Frog protocol

Briais and Nestmann use the Wide Mouth Frog protocol[BAN90] as an example of their translation to spi calculus in [BN07]. By adding an equational theory for symmetric encryption it becomes the equational protocol narration shown in Listing 5.4. After compilation, it becomes the process in Listing 5.6.

```
dec(enc(x, y), y) = x

A,B,S know (A,B,S)
A,S share kAS; B,S share kBS
A generates kAB
A generates m

A->S: (A, enc((B, kAB), kAS))
S->B: enc((A, B, kAB), kBS)
A->B: enc(m, kAB)
```

Listing 5.4: The Wide Mouth Frog protocol narration.

Diffie-Hellman Key Exchange

We use the Diffie-Hellman key exchange as an example of a simple protocol that cannot be compiled by `spyer`, because it depends on Diffie-Hellman exponentiation. Further more, we use non-deterministic symmetric encryption, which is also not supported by `spyer`. Its equational protocol narration is shown in Listing 5.5. Note that, the compiled process in Listing 5.7 only performs a single check. This is a well-formed test, which verifies that the message sent by A can be decrypted using the generated shared key `exp(exp(g(),`kA`),`kB`)`.

```
exp(exp(g(), x), y) = exp(exp(g(), y), x)
dec(enc(x, y, z), y) = x

A,B know (A,B)
A generates kA; B generates kB
A generates m; A generates n

A->B: exp(g(), kA)
B->A: exp(g(), kB)
A->B: enc(m, exp(exp(g(), kB), kA),n)
```

Listing 5.5: Diffie-Hellman Key Exchange.

```
fun dec/2.
fun enc/2.
equation dec(enc(x, y), y) = x.

let A =
  new kAB;
  new m;
  out(S, (A, enc((B, kAB), kAS)));
  out(B, enc(m, kAB)).

let S =
  in(S, x_4);
  let (B_7, kAB_8) = dec(x_6, kAS) in
  let (A_5, x_6) = x_4 in
  if A_5 = A then
  if x_6 = enc((B, kAB_8), kAS) then
  if B_7 = B then
  out(B, enc((A, B, kAB_8), kBS)).

let B =
  in(B, x_9);
  let (A_10, B_11, kAB_12) = dec(x_9, kBS) in
  if x_9 = enc((A, B, kAB_12), kBS) then
  if A_10 = A then
  if B_11 = B then
  in(B, x_13);
  if x_13 = enc(dec(x_13, kAB_12), kAB_12) then 0.

process
  new kAS;
  new kBS;
  !(A | S | B)
```

Listing 5.6: The Wide Mouth Frog process.

```
fun g/0.
fun exp/2.
equation exp(exp(g(), x), y) = exp(exp(g(), y), x).
fun dec/2.
fun enc/2.
equation dec(enc(x, y), y) = x.

let A =
  new kA;
  new m;
  out(B, exp(g(), kA));
  in(A, x_5);
  out(B, enc(m, exp(x_5, kA))).

let B =
  new kB;
  in(B, x_4);
  out(A, exp(g(), kB));
  in(B, x_6);
  if dec(x_6, exp(x_4, kB)) = dec(x_6, exp(x_4, kB))
      then 0.

process
  !(A | B)
```

Listing 5.7: Diffie-Hellman Key Exchange process.

Chapter 6

Conclusion

The focus of this report was to define a translation from protocol narrations to applied pi processes, as well as automatic compilation to ease the process of designing and analysing protocols.

After presenting th background theory for our work in chapter 2, we examine the obstacles of adding equational theory to protocol narrations chapter 3, where we also proved that the term deriviation problem is NP-complete. The syntax of equational protocol narrations is defined in section 4.1, which is followed by the definition of analysis and synthesis in section 4.2, generation of checks in section 4.3, and compilation to applied pi processes in section 4.4. The compilation is implemented in our compiler `narcapi`, that compiles equational protocol narrations into `ProVerif`'s applied pi syntax. The implementation is described in chapter 5 with a focus on deviations to the formal translation.

6.1 Future Work

Due to time constraints, we had to make some compromises that could be interesting to explore in future projects.

Correctness of compilation Due to the complexity of our compilation, and the fact that we have nothing to compare it to, it is difficult to verify its correctness.

Automatic conversion from equations to rewrite systems Our compilation currently requires that the equational theory consists of left-to-right oriented rewrite rules. This is also the case when defining equations in `ProVerif`'s applied pi syntax[Bla01]. However, it should be possible to automatically convert a set of equations to an equivalent rewrite system[DJ90].

Order on terms By introducing an order on terms, it might be possible to replace the formula for term variants with normalisation, which would significantly reduce the space complexity for some input.

Bindings for complex terms For some equational protocol narrations, term deduction results in some very complex terms. It will in some cases be beneficial to bind these terms to a variable, in order to decrease the size of the generated process.

Simplifying checks The generated set of checks can, in many cases, be simplified, which will decrease the size of the generated processes. We already avoid many redundant checks, but further reduction is possible, as described in [BN07].

Bibliography

[AF01] Martín Abadi and Cédric Fournet. "Mobile values, new names, and secure communication". In: *ACM SIGPLAN Notices* 36.3 (2001), pp. 104–115.

[AG97] Martín Abadi and Andrew D Gordon. "A calculus for cryptographic protocols: The spi calculus". In: *Proceedings of the 4th ACM conference on Computer and communications security.* ACM. 1997, pp. 36–47.

[BAN90] Michael Burrows, Martín Abadi, and Roger Needham. "A logic of authentication". In: *ACM Transactions on Computer Systems* 8 (1990), pp. 18–36.

[Bla01] Bruno Blanchet. "An efficient cryptographic protocol verifier based on Prolog rules". In: *Computer Security Foundations Workshop.* IEEE Computer Society. 2001, pp. 0082–0082.

[BN07] Sébastien Briais and Uwe Nestmann. "A formal semantics for protocol narrations". In: *Theoretical Computer Science* 389.3 (2007), pp. 484–511.

[Bri08] Sébastien Briais. *spyer user's guide.* 2008. URL: `http://sbriais.free.fr/tools/spyer/`.

[DH76] Whitfield Diffie and Martin E Hellman. "New directions in cryptography". In: *Information Theory, IEEE Transactions on* 22.6 (1976), pp. 644–654.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. "Rewrite Systems". In: *Handbook of Theoretical Computer Science.* Vol. B: Formal Methods and Semantics. 1990, pp. 243–320.

[HYC08] Kohei Honda, Nobuko Yoshida, and Marco Carbone. "Multiparty asynchronous session types". In: *ACM SIGPLAN Notices* 43.1 (2008), pp. 273–284.

[INR] INRIA. *OCaml.* URL: `https://ocaml.org`.

[Ler14] Xavier Leroy. *The OCaml system release 4.02: Documentation and user's manual.* 2014. URL: `http://caml.inria.fr/pub/docs/manual-ocaml/`.

[Mod14] Paolo Modesti. "Efficient Java Code Generation of Security Protocols Specified in AnB/AnBx". In: *Security and Trust Management.* Vol. 8743. Springer, 2014, pp. 204–208.

[Ole15] Sam Sepstrup Olesen. *Cryptographic Protocols for Election Monitoring.* 2015.