ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bachelor's Thesis

Converting Alice&Bob Protocol Specifications to Tamarin

Michel Keller
mickell@student.ethz.ch

Supervisor:	Dr. Ralf Sasse
Professor:	Prof. Dr. David Basin
Issue Date:	February 10th, 2014
Submission Date:	August 8th, 2014

Abstract

Security protocols are cryptographic protocols that achieve certain properties such as secrecy and authentication. These properties should be guaranteed even if the security protocol is executed over an insecure network with potential interference of an adversary. A flaw in such a protocol can have disastrous impact on both business and civil infrastructure. It is therefore desirable to have formal proofs of the security goals of a protocol, a task for which automated verification tools are often used. All of the available verifiers use their own input language for specifying both the protocol and its desired properties. While this has the advantage that protocol specifications can be tailored to specific properties of the proving theory, it comes at the cost that using more than one verifier can be a cumbersome and time-consuming process.

Protocols are often outlined in so-called Alice&Bob notation that describes the messages that are exchanged between honest principals in successful protocol runs. Alice&Bob notation aims at brevity and readability, not formal preciseness, and therefore its meaning is usually only clear when considered in a context. This makes it unsuitable as an input language at first glance. Still, some researchers have investigated ways of formalizing it and proposed concrete semantics for Alice&Bob notation.

In this thesis, we present the tool-independent A&B protocol specification language that closely resembles Alice&Bob notation and is based on the results of previous work. We try to find a balance between the expressiveness, flexibility and simplicity of Alice&Bob notation and the unambiguous semantics of other protocol specification languages. In particular, a protocol is usually much shorter in A&B than in other languages.

Furthermore, we implement a translator that takes A&B protocol specifications and translates them to the input language of Tamarin. The translator first compiles the A&B input to the intermediate representation format (IR) that specifies the actions that principals have to take during a run of the protocol. The IR is designed as a basis for convenient translation to any protocol specification language. In our case, the IR is then translated to the input language of Tamarin.

Acknowledgements

First and foremost, I would like to thank my supervisor, Dr. Ralf Sasse. His support went far beyond what I could have expected. The advice he gave in our regular meetings was very valuable and he managed to keep me motivated during the last six months. His expertise has spared me quite some research, the write-up is better because of his proof-reading and our discussions were often enlightening and brought new insights and ideas. His help and dedication has improved the final result with certainty. Thank you!

I would also like to express my thanks to the Information Security Group at ETH Zurich and Prof. David Basin in particular for providing me with the opportunity to write this thesis. It has been a highly interesting topic and I was granted a lot of freedom.

Contents

1 Introduction

In recent years, the amount of data transported over insecure networks such as the internet has increased enormously and further massive growth can be expected for the time to come. The transmitted data is often sensitive and protocols that guarantee certain security properties such as integrity and secrecy of data or authentication of participating entities are therefore widely applied. The correctness of these protocols is of utmost importance since vulnerabilities can cause severe damage. The design of security protocols, however, has proven to be a delicate topic. Hundreds of flaws have been discovered in protocols that used to be widely applied for many years. This has led to the development of a number of verification theories and automated verification tools such as ProVerif [2], Scyther [4], Maude-NPA [5], and Tamarin [13] that can handle an ever larger number of protocols. These tools are based on their own verification theories that come with strengths and weaknesses, and it is therefore often desirable to work with more than one tool. This can however be a time-consuming process since all of the aforementioned tools use their own input language. While this allows the specification to be tailored for the internal verification theory of the concrete tool, it requires that a protocol is specified many times.

It is therefore desirable to have a protocol-independent input language that can be translated to the input languages of different verification tools. One of the most common ways of describing security protocols is Alice&Bob notation that specifies a protocol by explicitly stating the message exchange steps and the involved principals. The following example expresses that Alice sends *message* to Bob:

$$Alice \rightarrow Bob \ : \ message.$$

Alice&Bob notation is probably the most intuitive, descriptive and readable way of specifying a protocol. However, the devil is in the details; the meaning of a protocol specification can be ambiguous and dependent on a context. Several researchers have investigated ways of formalizing the semantics of Alice&Bob notation and making explicit what is implicit (that is, the context). Caleiro, Viganò and Basin [3] provide a complete operational semantics based on the spi calculus that formalizes how principals construct and parse messages and makes explicit what checks can be done by honest principals for ensuring that there was no involvement of the adversary. They assume a Dolev-Yao style adversary (we will do the same when generating Tamarin code). Mödersheim [11] provides a more general approach that is based on a specification of the algebraic properties of the messages that can be used in a protocol. Their work provides a thorough foundation for devising a concise and tool-independent protocol specification language that is based on Alice&Bob notation.

1.1 Related Work

Input Languages. All major automated verification tools use their own input language that have been designed with the verification theory of the tool in mind. Most of these languages look completely different from Alice&Bob notation at first glance. However, the concepts are often not unlike the core idea of Alice&Bob notation, namely that communication is modeled by stating the messages that are sent and received by principals participating in a protocol run. Most input languages do not explicitly pair sender and receiver as is the case in Alice&Bob notation, though.

In Maude-NPA [5], for instance, protocols are specified by defining *strands*, a concept very similar to the *roles* that we use in this thesis. A strand states a sequence of sending and receiving steps (along with other information) from the point of view of one participant. Similarly, protocols in Scyther [4] are specified by explicitly stating which actions (sending, receiving, generation of fresh numbers and claims of security properties) have to be taken by principals. Scyther-proof [8] is a tool based on a proof-generating variant [9] of the verification theory underlying Scyther. Its input language uses proper Alice&Bob-style notation for specifying protocols.

Tamarin's [13] input language, on the other hand, is based on specifying rewriting rules for multisets of so-called facts. State is usually expressed with the help of user-defined facts, communication by the predefined `In` and `Out` facts that represent sending and receiving actions; hence, in some sense, Tamarin also works by stating the sending and receiving actions.

Even if all of the input languages we just discussed have some aspects in common with Alice&Bob notation, all of them heavily rely on the specification of additional information (such as algebraic properties and typing rules) that has to be stated explicitly. Mödersheim uses an elegant Alice&Bob-style language called AnB in a research paper [12] where the algebraic properties of the message model

are assumed to be fixed and consequently do not have to be included in the protocol specification itself. We try to fuse the different approaches into a more comfortable input language by working with a fixed message model for honest principals in this thesis (leaving the adversary model unspecified).

Formalization of Alice&Bob Notation. We have indicated that Alice&Bob notation suffers from ambiguities in many cases and that Caleiro et al. [3] and Mödersheim [11] are among the researchers that have investigated the semantics of Alice&Bob notation. Let us give a somewhat more detailed overview of their work here.

Caleiro et al. work with a fixed message model. They first propose the *coarse interpretation* that can only cope with a limited number of protocols. Based on the weaknesses of the coarse interpretation, they then proceed to introduce the *fine interpretation* that is based on the growing knowledge that principals have during a run of a protocol. Finally, they provide an operational semantics based on the spi calculus that makes explicit the actions that a principal has to execute. The most important aspect of the operational semantics is that it provides the precise checks that a principal has to perform to ensure that there was no involvement of the adversary.

While Caleiro et al.'s semantics are based on a fixed message model, Mödersheim gives a formalization of Alice&Bob notation that is defined over an arbitrary algebraic theory. However, his method does not directly give us the actions that have to be taken by honest principals.

Our semantics of Alice&Bob protocol specifications are based on the work of Caleiro et al. However, their message model does not contain Diffie-Hellman exponentiation, a feature we would like to include into our semantics. To do this, we use some of the ideas of Mödersheim.

1.2 Structure

In this thesis, we introduce the Alice&Bob-style A&B protocol specification language and implement a translator from A&B to the input language of Tamarin.

We first talk about Alice&Bob notation in general. In particular, we specify our message model that includes Diffie-Hellman exponentiation and introduce and formalize Alice&Bob protocol specifications.

Next, we introduce the A&B protocol specification language that is based on our previous discussion of Alice&Bob notation. Here, we also introduce the security properties that can be declared in A&B.

We then turn towards the implementation of a translator to the input language of Tamarin. To this end, we introduce the *intermediate representation format* (IR) that represents the protocol in a language-independent way and makes explicit what actions have to be taken by principals. The IR is designed in such a manner that translation to any protocol specification language (not necessarily Tamarin) is as straightforward as possible. The actual translator consists of two steps. The first step parses and analyzes A&B input and generates the IR of the protocol. The second step translates from IR to the desired output language, in our case the input language of the Tamarin prover.

Finally, we demonstrate the A&B language and its translation to Tamarin by way of a case study.

1.3 Contributions

Summarizing, our contributions are as follows. We introduce and formalize the Alice&Bob-style A&B language for specifying cryptographic protocols with Diffie-Hellman exponentiation. We present a language-independent intermediate representation format that is designed as a basis for translation to the input languages of a wide range of protocol verification tools. What is more, we implement a translator from A&B to the input language of Tamarin.

2 The Semantics of Alice&Bob Protocol Specifications

In this section we discuss and formalize the semantics of Alice&Bob notation. First, we point out some of the problems that come with Alice&Bob notation and explain the general idea of the formalization. After that we specify the notation that is used in this thesis and its semantics.

2.1 Overview

In Alice&Bob notation, a protocol is specified as a list of *message exchange steps* of the form:

$$A \to B \ : \ msg.$$

Such a protocol specification describes the actions that are performed by honest principals in a successful protocol run. While this is intuitive, it leaves implicit or even unspecified what happens if a protocol run is not successful, for example when the message received by B does not match the expected pattern (this can be due to involvement of the adversary or a transmission error in the network). Moreover, the actions taken by honest principals can be ambiguous; it is not clear how A can construct the message msg or what checks B can perform to ensure that the received message has the expected shape. Caleiro, Viganò and Basin have investigated the semantics of Alice&Bob protocol specifications [3] and provide an operational semantics based on the spi calculus. The semantics of Alice&Bob notation that we use in this thesis is based on their work.

$$(\mathbf{cr_1}) \quad C \to R : \{n\}^{\mathbf{a}}_{pkR}$$
$$(\mathbf{cr_2}) \quad R \to C : \mathbf{h}(n)$$

Figure 1: The CR protocol.

Let us give a simple example to demonstrate the problems mentioned above. Consider the CR protocol specified in Figure 1. At first glance, the meaning of this protocol seems quite clear. The principal in role C sends the nonce n, asymmetrically encrypted (hence the superscript $\mathbf{a}$) with the public key of R, and the principal in role R then responds by returning the hash of n. However, this is only an intuitive interpretation. What the principals have to do in detail is not clear and open to interpretation. In order to send the message $\{n\}^{\mathbf{a}}_{pkR}$ in message exchange step $(\mathbf{cr_1})$, (the principal in role) C first needs to possess $\{n\}^{\mathbf{a}}_{pkR}$. Intuitively, one would assume that C knows both n and pkR and can therefore construct the message. But this is not stated explicitly; it is possible that the designer of the protocol had in mind that C knows neither n nor pkR, but only the message $\{n\}^{\mathbf{a}}_{pkR}$ as a whole; the specification simply does not tell. Apart from that, it is also not clear if n actually is a nonce (even though the choice of name seems to suggest this), and if pkR is the public key of R. The variable n could just as well be a constant, or a publicly known value (which we would not assume of a nonce).

When writing down a protocol in Alice&Bob notation, one usually does this in a context and provides explanations that can help a human reader understand the meaning. Since we want to design a protocol specification language, we cannot allow ambiguities such as in the CR protocol. If we do not want to narrow the family of protocols that can be expressed with our language, we have to allow the specification to provide some additional information about the protocol. Therefore, the Alice&Bob notation that we describe in this section is based on the notion of a *knowledge* that describes the values (we refer to values as *messages*) that are known to a principal at any time during protocol execution. In particular, it is possible to specify what messages are known to a principal at the very beginning, before the first message exchange step happens.

Another aspect that is left implicit in the CR protocol is what the principals do with the messages they receive. If we assume that pkR actually denotes the public key of R and that R knows the corresponding secret key, skR, then she should extract the value of n by decrypting $\{n\}^{\mathbf{a}}_{pkR}$ with this secret key. If we additionally assume that she does not know n initially, this is the only way she is able to get the value of n in order to construct $\mathbf{h}(n)$ for performing her sending action in $(\mathbf{cr_2})$; otherwise, she could not execute the action. For this reason, we need to formalize what new information can be gained by analyzing incoming messages based on the knowledge that a principal has. Furthermore, we introduce the notion of executability that determines if a protocol can be performed by honest principals if there

is no adversary interference. Another aspect that we have to consider is what kinds of messages there are and what capabilities honest principals have when constructing and analyzing messages.

If we return to the receiving action of R at ($\mathbf{cr_1}$), then we can see that there is no way for her to check if the received message was actually sent by C; it might have been changed, replaced or injected into the network by the adversary. There are situations though, where a principal can detect that the adversary has fiddled with a message. For instance, C knows n and can hence construct $\mathbf{h}(n)$ to check if the message he receives in ($\mathbf{cr_2}$) matches the expected value. If the values do not match, C should abort protocol execution.

These checks are essential since they can make it impossible for an adversary to attack an otherwise vulnerable protocol. If we assume a Dolev-Yao adversary model and that the adversary does not know the secret key of R, then he cannot decrypt the message sent by C and, consequently, if C receives the expected answer from the network he can be sure that R now possesses the value n.

From the points illustrated by the example above, we can conclude that a formalization of Alice&Bob notation should be based on the notion of a knowledge that gradually grows during protocol execution, when new messages are received or fresh numbers are generated. It should explicitly state what information is stored, how incoming messages are parsed and compared to the existing knowledge and how messages are composed for sending. Note that Alice&Bob notation is completely independent of the adversary model. What we need to define is not the capabilities of the adversary but the capabilities (and, duties) of the honest principals. The adversary capabilities can be independently specified.

In the rest of this section, we provide a complete formalization of Alice&Bob notation which is the basis for the semantics of the A&B input language and the translation to the intermediate representation format that follow later.

2.2 Messages and Message Model

The exchange of messages is at the very core of Alice&Bob protocol specifications and before we can discuss the actual communication, we have to introduce what types of messages we work with in this thesis and define their properties.

2.2.1 Messages

There are two types of atomic messages: *Agent names* and *numbers*. The set of all agent names is denoted by $Agent$, the set of numbers by Num. We assume both sets to be infinite which means that we never run out of fresh symbols. The two sets are disjoint and the set Num contains the symbol 1. Agent names identify principals that can participate in a protocol run; we assume that all principals are uniquely defined by such an agent name. Numbers are values that are taken by variables such as n or pkR in the CR protocol. Furthermore, there is the set Fun of function symbols that is assumed to be infinite, too. Based on Num, $Agent$ and Fun, we can now define what a message is:

Definition 2.1. *A message is defined by the following rules:*

- *m is a message if $m \in Agent$ or $m \in Num$.*
- *Let m_1 and m_2 be messages. Then the concatenation of m_1 and m_2, denoted by $\langle m_1 \,.\, m_2 \rangle$, is a message. The angles ('$\langle$' and '$\rangle$') can be omitted if this does not lead to ambiguities.*
- *Let m and k be messages. Then the symmetric encryption of m with k, denoted by $\{m\}_k^{\mathbf{s}}$, is a message.*
- *Let m and k be messages. Then the asymmetric encryption of m with k, denoted by $\{m\}_k^{\mathbf{a}}$, is a message.*
- *Let m be a message. Then the hash of m, denoted by $\mathbf{h}(m)$, is a message.*
- *Let m_1 and m_2 be messages. Then the multiplication of m_1 with m_2, denoted by $m_1 \odot m_2$, is a message.*
- *Let m_1 and m_2 be messages. Then the exponentiation of m_1 with m_2, denoted by $m_1^{m_2}$, is a message.*

- *Let m be a message. Then the inverse of m, denoted by m^{-1}, is a message.*
- *Let $m_1, \ldots, m_i$ be messages, $i \in \mathbb{N}$, and let $fun \in Fun$. Then the application of fun on $m_1, \ldots, m_i$, denoted by $fun(m_1, \ldots, m_i)$, is a message.*

The message 1 is contained in Num but cannot be used in Alice&Bob protocol specifications directly. Nonetheless, we need it when it comes to synthesizing messages. We denote the set of all messages by $Msgs$. We would like to mention here that the Alice&Bob notation that is used in this thesis does not support the specification of arbitrary inverse messages like $\langle m_1 \,.\, m_2 \rangle^{-1}$ even though the definition above is more general and allows such messages. We only work with one public and one private key per role that are mutual inverses. We have introduced inversion of messages because it allows us to represent the message model more nicely. Note that this implies that messages can only be *encrypted asymmetrically* with public and private keys; general keys just make no sense since such messages cannot be decrypted.

Let us now define which messages are atomic before we move on to the properties of our message model:

Definition 2.2. *A message m is called* atomic *if $m \in Agent$, or if $m \in Num$.*

For instance, the public key pkR, the nonce n and the role names C and R from the CR protocol above are all atomic messages; $\{n\}^{\mathbf{a}}_{pkR}$ and $\mathbf{h}(n)$ on the other hand are not atomic.

2.2.2 Message Model

Let us now introduce the empty equational theory S in which two messages are equal if and only if they have equal shape (that is, if they are syntactically the same). For example, $m_1 \odot m_2 =_S m_1 \odot m_2$ but $m_1 \odot m_2 \neq_S m_2 \odot m_1$. The examples show that equality by shape does not capture all properties that we would expect of our messages; for instance, we can assume that Diffie-Hellman multiplication is commutative. For this reason, we additionally introduce the equational theory M that specifies a more sophisticated message model. Two messages are equal in equational theory M if they have the same shape, or if one of the following properties applies:

$$\langle m_1 \,.\, \langle m_2 \,.\, m_3 \rangle \rangle =_M \langle \langle m_1 \,.\, m_2 \rangle \,.\, m_3 \rangle \tag{1}$$
$$\{\{m\}^{\mathbf{s}}_k\}^{\mathbf{s}}_k =_M m \tag{2}$$
$$\{\{m\}^{\mathbf{a}}_k\}^{\mathbf{a}}_{k^{-1}} =_M m \tag{3}$$
$$(k^{-1})^{-1} =_M k \tag{4}$$
$$m_1 \odot m_2 =_M m_2 \odot m_1 \tag{5}$$
$$(m_1 \odot m_2) \odot m_3 =_M m_1 \odot (m_2 \odot m_3) \tag{6}$$
$$(m_1^{m_2})^{m_3} =_M m_1^{m_2 \odot m_3} \tag{7}$$
$$m_1 \odot 1 =_M m_1 \tag{8}$$
$$(m_1)^1 =_M m_1 \tag{9}$$
$$1^{m_1} =_M 1 \tag{10}$$

We generalize $=_M$ and $=_S$ in the natural way for sets. We will use equational theory M for messages in this thesis. Nonetheless, sometimes we want to make explicit that two messages have equal (or do not have equal) shape, in which case we use $=_S$ or $\neq_S$ to emphasize this. We have $m_1 =_S m_2 \Rightarrow m_1 =_M m_2$.

In our model, we use the same function for encryption and decryption. Note that there is no explicit way given for signing messages. This can be achieved by using asymmetric encryption with a secret key.

Concatenation is associative according to property (1). Property (2) expresses that a message m that is symmetrically encrypted with a key k can be decrypted with that same key k. Property (3) states that a message m that is asymmetrically encrypted with a key k can be decrypted with the inverse of that key, k^{-1}. Properties (3) and (4) combined imply that $\{\{m\}^{\mathbf{a}}_{k^{-1}}\}^{\mathbf{a}}_k =_M m$, that is, if m is asymmetrically encrypted with the inverse of key k, then the message can be decrypted with the key k itself. From now on, we use $\mathbf{pk}(A)$ and $\mathbf{sk}(A)$ as our notation for the public and the secret key of A where $(\mathbf{sk}(A))^{-1} =_M \mathbf{pk}(A)$ holds, and $\mathbf{k}(A, B)$ to represent the shared symmetric key of A and B. We talk about the notational conventions we use in this thesis in more detail in Section 2.3.4

Properties (5) and (6) express that multiplication is associative and commutative and property (7) brings multiplication and exponentiation into a relationship. Note that properties (5) and (7) imply that $(m_1^{m_2})^{m_3} =_M (m_1^{m_3})^{m_2}$, a property that lies at the heart of the Diffie-Hellman key exchange protocol.

The last three properties, (8), (9), and (10) express that the 1 message is the neutral element of multiplication.

2.3 Alice&Bob Notation

Based on our notion of a message, we can now give a definition of Alice&Bob notation. Before we do this, however, we would like to talk about the knowledge that a principal has during protocol execution and, in particular, about the initial knowledge.

2.3.1 Knowledge and Basic Sets

We have indicated at the beginning of this section that the principals need to remember the messages they acquire during protocol execution. We call the messages that are known to a principal the current *knowledge* of that principal. Knowledge is essential when it comes to constructing messages for sending, analyzing received messages, and performing checks to ensure that there was no involvement of the adversary.

The knowledge of a principal in general is infinite; if Alice acquires only one message m, she can immediately form infinitely many messages, for example by concatenating arbitrarily many m. This is no problem from a mathematical point of view; however, since we need to store this knowledge in memory when translating from an A&B protocol specification to the intermediate representation format, it is necessary to have a finite representation. To achieve this, we use *basic sets* like proposed by Caleiro et al. [3] to represent the knowledge of the principals that participate in a protocol.

Let us start with an example. Principals get hold of new knowledge either by receiving messages from the network or by generating new numbers themselves. When Alice receives a new message, she can analyze it using her current knowledge. Let us assume that she receives the message $\{m\}^{\mathbf{a}}_{\mathbf{sk}(Bob)}$, where $\mathbf{sk}(Bob)$ denotes the secret key of Bob. If she possesses the public key of Bob, $\mathbf{pk}(Bob)$, she can decrypt the message and obtain m. If she additionally knows $\mathbf{sk}(B)$ (which is unlikely in a useful protocol), she can reconstruct $\{m\}^{\mathbf{a}}_{\mathbf{sk}(Bob)}$ from m and $\mathbf{sk}(Bob)$. In this case, there is no reason for Alice to store $\{m\}^{\mathbf{a}}_{\mathbf{sk}(Bob)}$ as a whole since this would not allow her to construct any more messages. If she does not know $\mathbf{sk}(Bob)$, on the other hand, she needs to store both m and $\{m\}^{\mathbf{a}}_{\mathbf{sk}(Bob)}$ because otherwise, she would "forget" the value of $\{m\}^{\mathbf{a}}_{\mathbf{sk}(Bob)}$. The idea behind a basic set is to analyze messages as far as possible and to store only those messages that are necessary so that we never lose the ability to construct a message. We have seen that sometimes (in the case of asymmetric encryption), we have to keep a message even though we can further analyze it (since we cannot reconstruct it otherwise).

Note that messages may be removed from a basic set as it evolves; if Alice does not know $\mathbf{sk}(Bob)$ at first but obtains it some time later, she can remove $\{m\}^{\mathbf{a}}_{\mathbf{sk}(Bob)}$ since it has become synthesizable from m and $\mathbf{sk}(Bob)$.

We revisit basic sets later when we have defined how messages can be analyzed and constructed and formalize the intuition given here.

2.3.2 Initial Knowledge

We have seen that the knowledge that principals have is essential for the meaning of an Alice&Bob protocol specification. It makes sense in almost all situations that all participants know their own name as well as their own public and private key which is why we define the initial knowledge of a principal to be:

Definition 2.3. *The* initial knowledge *of a principal in role P is defined to be the basic set corresponding to the set*

$$\chi_P^{init} := \{P, \mathbf{pk}(P), \mathbf{sk}(P)\} \cup \chi_P^{explicit}.$$

where $\chi_P^{explicit} \subseteq Msgs$, and $\chi_P^{explicit}$ needs to be stated explicitly.

The initial knowledge is the basic set of messages that a principal knows before he or she performs any actions, that is, before the first sending or receiving action. From the definition above, we can see that every participant initially knows his or her own name as well as the public and the secret key. Additional knowledge is added if $\chi_P^{explicit}$ is not empty.

One important aspect of the initial knowledge is that it allows honest principals to get hold of authenticated information. It is a well-known fact that there is no way to construct an authenticated channel from an insecure network without any initial authenticated information. The purpose of the CR protocol is for C to authenticate R; this is only possible if C possesses an authenticated copy of the public key of R so he can be sure that really only R can decrypt the message containing n.

Alice&Bob notation is independent of the adversary model and consequently it does not make sense to talk about the initial knowledge of the adversary here. Still, in most sensible models (for example a Dolev-Yao adversary) one can assume that the identities of the honest principals as well as their public keys are known to the adversary. The A&B protocol specification language that we introduce in the next section is such that the adversary knows exactly this information in the beginning.

2.3.3 Alice&Bob Protocol Specifications

We can now give a definition of Alice&Bob protocol specifications. Such a protocol specification must of course contain the message exchange steps that define the communication in the protocol. We have seen that it is also necessary to include the initial knowledge to achieve a complete specification of the protocol. Since we have defined the initial knowledge to contain some implicit knowledge, only the additional knowledge needs to be declared explicitly.

Definition 2.4. *An* Alice&Bob protocol specification *is a pair* $(Spec, X)$ *where*

- *Spec is a finite sequence* $step_1, \ldots, step_n$ *of message exchange steps, where the message exchange step* $step_t$, $t \in \{1..n\}$, *has the form*

$$(\mathbf{label_t}) \quad S \to R \ (n_1, \ldots, n_v) \ : \ M,$$

  where R *and* S *are distinct role names (symbols from Agent) and* $n_1, \ldots, n_v$ *are distinct variable names.* $\mathbf{label_t}$ *is a* unique *name given to this message exchange step and* M *is a message.*

- X *is a mapping* $Agent \to Msgs$ *from role names to sets of messages representing the explicit initial knowledge of that role in the protocol, that is,* $X(P) = \chi_P^{explicit}$.

Note that we require fresh numbers $(n_1, \ldots, n_v)$ to be stated explicitly. They are assumed to be generated randomly by the sender at the beginning of the step, before the message is constructed and sent. This information is redundant in some sense; since we know the initial knowledge, we can find out if a number is fresh. Nonetheless, we have decided to state fresh numbers explicitly to improve readability and it should also help prevent errors.

The definition says that the fresh numbers need to have *distinct* variable names. This does not only include the current message exchange step, but the complete protocol. No two fresh variables with the same name must appear in a protocol, nor must a variable that appears in the initial knowledge of a principal be redefined as a fresh variable. What is more, a fresh variable name should not coincide with any role name.

The labels have no special purpose, they are there for reference and we omit them in many cases. Also, we drop the parentheses enclosing the fresh variable names if there are none.

2.3.4 Notational Conventions

Alice&Bob protocol specifications rely heavily on implicit notational conventions. In the CR example at the very beginning of this section, the public key of R was denoted by pkR and n indicated that the value was a nonce.

Another example that demonstrates this nicely is the sample implementation of the (flawed) NSPK protocol for ProVerif [7]. While ProVerif does not use Alice&Bob-like notation, many examples contain comments with the protocol in Alice&Bob notation to improve readability. The first two messages exchange steps are denoted as follows:

```
Message 1: A -> S : (A, B)
Message 2: S -> A : { pkB, B }skS
```

It is implicitly clear that `pkB` denotes the public key of B and `skS` the secret key of S. There is also no need to explicitly mention that S is the server and A and B are clients. We need to be a little bit more formal to achieve a computer-interpretable input language, but notational conventions and some short-hands help keep our Alice&Bob notation compact and still precise.

We work with the following notational conventions on messages with regard to an Alice&Bob protocol specification in this thesis:

- Variables representing numbers (elements from the set Num) are denoted by lower case letters, where subscripts may be added. For example, n in message exchange step ($\mathbf{cr_1}$) in Figure 1 on page 9 represents a number. Other variable names that we use include m_1, m_2, a, and b.

- Variables representing principals (elements from the set $Agent$) are denoted by the name of the corresponding role. Role names (and hence variable names representing principals) are denoted as a single capital letter. In the following example the principal in role A sends her own name to the principal in role B.

  $$A \rightarrow B \ : \ A.$$

  Note that the 'A' before the colon denotes the name of a role while the 'A' after the colon denotes the name of the principal that executes role A during a execution of the protocol.

- Constant numbers (the same in every run) are denoted as strings in single quotes. In the following example, the principal in role A sends the constant string 'Hello!' to the principal in role B:

  $$A \rightarrow B \ : \ \text{'Hello!'}.$$

  Constants are taken from the set Num. Note that constants are de-facto public because they are the same in every run of the protocol (in the spirit of Kerckhoffs, we usually assume that all principals, including the adversary, know the complete protocol specification).

- The secret key of the principal in role A is denoted by $\mathbf{sk}(A)$, the corresponding public key is defined as $(\mathbf{sk}(A))^{-1}$ and denoted by $\mathbf{pk}(A)$.

- The shared symmetric key of the principals in the roles A and B is denoted by $\mathbf{k}(A, B)$.

- M denotes any message, where subscripts may be added.

It is important to note that symmetric, public and private keys are atomic, that is, $\mathbf{pk}(A)$, $\mathbf{sk}(A)$ and $\mathbf{k}(A, B)$ are atomic messages for any A and B.

2.4 Capabilities

We can now talk about how principals can construct messages for sending or analyze newly obtained messages based on their knowledge. We introduce what capabilities honest principals have when working with messages and point out some of the challenges that have to be met, in particular when it comes to constructing messages. We are working towards an automated translator and therefore design our model in such a way that it can later be turned into a computer program in a straightforward way.

2.4.1 Synthesization Capabilities

When performing a sending action, a principal first generates the fresh numbers and adds them to his knowledge. After that, he constructs the message (i.e., the fresh values can be used immediately). Messages are constructed by concatenation, symmetric and asymmetric encryption, hashing, function application, exponentiation and multiplication. We now define the set of messages that a principal can construct from his knowledge.

$$
\begin{array}{lll}
(\mathbf{dh_1}) & A \to B \;\; (g, a) & : \langle g \,.\, g^a \rangle \\
(\mathbf{dh_2}) & B \to A \;\; (b) & : g^b \\
(\mathbf{dh_3}) & A \to B \;\; (n) & : \{n\}^{\mathbf{s}}_{g^{a \odot b}}
\end{array}
$$

Figure 2: The Diffie-Hellman key exchange protocol. The principals have no additional initial knowledge.

Definition 2.5. *Let χ be a set of messages. The set synth(χ) is the least superset closed under the rules*

$$\textsc{Sconcat}\ \frac{m_1 \qquad m_2}{\langle m_1 \,.\, m_2 \rangle} \qquad\qquad \textsc{Saenc}\ \frac{m_1 \qquad m_2}{\{m_1\}^{\mathbf{a}}_{m_2}}$$

$$\textsc{Ssenc}\ \frac{m_1 \qquad m_2}{\{m_1\}^{\mathbf{s}}_{m_2}} \qquad\qquad \textsc{Sappl}\ \frac{f_i(\cdot) \qquad m_1 \qquad \ldots \qquad m_i}{f_i(m_1, \ldots, m_i)}$$

$$\textsc{Sexp}\ \frac{m_1 \qquad m_2}{m_1^{m_2}} \qquad\qquad \textsc{Smul}\ \frac{m_1 \qquad m_2}{(m_1 \odot m_2)}$$

$$\textsc{Shash}\ \frac{m_1}{\mathbf{h}(m_1)}$$

where m_1 and m_2 are messages and f_i is a function symbol. We denote manifold application of synth(.) to a set of messages by synth*(.).

The above definition formalizes which messages a principal can construct. Note that in our model, principals cannot construct inverse messages; all inverse messages that appear in a protocol need to have been in the initial knowledge of a principal. This goes nicely with the restriction that we only allow public and private keys of the form $\mathbf{pk}(P)$ and $\mathbf{sk}(P)$ for asymmetric encryption. Since a principal knows his own public and private key implicitly via his knowledge, they are in the system automatically.

We would now like to point out some of the challenges we have to master when formalizing explicitly how a message can be constructed. Let the knowledge of A be the standard knowledge of $\{\mathbf{sk}(A), \mathbf{pk}(A), A\}$. From the rules above, we know that she can execute the action

$$A \to B \;\; (n) \;:\; \{n\}^{\mathbf{a}}_{\mathbf{sk}(A)}$$

by constructing the message by applying the rule Saenc to n (generated before the construction step) and $\mathbf{sk}(A)$.

However, it is not always so clear how a message can be constructed from the current knowledge. Let us take the well-known Diffie-Hellman key exchange protocol for another example. A modified version of it is listed in Figure 2.

In message exchange step ($\mathbf{dh_1}$), A randomly chooses the generator g and her private number a and then sends both g and g^a to B. B receives these numbers and randomly generates his own private number, b, in ($\mathbf{dh_2}$), and returns g^b.

A now possesses g^b and a and can therefore compute $(g^b)^a$ while b possesses g^a and b from which $(g^a)^b$ can be computed. With the properties (6) and (7) from our message model we can conclude that $(g^a)^b \overset{(7)}{=_M} g^{a \odot b} \overset{(6)}{=_M} g^{b \odot a} \overset{(7)}{=_M} (g^b)^a$, in other words, both A and B know $g^{a \odot b}$. Therefore, it is now possible for A to execute the sending action of ($\mathbf{dh_3}$) by generating the new secret number n and then symmetrically encrypt it with $(g^b)^a$. At the receiving side, B can extract n by decrypting the message with key $(g^a)^b$.

We can conclude from the Diffie-Hellman protocol that it is not sufficient to compare messages by pattern matching (equational theory S), we also have to take the message model (equational theory M) into account. This is because the synthesis rules are based on pattern matching while equational theory M specifies that messages can be equal even if they do not have the same shape. We have to discuss how we can determine if a message is constructible from the current knowledge, where we work with $=_M$, and, if the message is constructible, how it can by synthesized. The view that a principal has of synthesizing a message given the current knowledge is expressed by the *constructive form* that we will soon introduce.

Before we continue, let us give the definition of the *basic set* property that we promised earlier:

Definition 2.6. *A set of messages χ is called a* basic set *if for all $m \in \chi$ we have $m \notin_M synth^*(\chi \backslash \{m\})$.*

In other words, a set is a basic set if no message in the set can be synthesized from the other messages in the set and m is therefore not redundant information. The following proposition states that basic sets are unique.

Proposition 2.7. *Let χ_1 and χ_2 be sets of messages and let χ_1' and χ_2' be basic sets with $synth^*(\chi_1) =_M synth^*(\chi_1')$ and $synth^*(\chi_2) =_M synth^*(\chi_2')$. Then $synth^*(\chi_1) =_M synth^*(\chi_2)$ if and only if $\chi_1' =_M \chi_2'$.*

We provide proof sketches for the propositions we state in this thesis in Appendix A. In these we argue why the statements we make are correct. However, we do not give formal proofs.

Proposition 2.7 provides us with a way for comparing sets of messages. We have mentioned that in the typical case, the knowledge is infinite and have introduced basic sets as a finite representation. Proposition 2.7 tells us that this representation is unique and consequently we can compare knowledge by representing it as basic sets and then comparing the individual elements. We will later show that there is a basic set representation for every set of messages and provide an explicit way of constructing it.

2.4.2 Analysis Capabilities

When a principal receives a message from the network, he or she often not only learns the message itself, but can also extract more information, for example by taking concatenated messages apart or decrypting a message when he knows the corresponding key. Principals have the following analysis capabilities:

Definition 2.8. *Let χ be a set of messages. The set $analyze(\chi)$ is the least superset of M closed under the rules*

$$\textsc{Asenc}\ \frac{\{m_1\}^{\mathbf{s}}_{m_2} \qquad m_2}{m_1} \qquad \textsc{Aaenc1}\ \frac{\{m_1\}^{\mathbf{a}}_{m_2} \qquad m_2^{-1}}{m_1}$$

$$\textsc{Aaenc2}\ \frac{\{m_1\}^{\mathbf{a}}_{m_2^{-1}} \qquad m_2}{m_1} \qquad \textsc{Aconcat1}\ \frac{\langle m_1\ .\ m_2\rangle}{m_1}$$

$$\textsc{Aconcat2}\ \frac{\langle m_1\ .\ m_2\rangle}{m_2}$$

where m_1 and m_2 are arbitrary messages. We denote manifold application of $analyze(.)$ to a set of messages by $analyze^*(.)$.

Let us again demonstrate some of the rules above on the Diffie-Hellman protocol to point out a few of the aspects we have to take into consideration when giving a more explicit description of how principals analyze messages. In message exchange step ($\mathbf{dh_1}$), B receives $\langle g\ .\ g^a\rangle$ on which he can apply the rule Aconcat1 to extract g and Aconcat2 to learn g^a. He can then try to analyze g^a even further to extract a too. However, there is no rule for this (honest principals can of course not solve the discrete logarithm problem).

In ($\mathbf{dh_3}$), B receives $\{n\}^{\mathbf{s}}_{g^{a\odot b}}$; he can obtain n by decrypting it with the key $g^{a\odot b}$. We have shown above that he can construct this message from the knowledge he has at that time. We can see here that synthesis and analysis of messages are not at all independent of each other but may be intertwined; it may well be necessary to synthesize a message in order to analyze another message. This leads to the following definition:

Definition 2.9. *Let χ be a set of messages. The least superset of χ that is closed under both the analysis and the synthesis rules is denoted by $close(\chi)$.*

2.4.3 Ghost Symbols

Let us stay with the last message exchange step of the Diffie-Hellman key exchange protocol in Figure 2 yet a little longer. There, A has to construct the message $(g^b)^a$. In step ($\mathbf{dh_2}$), A received g^b from B. Since we assume that honest principals cannot solve the discrete logarithm problem (that is, the analysis rules we have defined do not provide honest principals this capability), there is no way for A to get hold

of b and therefore, the knowledge of A (a basic set) contains the message g^b. A can then construct $(g^b)^a$ by taking g^b to the power of a.

From $(g^b)^a$ alone, however, we cannot read off how A constructs the message. A could just as well possess g, a and b and compose the message by first taking g to the power of b and taking the resulting message to the power of a. As a remedy for this ambiguity, we introduce the *ghost abstraction*. From now on, we write every non-atomic message M that occurs in the knowledge of a principal as γ_M, where γ is called the ghost symbol. Using this notation, the message that is sent by A would be represented as $(\gamma_{g^b})^a$. This tells us that A possesses γ_{g^b} and a and constructs $(g^b)^a$ by calculating the exponentiation of γ_{g^b} with a. The purpose of ghost abstraction is to express how a message can be constructed, it does not change the message itself. Therefore, we have for every message M that $M =_S \gamma_M$.

Caleiro et al. [3] use a very similar construct of the same name. However, ghost symbols are used in a slightly different manner in this thesis since we additionally work with a more complex message model.

2.4.4 Multiset Representation of Multiplication and Canonical Form of Messages

We have indicated that we cannot work with pattern matching alone when it comes to synthesizing and analyzing messages but have to include the equality properties of our message model as well. We tackle this challenge by introducing a *canonical form* for messages. First, we note that multiplication is both associative and commutative. In other words, the order in which the terms are multiplied does not matter at all and it is consequently sufficient to know which terms are to be multiplied with one another. We exploit this property by using a multiset representation for multiplication from now on in this thesis. This multiset representation is also used in our data structure for protocols in the implementation of the translator. Let us illustrate this with an example. Suppose, the message $(a \odot (b \odot c)) \odot a$ appears somewhere in a protocol. We represent this as $\odot\{\!|a, a, b, c|\!\}$, where $\{\!|.|\!\}$ denotes a multiset. This multiset representation works very well with properties (5) and (6), namely commutativity and associativity of multiplication.

Note that in multisets the multiplicity of an element matters ($\{\!|a|\!\} \neq \{\!|a, a|\!\}$) while the order in which the elements are listed does not matter ($\{\!|a, b|\!\} = \{\!|b, a|\!\}$). Equality with respect to equational theories S and M is extended to multisets in the natural way . For example, we have $\{\!|a, a \odot 1, 1 \odot a|\!\} \neq_S \{\!|a, a, a|\!\}$ but $\{\!|a, a \odot 1, 1 \odot a|\!\} =_M \{\!|a, a, a|\!\}$.

To cope with the exponentiation property (7), we require that all messages of the form $(a^b)^c$ are written as $a^{b \odot c}$. Using these conventions, we write all of $(a^b)^{c \odot d}$, $((a^b)^c)^d$, and $((a^d)^b)^c$ as $a^{\odot\{\!|b,c,d|\!\}}$, making obvious that they are equal.

Let us now formalize this.

Definition 2.10. *A message m is in its* canonical form *if and only if any of the following holds:*

- *m is an atomic message.*
- *m has the shape $\langle m_1 \,.\, m_2 \rangle$ and both m_1 and m_2 are in canonical form and m_1 is no concatenation (i.e., concatenation is represented in a right-associated way).*
- *m has the shape $\{m_1\}^{\mathbf{a}}_{m_2}$ and both m_1 and m_2 are in canonical form and m_1 does not have the shape $\{m_3\}^{\mathbf{a}}_{m_4}$ with m_4 the inverse message of m_2 (e.i., no asymmetric encryptions that cancel each other).*
- *m has the shape $\{m_1\}^{\mathbf{s}}_{m_2}$ and both m_1 and m_2 are in canonical form and m_1 does not have the shape $\{m_3\}^{\mathbf{s}}_{m_2}$ (i.e., no symmetric encryptions that cancel each other).*
- *m has the shape $\mathbf{h}(m_1)$ and m_1 is in canonical form.*
- *m has the shape $foo(m_1, \ldots, m_i)$, $i \geq 0$, for a function symbol foo, and all $m_1, \ldots, m_i$ are in canonical form.*
- *m has the shape $\odot\{\!|m_1, \ldots, m_i|\!\}$, $i \geq 2$, and all $m_1, \ldots, m_i$ are in canonical form and none of $m_1, \ldots, m_i$ is a multiplicative term or equal to the message 1 (in equational theory M).*
- *m has the shape ${m_1}^{m_2}$ and both m_1 and m_2 are in canonical form and m_1 is no exponential message and neither $m_1 =_M 1$ nor $m_2 =_M 1$.*

- *m has the shape m_1^{-1} and m_1 does not have the shape m_2^{-1} (i.e., no double inversion).*

Note that the multiset representation of multiplication is required to involve at least two terms. The term $\odot\{\!|m|\!\}$ is therefore illegal and has to be represented as m.

We have demonstrated above that $(a^b)^{c\odot d}$, $((a^b)^c)^d$, and $((a^d)^b)^c$ all have the same canonical form $a^{\odot\{\!|b,c,d|\!\}}$. In general, the following procedure can be applied to transform any message m into its canonical form $canonical(m)$, where we assume that no empty multiplications ($\odot\{\!|\!|\!\}$) occur:

- $canonical(m) = m$ if m is atomic.
- $canonical(\langle m_1 \,.\, m_2\rangle) = canonical(\langle m_1' \,.\, \langle m_1'' \,.\, m_2\rangle\rangle)$ if $m_1 =_S \langle m_1' \,.\, m_1''\rangle$, else
  $canonical(\langle m_1 \,.\, m_2\rangle) = \langle canonical(m_1) \,.\, canonical(m_2)\rangle$.
- $canonical(\{m\}_k^{\mathbf{a}}) = canonical(m')$ if $m =_S \{m'\}_{k'}^{\mathbf{a}}$ and $k^{-1} =_M k'$, else
  $canonical(\{m\}_k^{\mathbf{a}}) = \{canonical(m)\}_{canonical(k)}^{\mathbf{a}}$.
- $canonical(\{m\}_k^{\mathbf{s}}) = canonical(m')$ if $m =_S \{m'\}_{k'}^{\mathbf{s}}$ and $k =_M k'$, else
  $canonical(\{m\}_k^{\mathbf{s}}) = \{canonical(m)\}_{canonical(k)}^{\mathbf{s}}$.
- $canonical(\mathbf{h}(m)) = \mathbf{h}(canonical(m))$.
- $canonical(foo(m_1, \ldots, m_i)) = foo(canonical(m_1), \ldots, canonical(m_i))$.
- $canonical(\odot\{\!|m|\!\}) = canonical(m)$, else
  $canonical(\odot\{\!|m_1, \ldots, m_{j-1}, m_j, m_{j+1}, \ldots, m_i|\!\}) = canonical(\odot\{\!|m_1, \ldots, m_{j-1}, m_{j+1}, \ldots, m_i|\!\})$ if $m_j =_M 1$ (note that this gets repeated until there are no more 1 factors or only one single factor remains), else
  $canonical(\odot\{\!|m_1, \ldots, m_{j-1}, m_j, m_{j+1}, \ldots, m_i|\!\}) = canonical(\odot\{\!|m_1, \ldots, m_{j-1}, m_j^1, \ldots, m_j^k, m_{j+1}, \ldots, m_i|\!\})$ if $m_j =_S \odot\{\!|m_j^1, \ldots, m_j^k|\!\}$ (note that this gets repeated until there are no more multiplicative factors or only one single factor remains), else
  $canonical(\odot\{\!|m_1, \ldots, m_i|\!\}) = \odot\{\!|canonical(m_1), \ldots, canonical(m_i)|\!\}$.
- $canonical(m_1^{m_2}) = canonical(m_1)$ if $m_2 =_M 1$, else
  $canonical(m_1^{m_2}) = 1$ if $m_1 =_M 1$, else
  $canonical(m_1^{m_2}) = canonical((m_1')^{\odot\{\!|m_1'', m_2|\!\}})$ if $m_1 =_S (m_1')^{m_1''}$, else
  $canonical(m_1^{m_2}) = canonical(m_1)^{canonical(m_2)}$.
- $canonical(m^{-1}) = canonical(m')$ if $m =_S (m')^{-1}$, else
  $canonical(m^{-1}) = (canonical(m))^{-1}$.

The most interesting aspect here is how we transform a multiplicative message into its canonical form. We have defined that a multiplicative message must not contain a message that is equal to the message 1. However, we cannot just remove all the messages that are 1 from a multiplication. Consider the message $\odot\{\!|1, 1|\!\}$. If we just removed all of the 1-messages, we would get $\odot\{\!|\!|\!\}$, the empty multiplication which is no legal message. For this reason, the above definition only removes one 1 at a time, leading to the evaluation chain $canonical(\odot\{\!|1, 1|\!\}) = canonical(\odot\{\!|1|\!\}) = canonical(1) = 1$. It is important to note that the check if a message is equal to 1 is performed on the canonical form of that message. For example, $canonical(\{\{1^1\}_{\mathbf{pk}(A)}^{\mathbf{a}}\}_{\mathbf{sk}(A)}^{\mathbf{a}}) = canonical(1^1) = canonical(1) = 1$ and therefore $canonical(\odot\{\!|\{\{1^1\}_{\mathbf{pk}(A)}^{\mathbf{a}}\}_{\mathbf{sk}(A)}^{\mathbf{a}}, 1|\!\}) = canonical(\odot\{\!|1|\!\}) = canonical(1) = 1$.

The canonical form and the function *canonical* are very useful when it comes to implementing the translator. Therefore, we would like to state a few properties of them explicitly and investigate how they are related. Let us start with the following lemma:

Lemma 2.11. *Let m be an arbitrary message. Then $canonical(m)$ is in canonical form and $canonical(m) =_M m$.*

We can further show that the canonical form is unique:

Lemma 2.12. *Let m be an arbitrary message. Then the canonical form of m exists and is unique, i.e., there is exactly one message m' in canonical form such that $m =_M m'$.*

The following is a direct corollary of Propositions 2.11 and 2.12.

Proposition 2.13. *Let m_1 and m_2 be messages. We have $m_1 =_M m_2$ if and only if $canonical(m_1) =_S canonical(m_2)$.*

As a consequence, we can compare messages by pattern matching if they are in canonical form. This significantly simplifies matters when implementing the translator.

2.5 Synthesizing Messages

We can now state explicitly how messages are constructed by principals. We first introduce the constructive form that represents the view that a principal has of synthesizing a message, given his or her knowledge. The most challenging part here is how we can synthesize multiplicative and exponential messages with respect to the equational theory M. The canonical form of messages simplifies this significantly.

2.5.1 Constructive Form

Caleiro et al. [3] provide operational semantics for Alice&Bob protocol specifications that are based on the spi calculus. In this context, they introduce the *constructive form* that describes how a message can be constructed from the current knowledge. We take up the same idea here, even though we have to adapt it a little bit since our message model is more complex. Furthermore, our version of the constructive form is not defined if the message cannot be constructed.

We have already introduced the basic idea behind the constructive form in the section about ghost symbols. It is based on the current knowledge of a principal and tells us the view he or she has of constructing that message.

Definition 2.14. *Let χ be a set of messages (usually, but not necessarily a basic set) with $synth^*(\chi) =_M close(\chi)$ and let m be a message represented in canonical form. The* constructive form $cf_\chi(m)$ *is the view that a principal has of constructing m from knowledge χ. It is defined as follows ($constrMultTerm_\chi$ and $constrExpTerm_\chi$ are defined later in Section 2.5.3):*

- $cf_\chi(m) = m$, *if m is atomic and $m \in_S \chi$, else*
  $cf_\chi(m)$ *is undefined.*
- $cf_\chi(\langle m_1 \,.\, m_2\rangle) = \langle cf_\chi(m_1) \,.\, cf_\chi(m_2)\rangle$ *if $cf_\chi(m_1)$ and $cf_\chi(m_2)$ are defined, else*
  $cf_\chi(\langle m_1 \,.\, m_2\rangle)$ *is undefined.*
- $cf_\chi(\{m_1\}^{\mathbf{a}}_{m_2}) = \{cf_\chi(m_1)\}^{\mathbf{a}}_{cf_\chi(m_2)}$ *if $cf_\chi(m_1)$ and $cf_\chi(m_2)$ are defined, else*
  $cf_\chi(\{m_1\}^{\mathbf{a}}_{m_2}) = \gamma_m$ *with $m =_S \{m_1\}^{\mathbf{a}}_{m_2}$ if $m \in_S \chi$, else*
  $cf_\chi(\{m_1\}^{\mathbf{a}}_{m_2})$ *is undefined.*
- $cf_\chi(\{m_1\}^{\mathbf{s}}_{m_2}) = \{cf_\chi(m_1)\}^{\mathbf{s}}_{cf_\chi(m_2)}$ *if $cf_\chi(m_1)$ and $cf_\chi(m_2)$ are defined, else*
  $cf_\chi(\{m_1\}^{\mathbf{s}}_{m_2}) = \gamma_m$ *with $m =_S \{m_1\}^{\mathbf{s}}_{m_2}$ if $m \in_S \chi$, else*
  $cf_\chi(\{m_1\}^{\mathbf{s}}_{m_2})$ *is undefined.*
- $cf_\chi(\mathbf{h}(m_1)) = \mathbf{h}(cf_\chi(m_1))$ *if $cf_\chi(m_1)$ is defined, else*
  $cf_\chi(\mathbf{h}(m_1)) = \gamma_m$ *with $m =_S \mathbf{h}(m_1)$ if $m \in_S \chi$, else*
  $cf_\chi(\mathbf{h}(m_1))$ *is undefined.*
- $cf_\chi(fun(m_1, \ldots, m_i)) = fun(cf_\chi(m_1), \ldots, cf_\chi(m_i))$ *if $cf_\chi(m_1), \ldots, cf_\chi(m_i)$ are defined, else*
  $cf_\chi(fun(m_1, \ldots, m_i)) = \gamma_m$ *with $m =_S fun(m_1, \ldots, m_i)$ if $m \in_S \chi$, else*
  $cf_\chi(fun(m_1, \ldots, m_i))$ *is undefined.*
- $cf_\chi(\odot\{\!|m_1, \ldots, m_i|\!\}) = constrMultTerm_\chi(\odot\{\!|m_1, \ldots, m_i|\!\})$.
- $cf_\chi(m_1^{m_2}) = constrExpTerm_\chi(m_1^{m_2})$

We call a message m synthesizable from knowledge χ *whenever $cf_\chi(m)$ is defined.*

This deserves some explanation. The constructive form is defined exactly when the message can be constructed. Atomic messages cannot be constructed from other messages and hence, if a principal wants to construct an atomic message, he can only do this if this message is in his knowledge. For this reason, the constructive form is only defined if the message is actually contained in the current knowledge χ.

For composed messages, it may be possible to construct them from other messages in the knowledge. Let us start with concatenation; if a principal wants to construct the message $\langle m_1 \,.\, m_2 \rangle$, the only way to go is to first construct m_1 and m_2 and then concatenating them. If this is not possible (that is, the constructive form of m_1 or m_2 is not defined) then the message cannot be synthesized. Note that there will never be be a ghost symbol of the form $\gamma_{\langle m_1 \,.\, m_2 \rangle}$ since concatenation can always be analyzed and the original message can always be reconstructed from the sub-messages. Therefore, it would make no sense to check if $\langle m_1 \,.\, m_2 \rangle$ as a whole is in χ.

In the case of symmetric encryption, we can construct the message if we know both the encryption key and the message that should be encrypted. If either is not in the knowledge, there is still the chance that the principal knows the message as a whole, i.e., $\gamma_{\{m_1\}^{\mathbf{s}}_{m_2}}$ is in χ. The case of asymmetric encryption is analogous.

Hashing and function application are very similar as well. The hash of a function or the application of functions can be constructed if we can construct all the arguments. If this is not the case, it is still possible that the message as a whole is in the knowledge, that is, $\gamma_{\mathbf{h}(m_1)}$ or $\gamma_{fun(m_1,\ldots,m_i)}$, respectively, is in χ. Otherwise, it is not possible to construct the message and the constructive form is undefined.

The final two cases are dedicated to exponentiation and multiplication which are a little bit more involved. We will soon define $constrMultTerm_\chi$ and $constrExpTerm_\chi$, but before we do that, we need to talk about division and reductions.

We should mention here that, in general, the constructive form of a message is not uniquely defined, even when applied to a basic set. For instance, $\chi = \{a, \gamma_{g^a}, b, \gamma_{g^b}\}$ is a basic set since no message can be constructed from the others. In this case, $g^{\odot\{\!|a,b|\!\}}$ can be constructed either as ${\gamma_{g^a}}^b$ or ${\gamma_{g^b}}^a$. We will later see that $constrMultTerm_\chi$ and $constrExpTerm_\chi$ are not deterministic in such cases and consequently cf_χ is not deterministic.

2.5.2 Division, Left and Right Reduction

We have already seen in the Diffie-Hellman example that it is not obvious how exponential and multiplicative messages can be constructed. To relieve this problem, we have introduced the canonical form for messages that ensures that they are always represented in the same way. However, we have not yet completely defined the constructive form since $constrMultTerm_\chi$ and $constrExpTerm_\chi$ are still missing. Before we can give the corresponding definitions, we have to discuss multiplication and exponentiation in some more detail. Namely, if we want to construct a multiplicative or exponential message, we have to do this with respect to properties (5), (6) and (7) on page 11. To this end, we make a distinction between multiplication, left exponentiation and right exponentiation. Let us explain this:

Construction by Multiplication. Consider the message $\odot\{\!|a, b, c|\!\}$. Depending on the knowledge, there are several ways of how a principal can construct this message. Let us approach this systematically by looking at the messages that are in the knowledge of that principal. If he knows a, he can construct the message if he also possesses $\odot\{\!|b, c|\!\}$, that is, if he either has $\gamma_{\odot\{\!|b,c|\!\}}$ in his knowledge or he is able to construct $\odot\{\!|b, c|\!\}$ from other messages by forming the multiplication $a \odot (\odot\{\!|b, c|\!\})$. This is a form of division; if we want to construct the message m and know the message d, we can construct m if we also know $m \div d$ since $m = d \odot (m \div d)$. Let us therefore formalize division with respect to our message model:

Definition 2.15. *Let m and d be messages in canonical form. The* division of m by d, *denoted by $m \div d$, is defined as follows:*

- $m \div d =_S 1$ *if* $m =_S d$, *else*
- $m \div 1 =_S m$, *else*
- $(\odot M) \div (\odot D) =_S canonical(\odot(M \backslash D))$ *if* $D \subset_S M$ *(where M and D are multisets), else*
- $(\odot M) \div d =_S canonical(\odot(M \backslash \{\!|d|\!\}))$ *if* $d \in_S M$ *(where M is a multiset), else*

- $m \div d$ *is undefined.*

We call m divisible by d *if and only if* $m \div d$ *is defined.*

Note that we treat messages under ghost abstraction like normal messages, for instance, $\gamma_{\odot\{\!|a,b,c|\!\}} \div \odot\{\!|b,c|\!\} = a$. We require that the operands of the division are in canonical form because this allows us to always use $=_S$ instead of $=_M$ (compare Proposition 2.13 on page 19) which simplifies matters for the implementation. If the messages are not in canonical form, this can easily be corrected by applying *canonical*. The result of a division is again in canonical form.

Construction by Left Exponentiation. We can apply a similar idea in the case of exponentiation. We call a message m left reducible by a message r if there is a message a such that $m =_M r^a$. Let us illustrate this with an example. Consider the message $b^{\odot\{\!|c,d|\!\}}$ and the message b^d; $b^{\odot\{\!|c,d|\!\}}$ is left reducible by b^d since $b^{\odot\{\!|c,d|\!\}} =_M (b^d)^c$. In this case, we call c the *left reduction* of $b^{\odot\{\!|c,d|\!\}}$ and b^d. Let us put this in a definition:

Definition 2.16. *Let* m *and* r *be messages in canonical form. The* left reduction of m by r*, denoted by* $m \triangleleft r$*, is defined as follows:*

- $m_1^{m_2} \triangleleft r_1^{r_2} = m_2 \div r_2$ *if* $m_1 =_S r_1$ *and* m_2 *is divisible by* r_2*, else*
- $m_1^{m_2} \triangleleft r = m_2$ *if* $m_1 =_S r$*, else*
- $m \triangleleft r$ *is not defined.*

We call m left reducible by r *if and only if* $m \triangleleft r$ *is defined.*

Again, we require that the operands of the left reduction are in canonical form and guarantee that the result is in canonical form.

Construction by Right Exponentiation. Right reduction is the symmetrical case of left reduction. While $b^{\odot\{\!|c,d|\!\}}$ is left reducible by b^d and the left reduction is c, $b^{\odot\{\!|c,d|\!\}}$ is right reducible by c and the right reduction is b^d. That is, a message m is right reducible by a message r if there is a message a such that $m =_M a^r$ (compare to r^a in the case of left reduction).

Definition 2.17. *Let* m *and* r *be messages in canonical form. The* right reduction of m by r*, denoted by* $m \triangleright r$*, is defined as follows:*

- $m_1^{m_2} \triangleright r = m_1^{m_2 \div r}$ *if* m_2 *is divisible by* r*, else*
- $m \triangleright r$ *is undefined.*

We call m right reducible by r *if and only if* $m \triangleright r$ *is defined.*

Again, we require that the operands are in canonical form and guarantee that the result is in canonical form as well. Let us put the most important properties of division, left and right reduction in a lemma:

Lemma 2.18. *Let* m *and* r *be two messages in canonical form. Then we have:*

(i) If m *is divisible by* r*, then we have* $m = r \odot (m \div r)$ *and* $m \div r$ *is in canonical form.*

(ii) If m *is left reducible by* r*, then we have* $m = r^{(m \triangleleft r)}$ *and* $m \triangleleft r$ *is in canonical form.*

(iii) If m *is right reducible by* r*, then we have* $m = (m \triangleright r)^r$ *and* $m \triangleright r$ *is in canonical form.*

2.5.3 Multiplicative and Exponential Terms

We finally come to the point where we can devise algorithms for constructing multiplicative and exponential messages. We heavily rely on the definitions of division and left and right reduction for that. We first discuss multiplication and then exponentiation.

Suppose we want to construct the message $\odot\{\!|m_1, m_2, m_3|\!\}$ from the knowledge $\{\gamma_{\langle m_1\ .\ m_2\rangle}, m_3\}$ which is obviously possible. To this end, we can just iterate over our knowledge and see that $\odot\{\!|m_1, m_2, m_3|\!\}$ is divisible by $\gamma_{\langle m_1\ .\ m_2\rangle}$ and that we can construct $\odot\{\!|m_1, m_2, m_3|\!\}$ as $\gamma_{\langle m_1\ .\ m_2\rangle} \odot m_3$ (Lemma 2.18).

However, it is not always sufficient to iterate over the knowledge and look for a message that divides the message we want to construct. Consider the message $\odot\{\!|\langle m_1\ .\ m_2\rangle, \langle m_3\ .\ m_4\rangle|\!\}$. Obviously, we can construct this message from knowledge $\{m_1, m_2, m_3, m_4\}$ even though it is divisible by none of the messages in the knowledge. We can conclude that in the case of division, it is additionally necessary to check if we can construct any of the factors. If we check if $\langle m_1\ .\ m_2\rangle$ is constructible, we can immediately see that this is possible. Similarly, we can construct $\odot\{\!|\langle m_1\ .\ m_2\rangle, \langle m_3\ .\ m_4\rangle|\!\} \div \langle m_1\ .\ m_2\rangle =_S \langle m_3\ .\ m_4\rangle$.

Of course, it is also possible that a multiplicative message is stored as a ghost term in the knowledge. The three cases we have just discussed lead to the following algorithm that finds out if a message m in canonical form can be constructed from the knowledge χ and returns the constructive form if possible.

```
constrMultTerm_χ(m):
  if m ∈_S χ then:
    return γ_m
  else:
    for all r in χ:
      if m divisible by r and m ÷ r is synthesizable from χ:
        return ⊙{|r, cf_χ(m ÷ r)|}
    for all factors f of m:
      if both f and m ÷ f are synthesizable from χ:
        return ⊙{|cf_χ(f), cf_χ(m ÷ f)|}
    // We arrive here if m cannot be synthesized.
    return undefined.
```

We now come to synthesizing exponential messages. Our main tool here are left and right reduction. The idea here is similar to the idea behind $constrMultTerm_\chi$. We again first check if $m \in_S \chi$, that is, if m is in χ directly. If this is not the case, we try to construct the message by synthesizing the direct sub-messages. If this is not possible, we resort to left and right reduction. This leads to the following algorithm:

```
constrExpTerm_χ(m):
  if m ∈_S χ then:
    return γ_m
  else if m_1 and m_2 are both synthesizable from χ:
    return cf_χ(m_1)^{cf_χ(m_2)}.
  else:
    for all r in χ:
      if m left reducible by r and m ◁ r is synthesizable from χ:
        return r^{cf_χ(m◁r)}
      else if m right reducible by r and m ▷ r is synthesizable from χ:
        return (cf_χ(m ▷ r))^r
    // We arrive here if m cannot be synthesized.
    return undefined.
```

The order in which the algorithms $constrMultTerm_\chi$ and $constrExpTerm_\chi$ iterate over the knowledge is not determined which means that the result of the algorithms is not deterministic. Anyhow, if there is a way of constructing a message, the algorithms find it. With them, we have now completed our definition of the constructive form and have hence a complete specification of how a message can be constructed by a principal. The constructive form is defined if and only if a message m can be constructed from knowledge χ:

Proposition 2.19. *Let m be a message in canonical form and let χ be a basic set with $synth^*(\chi) = close(\chi)$. Then $cf_\chi(m)$ is defined if and only if $m \in synth^*(\chi)$. If defined, then $cf_\chi(m)$ is composed of messages that are in χ and describes a valid way of constructing m from χ.*

The requirement that χ needs to be completely analyzed ($synth^*(\chi) = close(\chi)$) is important. Suppose, the knowledge χ contains the messages $\gamma_{\langle m_1\ .\ m_2\rangle}$ and m_3 but not the messages m_1 and m_2 (which are analyzable from $\gamma_{\langle m_1\ .\ m_2\rangle}$). In this case, $cf_\chi(\langle m_1\ .\ \langle m_2\ .\ m_3\rangle\rangle)$ would not be defined since the sub-message $\langle m_2\ .\ m_3\rangle$ cannot be constructed directly from χ.

2.6 Analyzing Messages

Analyzing messages is significantly simpler than synthesizing, we can basically just apply our analysis rules to the message that has been received. We still provide an explicit way of analyzing a message (compare to Definition 2.8 on page 16).

Definition 2.20. *Given a set of messages χ and a message m, the set $analyzeOnce_\chi(m)$ is defined as follows:*

- $analyzeOnce_\chi(m) = \{m\}$ *if m is atomic.*
- $analyzeOnce_\chi(\langle m_1\ .\ m_2\rangle) = analyzeOnce_\chi(m_1) \cup analyzeOnce_\chi(m_2)$.
- $analyzeOnce_\chi(\{m_1\}^{\mathbf{s}}_k) = \{\gamma_{\{m_1\}^{\mathbf{s}}_k}\} \cup analyzeOnce_\chi(m_1)$ *if* $k \in_M synth^*(\chi)$, *else* $analyzeOnce_\chi(\{m_1\}^{\mathbf{s}}_k) = \{\gamma_{\{m_1\}^{\mathbf{s}}_k}\}$.
- $analyzeOnce_\chi(\{m_1\}^{\mathbf{a}}_k) = \{\gamma_{\{m_1\}^{\mathbf{a}}_k}\} \cup analyzeOnce_\chi(m_1)$ *if* $k^{-1} \in_M synth^*(\chi)$, *else* $analyzeOnce_\chi(\{m_1\}^{\mathbf{a}}_k) = \{\gamma_{\{m_1\}^{\mathbf{a}}_k}\}$.
- $analyzeOnce_\chi(\{m_1\}^{\mathbf{a}}_{k^{-1}}) = \{\gamma_{\{m_1\}^{\mathbf{a}}_{k^{-1}}}\} \cup analyzeOnce_\chi(m_1)$ *if* $k \in_M synth^*(\chi)$, *else* $analyzeOnce_\chi(\{m_1\}^{\mathbf{a}}_{k^{-1}}) = \{\gamma_{\{m_1\}^{\mathbf{a}}_{k^{-1}}}\}$.
- $analyzeOnce_\chi(\mathbf{h}(m_1)) = \{\gamma_{\mathbf{h}(m_1)}\}$.
- $analyzeOnce_\chi(fun(m_1, \ldots m_i)) = \{\gamma_{fun(m_1,\ldots m_i)}\}$.
- $analyzeOnce_\chi(m_1^{m_2}) = \{\gamma_{m_1^{m_2}}\}$.
- $analyzeOnce_\chi(m_1 \odot m_2) = \{\gamma_{m_1 \odot m_2}\}$.

The set $analyzeOnce_\chi(m)$ contains all messages that can be extracted *in one step* from m given the knowledge χ with the help of the analysis rules (Asenc, Aaenc1, Aaenc2, Aconcat1 and Aconcat2). The extracted information is not used for further analysis (hence the function name). For example, $analyzeOnce_\varnothing(\langle\{m\}^{\mathbf{s}}_k\ .\ k\rangle) = \{\{m\}^{\mathbf{s}}_k, k\}$. With the message k, we could actually extract m by decrypting $\{m\}^{\mathbf{s}}_k$, but $analyzeOnce_\chi$ works with constant knowledge. In the translator, we apply $analyzeOnce_\chi$ repeatedly (each time adding the new information to the knowledge) until nothing new can be extracted.

Let us walk through the definition of $analyzeOnce_\chi$. The first rule tells us that we cannot analyze an atomic message. The only thing we can therefore learn from an atomic message is the message itself. The second rule is based on the analysis rules Aconcat1 and Aconcat2 that enable honest principals to decompose concatenation. The messages under the concatenation are then recursively analyzed.

The next three rules handle encryption; a principal always learns the message he analyzes. We can also extract the message under the encryption if we we can synthesize the decryption key. The case of symmetric encryption is based on Asenc (which is in turn related to rule (2) on page 11 of the message model). The rules for asymmetric encryption are based on the rules Aaenc1 and Aaenc2 (which are related to properties (3) and (4) of the message model). The messages that can be extracted are then recursively analyzed. In all other cases, we get no information except for the messages themselves.

Note that in the definition of $analyzeOnce_\chi$, there are cases where we have to check if a message is in $synth^*(\chi)$ which is in general infinite. Therefore, the $analyzeOnce_\chi$ function cannot be implemented on a computer in the form that is given above. However, thanks to proposition 2.19, we know that $m \in_M synth^*(\chi)$ whenever $cf_\chi(m)$ is defined and consequently we can replace $m \in_M synth^*(\chi)$ by a check if $cf_\chi(m)$ is defined.

Whenever a principal obtains a new message, he should analyze it and update his knowledge with the newly obtained messages. However, by doing so he may violate the basic set property of the knowledge if he does not take care.

Initial Knowledge:

$$A : \{m, \mathbf{pk}(B)\}$$
$$B : \{\mathbf{pk}(A)\}$$

Actions:

$$\begin{array}{lll}
(\mathbf{asw_1}) & A \to B \;\; (n_1) & : \{\mathbf{pk}(A) \,.\, \mathbf{pk}(B) \,.\, m \,.\, \mathbf{h}(n_1)\}^{\mathbf{a}}_{\mathbf{sk}(A)} \\
(\mathbf{asw_2}) & B \to A \;\; (n_2) & : \{\{\mathbf{pk}(A) \,.\, \mathbf{pk}(B) \,.\, m \,.\, \mathbf{h}(n_1)\}^{\mathbf{a}}_{\mathbf{sk}(A)} \,.\, \mathbf{h}(n_2)\}^{\mathbf{a}}_{\mathbf{sk}(B)} \\
(\mathbf{asw_3}) & A \to B & : n_1 \\
(\mathbf{asw_4}) & B \to A & : n_2
\end{array}$$

Figure 3: The ASW protocol.

The ASW protocol is perfectly suited for demonstrating how an honest principal can analyze an incoming message. It was proposed by Asokan, Shoup and Waidner [1] and has various applications in the context of authentication and non-repudiation such as contract signing or exchange of certified mail. A simplified version (the sub-protocols for resolving conflicts and aborting have been omitted) of it can be found in Figure 3.

Here, B can look under the encryption of the message he receives in message exchange step ($\mathbf{asw_1}$) since he knows the public key of A and consequently he learns $\mathbf{h}(n_1)$. Since B does not know n_1, he has to remember $\mathbf{h}(n_1)$ and therefore adds it to his knowledge. Later though, in message exchange step ($\mathbf{asw_3}$), he is sent n_1 and therefore he can construct $\mathbf{h}(n_1)$ – he should remove $\mathbf{h}(n_1)$ from his knowledge.

To simplify matters we allow principals to temporarily violate the basic set property of their knowledge. In the case above, B adds n_1 to his knowledge and then applies the following algorithm that converts an arbitrary set S of messages back into a basic set:

basicSet(S):
  while S contains a message m that is synthesizable from $S\backslash\{m\}$:
    remove m from S
  return S.

The following lemma claims that *basicSet* does indeed establish the basic set property and, more importantly, that the same messages can be synthesized from the basic set as from the original set.

Lemma 2.21. *Let S be a set of messages. Then $\chi := basicSet(S)$ is a basic set and $synth^*(\chi) = synth^*(S)$.*

Lemma 2.21 implies that there is a basic set S' for every set S with $synth^*(S) =_M synth^*(S')$. From Proposition 2.7, we know that this basic set is also unique. Based on these results, we can now formulate the following useful proposition:

Proposition 2.22. *Let Q and R be two sets of messages. Then there are unique basic sets Q' and R' with $synth^*(Q) =_M synth^*(Q')$ and $synth^*(R) =_M synth^*(R')$. For these, we have $synth^*(Q) =_M synth^*(R)$ if and only if $Q' =_M R'$.*

Recall Lemma 2.12 that states that the canonical form of a message exists and is unique. If we combine this with Proposition 2.22, we get a very convenient way of checking for two sets of messages Q and R whether $synth^*(Q) =_M synth^*(R)$. The check works as follows: First, we construct the basic set of Q as $Q' := basicSet(Q)$ and the basic set of R as $R' := basicSet(R)$. We have to make sure that all messages in Q' and R' are in canonical form (we can use the function *canonical* for this purpose). Then we can check if $synth^*(Q) =_M synth^*(R)$ simply by checking $Q' =_S R'$.

When a principal receives a new message from the network he can analyze it under his current knowledge by using the function $analyzeOnce_\chi$. We have already demonstrated that computing $analyzeOnce_\chi$ once is not always sufficient. The example we gave was $analyzeOnce_\varnothing(\langle\{m\}^{\mathbf{s}}_k \,.\, k\rangle) =$

$\{\{m\}^{\mathbf{s}}_k, k\}$. We can see that not all information is extracted since, with the message k, we could extract m from $\{m\}^{\mathbf{s}}_k$. If the principal runs $analyzeOnce_\chi$ a second time, now with the newly obtained knowledge, he can extract all information: $analyzeOnce_{\{\{m\}^{\mathbf{s}}_k,k\}}(\langle\{m\}^{\mathbf{s}}_k \,.\, k\rangle) = \{\{m\}^{\mathbf{s}}_k, k, m\}$. However, it is not sufficient to analyze only the newly received messages. Suppose a principal knows $\{m\}^{\mathbf{s}}_k$ but only learns k at a later time; then $\{m\}^{\mathbf{s}}_k$, an "old" message, becomes synthesizable.

We can see that whenever we receive a new message, we need to analyze *all* messages in the knowledge. Therefore, we treat a newly received message m as follows. Firstly, we add m to the knowledge (note that this may violate the basic set property). Then we check if there is a message from which new messages can be extracted (we can use $analyzeOnce_\chi$ for this). If this is the case, then we add them to our knowledge. This process is repeated until no more new knowledge can be analyzed. Finally, we restore the basic set property. Let us give this algorithm in explicit form:

$addKnowledge_\chi(m)$
  let $\chi := \chi \cup \{m\}$
  while there is $m' \in_S \chi$ s.t. $analyzeOnce_\chi(m') \not\subseteq_M \chi$
    let $\chi := \chi \cup analyzeOnce_\chi(m')$.
  return $basicSet(\chi)$ // Removes synthesizable messages

The following proposition expresses that the knowledge produced by $analyzeKnowledge$ is correct:

Proposition 2.23. *Let m be a message and χ a basic set. Then $addKnowledge_\chi(m)$ is a basic set and $synth^*(addKnowledge_\chi(m)) =_M close(\chi \cup \{m\})$.*

2.7 Checks

Explicitly stating which checks a principal can execute is an essential part of the semantics of Alice&Bob protocol specifications. A principal that does not perform a check as foreseen by the designer of the protocol might open a door for attacks. Which checks are possible was investigated by Caleiro et al. [3]. They give three different types of checks; we will now illustrate the checks that are possible on examples. However, we will not make this distinction between three types of checks. We also do not give a formal specification of checks since we never need them explicitly in this thesis. We can rely completely on the internal proving theory of Tamarin in which fact rewriting rules can only be applied if they meet certain preconditions; this has the same effect as principals that perform checks. Therefore, we refer to the work of Caleiro and his colleagues for details and a formal specification.

A check should ensure that the knowledge of a principal is consistent, as far as this is possible. Suppose a principal has the knowledge $\{a, \gamma_{g^a}\}$. We can see that the principal does not know g and consequently there is no way for him to ensure that g^a and a are actually consistent. The message γ_{g^a} could be a dummy value generated by the adversary. This demonstrates that it is possible that inconsistencies in the knowledge cannot be detected since a principal lacks information. When the message g is now received, g^a becomes synthesizable. In this case, the principal should construct g^a from the a from the knowledge and the newly received g and ensure that the value agrees with γ_{g^a}; if this is not the case, he should abort protocol execution. Note that γ_{g^a} is removed from the knowledge since it has become synthesizable; this shows that checks and analysis are intertwined and should be performed together.

It is also possible that a newly received message renders another message in the knowledge analyzable. If a principal has the knowledge $\{m, \{m\}^{\mathbf{a}}_{\mathbf{pk}(A)}\}$ and then receives $\mathbf{sk}(A)$, he still cannot synthesize $\{m\}^{\mathbf{a}}_{\mathbf{pk}(A)}$; but now, he can ensure that m and $\{m\}^{\mathbf{a}}_{\mathbf{pk}(A)}$ agree since $\{m\}^{\mathbf{a}}_{\mathbf{pk}(A)}$ can now be decrypted and the two m compared.

Let us now illustrate some of the checks that are possible on the ASW protocol that we gave earlier (Figure 3 on the preceding page). In message exchange step ($\mathbf{asw_1}$), B receives a message that is encrypted with the secret key of A. Since B possesses the corresponding public key for decryption, he can extract the message under the encryption and consequently can extract $\mathbf{pk}(A)$, $\mathbf{pk}(B)$, m and $\mathbf{h}(n_1)$. B already possesses $\mathbf{pk}(A)$ and $\mathbf{pk}(B)$ and can therefore check if the received values match with his own view of these messages. Later, in message exchange step ($\mathbf{asw_3}$), B receives the message n_1. The message $\gamma_{\mathbf{h}(n_1)}$ which he has in his knowledge is synthesizable from n_1. Therefore, he should now check if $\gamma_{\mathbf{h}(n_1)}$ agrees with the n_1 that has just been received.

2.8 Wrap-up

We are now at a point where we have developed all the tools that we need to formalize Alice&Bob protocol specifications. Our goal is to give a description of the exact actions that a principal that executes a specific role has to perform.

2.8.1 Actions, Roles and Protocols

Alice&Bob protocol specifications describe the messages that are exchanged during a successful run of the protocol with honest principals; in each communication step, we have somebody who sends and somebody who receives. In the rest of this section, we specify the actions that are executed by both the sender and the receiver, and we do this by defining the *role* that has to be executed by a principal.

Let us recall what a message exchange step looks like in Alice&Bob notation:

$$S \to R \ (n_1, \ldots, n_v) \ : \ M.$$

The role of a principal that executes some role P (S or R) can be described by giving an initial state (that is, the initial knowledge) and a sequence of actions $act_1, act_2, \ldots act_n$ that need to be performed by the principal. We denote the knowledge that a principal has after executing action act_i by χ_i, where $\chi_0 := \chi_P^{init}$ denotes the initial knowledge. What we do now is to introduce the actions that define what a principal has to do in a protocol execution step. Afterwards, we define the roles which are based on our notions of actions and knowledge. Finally, we can define what a protocol is. Let us make a case distinction based on whether a principal is the sender or the receiver in a message exchange step:

- If a principal is the sender S, then he first has to generate the fresh values $n_1, \ldots, n_v$. For each variable n_i, a value is chosen (pseudo-)randomly from Num. The new variable n is then immediately added to the knowledge. The basic set property is not violated since n is atomic and, being fresh, it cannot appear in other messages. The old knowledge χ_{i-1} is updated in one step by adding all new knowledge at once and producing the new knowledge χ_i.

  After generating the fresh values, the principal builds the constructive form $cf_{\chi_i}(m)$ based on the knowledge with the fresh values χ_i. If $cf_{\chi_i}(m)$ is undefined then the sending action cannot be executed. Otherwise, the principal constructs the message from his knowledge and sends it to the receiving principal R via the network. The sending itself does not influence the knowledge.

- If a principal is the receiver R, then he first receives the message m from the network, apparently from sender S. He then first performs the checks as explained above on this message to ensure that it has the expected shape. The principal aborts the execution of his run of the protocol if any check fails.

  Finally, the knowledge is updated. This is done by calculating $addKnowledge_\chi$ on the old knowledge χ_{i-1} which then gives us the new knowledge χ_i.

Caleiro et al. [3] gives an operational semantics for Alice&Bob notation based on the spi calculus. We are not as formal here, even though we use a similar idea by defining a role as a sequence of actions. There is an action type for sending and another one for receiving along the lines of the explanation above:

Definition 2.24. *An* action *is one of the following:*

- **Send**(F, R, m). *Here, F is the set of fresh variables, R is the name of the receiver and m is the message that needs to be sent. The following actions are performed in the given order:*

  1. *Randomly choose values from Num for the fresh variables in the set F. Determine the new knowledge as $\chi_i := \chi_{i-1} \cup F$.*
  2. *If m is synthesizable from χ_i, then synthesize the message according to $cf_{\chi_i}(m)$ from χ_i and send it to R via the network. Otherwise,* **Send**(F, R, m) *is undefined.*

- **Recv**(S, m). *Here, S denotes the sender by whom the message was apparently sent. The expected message is denoted by m. The following actions are performed in the given order:*

  1. *Receive the message m from the network.*

2. *Execute the checks that are possible. If any check fails, abort the execution of the role.*
3. *Set the new knowledge to* $\chi_i := addKnowledge_{\chi_{i-1}}(m)$.

The definition of a role is now straightforward. A role is defined by the initial knowledge that a principal has and a sequence of the actions that need to be performed by that principal.

Definition 2.25. *The* role *of* P *is a pair* (χ^{init}, Σ)*, denoted by* $role_P$*, where*

- χ^{init} *denotes the knowledge that a principal executing this role has before any actions are executed, i.e.,* $\chi^{init} := \chi_P^{init}$.
- Σ *is a finite sequence* $act_1, \ldots, act_n$ *of actions.*

Finally, a protocol is nothing but the set of all the roles that appear in a protocol. Hence, a protocol can be defined by the following very simple definition.

Definition 2.26. *A set of roles is called a* protocol.

This definition allows for protocols that make absolutely no sense at all. The most important property of a sensible protocol is that the honest principals can execute their actions as required. While a principal can always receive a message, it may not be possible to synthesize a message because he lacks the necessary knowledge. The A&B language does not accept any protocols that cannot be executed even if there was no involvement of the adversary. If we go back to our definition of the **Send**-action, we can see that it is not defined whenever the message to be sent cannot be constructed. This leads to the following definition:

Definition 2.27. *A role is called* executable *if all of its actions* $act_1, \ldots, act_n$ *are defined. A protocol is called* executable *if all roles are executable.*

Note that, even though the checks that a principal performs when receiving a message can lead to the abortion of protocol execution, checks are completely independent of the concept of executability. No protocol run can be finished if the adversary controls the network and blocks communication. Executability only requires that all actions can be performed by principals *if* there is no involvement of the adversary.

This concludes our investigation of the semantics of Alice&Bob protocol specifications. We can now turn our attention towards the A&B protocol specification language. But before we do that we would like to give a final example that summarizes the results of this section.

2.8.2 Example

We have started this section with the CR protocol (Figure 1 on page 9) on which we have pointed out that Alice&Bob notation comes with certain ambiguities and have motivated why it is necessary to explicitly state initial knowledge and fresh numbers. Now that we have specified how we write protocols in Alice&Bob notation, let us give an improved version of the CR protocol specification. It can be found in Figure 4.

Initial Knowledge:

$$C : \{\mathbf{pk}(R)\}$$

Actions:

$$\begin{array}{lll} (\mathbf{cr_1}) & C \to R \ \ (n) & : \ \{n\}^{\mathbf{a}}_{\mathbf{pk}(R)} \\ (\mathbf{cr_2}) & R \to C & : \ \mathbf{h}(n) \end{array}$$

Figure 4: The CR protocol that was already specified in Figure 1 on page 9. Here, we obey our notational conventions and explicitly specify fresh values and initial knowledge to avoid the ambiguities that arose from the initial specification.

The CR protocol specification is now unambiguous and therefore we can extract the roles of C and R that state the initial knowledge and the actions taken by the principals:

$$role_C := (\{\mathbf{pk}(R), \mathbf{sk}(C), \mathbf{pk}(C), C\}, \langle \mathbf{Send}(\{n\}, R, \{n\}^{\mathbf{a}}_{\mathbf{pk}(R)}), \mathbf{Recv}(R, \mathbf{h}(n))\rangle)$$
$$role_R := (\{\mathbf{sk}(R), \mathbf{pk}(R), R\}, \langle \mathbf{Recv}(C, \{n\}^{\mathbf{a}}_{\mathbf{pk}(R)}), \mathbf{Send}(\varnothing, C, \mathbf{h}(n))\rangle)$$

This is the complete specification of the CR protocol, the actions have to be executed as declared above. Below, we give a detailed list of the actions that should be performed by both principals:

Execution of the Role of C.

- Initial knowledge: $\chi_0 := \{\mathbf{sk}(C), \mathbf{pk}(C), C, \mathbf{pk}(R)\}$. Here, $\{\mathbf{sk}(C), \mathbf{pk}(C), C\}$ is the implicit knowledge. The message $\mathbf{pk}(R)$ was explicitly stated in the protocol specification and provides C with an authenticated copy of R's public key.
- Execution of action $\mathbf{Send}(\{n\}, R, \{n\}^{\mathbf{a}}_{\mathbf{pk}(R)})$:
  - C (pseudo-)randomly chooses a fresh value for n from Num and adds it to her knowledge. The updated knowledge then contains n: $\chi_1 := \{\mathbf{sk}(C), \mathbf{pk}(C), C, \mathbf{pk}(R), n\}$.
  - The constructive form $cf_{\chi_1}(\{n\}^{\mathbf{a}}_{\mathbf{pk}(R)}) := \{n\}^{\mathbf{a}}_{\mathbf{pk}(R)}$ is defined and therefore, the message can be constructed and sent to R via the network.
- Execution of action $\mathbf{Recv}(R, \mathbf{h}(n))$:
  - Checks: $\mathbf{h}(n)$ is constructible from χ_1 and therefore C can check if the received message matches the $\mathbf{h}(n)$ constructed from χ_1. If there is a mismatch, she immediately aborts the run of the protocol. There are not further checks.
  - C analyzes the message: $\chi_2 := addKnowledge_{\chi_1}(h(n)) = \chi_1$. No new messages can be analyzed from $\mathbf{h}(n)$; $\mathbf{h}(n)$ does not need to be stored since it can be constructed from n.

Execution of the Role of R.

- Initial knowledge: $\chi_0 := \{\mathbf{sk}(R), \mathbf{pk}(R), R\}$, that is, R possesses only his implicit knowledge since no additional knowledge is stated.
- Execution of action $\mathbf{Recv}(C, \{n\}^{\mathbf{a}}_{\mathbf{pk}(R)})$:
  - R possesses $\mathbf{sk}(R)$ and can therefore analyze the message and extract n from $\{n\}^{\mathbf{a}}_{\mathbf{pk}(R)}$.
  - No checks are possible.
  - There are no checks that can fail and therefore, the extracted n is always added to the knowledge. The new knowledge is $\chi_1 := addKnowledge_{\chi_0}(\{n\}^{\mathbf{a}}_{\mathbf{pk}(R)}) = \{\mathbf{sk}(R), \mathbf{pk}(R), R, n\}$.
- Execution of action $\mathbf{Send}(\varnothing, C, \mathbf{h}(n))$.
  - No fresh values are generated, that is, $\chi_2 := \chi_1$.
  - The constructive form $cf_{\chi_2}(\mathbf{h}(n)) = \mathbf{h}(n)$ is defined and therefore, the message can be constructed and sent to C via the network.

We can read from the explanations above that R cannot perform any checks at all. It is therefore clear that R has no authentication guarantees with respect to C since R would accept absolutely any message in action $\mathbf{Recv}(C, \{n\}^{\mathbf{a}}_{\mathbf{pk}(R)})$, even if it is not a valid encryption of a message.

3 The A&B Input Language

We have used the protocols declared in Figures 2, 3, and 4 to illustrate some aspects of Alice&Bob protocols in Section 2. The corresponding protocols in the A&B protocol specification language can be found in Figure 5 (with some changes). All of the A&B protocols that are used in this thesis can be downloaded from the web page of Tamarin [10].

The semantics of the A&B input language closely follows the semantics and notational conventions of the Alice&Bob notation introduced in Section 2. We use the same message model and the same overall structure. Therefore, this section is mostly about defining a slightly different, computer-interpretable notation for the Alice&Bob protocol specifications introduced above.

The overall structure of A&B protocol specifications is inspired by a similar input format that is used by Mödersheim [12]. A protocol specification starts with the keyword **`Protocol`** followed by an identifier that names the protocol. After that, the actual protocol specification is given. It is closed with the **`end`** keyword.

We proceed in a similar fashion as in the previous section. First, we talk about the representation of messages in A&B because they lie at the heart of protocol specifications and then move on to talking about knowledge and actions. As the examples show nicely, an A&B protocol specification consists of several *blocks*. The CR protocol, for instance, contains a **`Knowledge`** block in which the initial knowledge of the roles is specified, an **`Actions`** block in which the message exchange steps are given, and, finally, a **`Goals`** block.

In the **`Goals`** block, the security properties that a protocol should guarantee can be declared. Such information is usually not included in Alice&Bob notation (as in our definition of Alice&Bob protocol specifications). However, it makes sense to be able to at least denote some basic security goals such as secrecy and (non-)injective agreement in an input language that is translated to the input language of protocol verifiers. In the last part of this section, we talk about these security goals.

For reference, the specification of the grammar of the A&B input language is attached to this thesis. It can be found in Appendix B.

3.1 Basics

Before we move on to messages, let us briefly discuss some basics of the A&B protocol specification language.

Identifiers. Identifiers in A&B start with a letter, followed by any number of letters, digits and underscores ('_'). Additionally, an identifier must not coincide with any keyword. A list of the reserved keywords can be found in Appendix B.

Case Sensitivity. A&B as such is not case sensitive. For example, the keyword **`Protocol`** that starts a protocol could also be written as **`PROTOCOL`** and hence, neither of the two is a valid identifier. However, since our intermediate representation should be independent of the target language (that may be case sensitive), we require that there appear no two identifiers in a protocol that are equal in a case insensitive way but not equal in a case sensitive way. For example, the identifiers `message` and `Message` may not appear both in the same protocol.

Blocks. We have already seen that an A&B protocol specification is composed of blocks. There are four types of blocks that may be used, namely **`Declarations`**, **`Knowledge`**, **`Actions`** and **`Goals`**. Any of these blocks may be left off if not needed (although protocols without an **`Actions`** block are not very useful). However, no block must occur twice and they must appear in the same order as in the list above.

3.2 Messages

In A&B we use the same messages and the same message model as in the Alice&Bob protocol specification defined in Section 2.2. A&B uses the following syntax for messages:

- *Variables:* A variable is denoted by an identifier such as `msg`. Variables may denote both numbers and agent names (see the discussion below).

```
Protocol CR:
Knowledge:
    C: pk(R);
Actions:
    [cr1] C -> R (n) : aenc{n}pk(R);
    [cr2] R -> C     : h(n);
Goals:
    [n_secret]   n secret of C, R;
    [authNonInj] C non-injectively agrees with R on n;
    [authInj]    C injectively agrees with R on n;
end
```

```
Protocol DIFFIE_HELLMAN:
Declarations:
    g/0;
Actions:
    [dh_1] A -> B (x) : g()^x;
    [dh_2] B -> A (y) : g()^y;
    [dh_3] A -> B (n) : senc{n}(g()^(x*y));
Goals:
    /* Goal that holds */
    [key_secret] g()^(x*y) secret of A, B;
    /* Goals that do not hold */
    [key_secretA] g()^(x*y) secret of A;
    [key_secretB] g()^(x*y) secret of B;
end
```

```
Protocol ASW:
Knowledge:
    A : m, pk(B), B;
    B : pk(A), A;
Actions:
    [asw1] A -> B (n_1) :
        aenc{ pk(A) . pk(B) . m . h(n_1) }sk(A);
    [asw2] B -> A (n_2) :
        aenc{ aenc{pk(A) . pk(B) . m . h(n_1)}sk(A) . h(n_2) }sk(B);
    [asw3] A -> B :
        n_1;
    [asw4] B -> A :
        n_2;
Goals:
    /* Goals that hold */
    [weakA] A non-injectively agrees with B on m;
    [weakB] B non-injectively agrees with A on m;
    [strongA] A injectively agrees with B on m;
    /* Goals that do not hold */
    [strongB] B injectively agrees with A on m;
    [secrecy] m secret of A, B;
end
```

Figure 5: The three protocols from Section 2 expressed in the A&B input language with security goals.

- *Constants:* Constants are identifiers in single quotes. For example, `'I_am_a_constant'` represents a legal constant number.
- *Concatenation:* The concatenation of messages `m1` and `m2` is denoted by `< m1 . m2 >` or simply `m1 . m2`.
- *Symmetric Encryption:* The symmetric encryption of a message `m1` with the message `m2` is denoted by **`senc`**`{m1}m2`.
- *Asymmetric Encryption:* The asymmetric encryption of a message `m1` with the public key of principal `A` is denoted by **`aenc`**`{m1}`**`pk`**`(A)`. The asymmetric encryption of a message `m1` with the secret key of a principal `A` is denoted by **`aenc`**`{m1}`**`sk`**`(A)`.
- *Hashing:* The hash of message `m` is denoted by **`h`**`(m)`.
- *Function Application:* Let `fun` be an i-ary function (other than in Alice&Bob notation we take the arity of functions into account in A&B; we elaborate on this later) and let `m1`, ..., `mi` be messages. Then the application of `fun` to `m1`, ..., `mi` is denoted by `fun(m1, ..., mi)`.
- *Multiplication:* Let `m1` and `m2` be messages. Then the multiplication of `m1` with `m2` is denoted by `m1 * m2`.
- *Exponentiation:* Let `m1` and `m2` be messages. Then the exponentiation of `m1` with `m2` is denoted by `m1 ^ m2`.

Note that inverse messages cannot be expressed. We have pointed out earlier that honest principals cannot construct inverse messages and that the only way for them to get hold of an inverse message is by either having them in their initial knowledge of by receiving them over the network. For this reason, we have decided not to include explicit inversion of messages in A&B. All participants have their public and secret key (which are mutually inverse) in their implicit knowledge and can therefore still sign or asymmetrically encrypt messages. We also have not included the message 1 in the A&B input language since it is intended for internal calculations only.

3.2.1 Operator Precedence

The only binary operators are the concatenation, the multiplication and the exponentiation operators. They have the following precedence in A&B where the operator precedence increases from left to right:

$$. \quad \rightarrow \quad * \quad \rightarrow \quad \hat{}$$

Let us illustrate this with `a . b ^ c * d ^ e . f`. This message in A&B represents the message $\langle a \,.\, (b^c \odot d^e) \,.\, f \rangle$.

3.2.2 Associativity of Operators

All binary operators (concatenation, multiplication, and exponentiation) are right-associative. This is of no actual importance for concatenation and multiplication since they are associative. In the case of exponentiation, however, this is decisive for meaning. `a ^ b ^ c`, for instance, is interpreted as $a^{(b^c)}$.

3.2.3 Influencing Operator Precedence and Associativity

Associativity and precedence can be overruled by using parentheses such as in `(a ^ b) ^ c`. This represents the message $(a^b)^c$ ($a^{b \odot c}$ in canonical representation). Concatenation comes with a special grouping notation. While it is possible to use parentheses like in `(a . b) ^ (a . c)`, the recommended notation is the angle notation that looks like `< a . b > ^ < a . c >`. This improves readability, especially if there are many parentheses (and angles).

3.3 Specifying a Protocol

We now come to the actual protocol specifications in A&B. Before the message exchange steps can be specified, one needs to declare the functions that are used in the protocol and provide the explicit knowledge of the roles.

3.3.1 Declaring Functions

In A&B, functions have to be stated explicitly before they can be used in a protocol. This is done in the first block, **`Declarations`**. Let us demonstrate this with a small example of such a block:

```
Declarations:
    foo/2;
    public bar/0;
    private baz/1;
```

The example declares three functions, `foo`, `bar` and `baz`. The numbers after the slashes specify the arity; while `bar` takes no parameters, `baz` takes exactly one and `foo` two.

A possible application of `foo` might look like `foo(m1, m2 . m3)`. Note that the comma operator separating the arguments has lowest precedence, that is, there is no need to write `< m2 . m3 >` in angles. Since `bar` accepts no arguments, it is written with empty parentheses like `bar()` when used in a message. The parentheses must never be omitted.

We claimed earlier that Alice&Bob notation is independent of the adversary model; however, when it comes to functions, we make a little exception and allow the specifier of the protocol to declare functions **`private`**. Only honest principals have the ability to apply such functions. By default, functions are public even though it is possible to explicitly mark them **`public`**. In the example given above, `foo` and `bar` are both public while `baz` is private. We call the **`private`** and **`public`** keywords *access modifiers*.

We would like to stress here that the access modifiers have no effect whatsoever on the capabilities of honest principals – they can always apply all functions like we have defined above in Section 2. Let us illustrate this with the following two message exchange steps (in Alice&Bob notation):

$$\begin{aligned} C \to R \ \ (n_1) &: n_1 \\ R \to C \qquad &: fun(n_1) \end{aligned}$$

If function fun is declared public, then the adversary can intercept n_1 from C and then construct $fun(n_1)$ and send it back to C. R would not even realize that C has sent a message and for C, there is no possibility to find out if he is talking to the adversary or to R. Conversely, if fun is declared private, then the adversary cannot construct $fun(n_1)$ and C can be sure that the returned value actually comes from R. It is important to note here that C can check if the message $fun(n_1)$ has the correct shape since she, being an honest principal, can apply fun.

3.3.2 Declaring Initial Knowledge

A&B uses the same semantics as Alice&Bob notation defined in Section 2 when it comes to the initial knowledge that a principal in some role possesses. A principal implicitly knows his own name and his own public and private keys; in case he should have more knowledge we can state this explicitly in the **`Knowledge`** block. If we take a glance at the ASW example in Figure 5, we can see that a principal executing role A knows the message m and the name and public key of the principal in role B, in addition to the implicit knowledge. While B knows A's public key and name he does not know m initially. Summarizing, A and B have the following initial knowledge:

$$\begin{aligned} \chi_A^{init} &= \{A, \mathbf{sk}(A), \mathbf{pk}(A), B, \mathbf{pk}(B), m\}, \text{ and} \\ \chi_B^{init} &= \{B, \mathbf{sk}(B), \mathbf{sk}(B), A, \mathbf{pk}(A)\}. \end{aligned}$$

Properties of Initial Knowledge. It is important to mention that there is an inherent difference between implicit ($\{P, \mathbf{sk}(P), \mathbf{pk}(P)\}$) and explicitly stated knowledge. While the implicit knowledge is bound directly to a principal, the explicitly stated knowledge is bound to a specific run of a protocol.

In the case of the ASW protocol, A originally knows m. It is a fresh number in every run of the protocol, and therefore, even if the adversary could obtain m in one run of the protocol he could not reuse it later.

In the case of the ASW protocol, it would actually not make any difference if we declared m as a fresh variable together with n_1 in action ($\mathbf{asw_1}$). We have chosen to write it in the initial knowledge block to emphasize that m is not just a protocol variable but that it is the value that should be exchanged between A and B.

A more important aspect of the initial knowledge is that it allows us to specify that principals share some knowledge before the start of protocol execution, for example a password. If we added m to the initial knowledge of B, A and B would be guaranteed to have the same view of m right from the start. We can express this as follows:

The values of all atomic messages that appear in the initial knowledge of a principal in a protocol specification are guaranteed to be equal for all participants. Moreover, all numbers in the explicit initial knowledge are guaranteed to be unique for every run of the protocol.

Let us give one last example:

```
Knowledge:
  A : senc{m1}m2;
  B : m2;
```

Here, `m2` is guaranteed to have equal value for both A and B.

Initial Knowledge, Nullary Functions and Constant Numbers. The A&B language uses strings in single quotes like `'constant'` as its syntax for constant numbers. Nullary functions take no arguments and are therefore in some sense also constants. Still, there is one little difference, namely, that functions can be declared private. While constants in A&B are assumed to be generally known (also to the adversary), private nullary functions are only known to honest principals (however, all of them; this is not a problem tough since they are honest). If we do not want them to know a message, we have to declare it as a variable in the initial knowledge (but then it is not a constant).

It is important to be aware of these details because they can have a security-relevant influence on the meaning of a protocol. For example, assume that we want to model a password p that is shared by two principals and used many times.

The fact that the password is used multiple times could tempt us to declare it as a constant. However, this is clearly wrong since constants are public. Declaring the password as a variable in the initial knowledge would be even worse. Such variables are instantiated with a fresh value in every protocol execution and consequently, a replay attack might not be discovered! The only appropriate way of representing p is as a private nullary function.

3.3.3 Declaring Message Exchange Steps

A message exchange step in A&B has exactly the same shape as in Alice&Bob notation. In general, it has the following shape:

```
[label] A -> B (n_1, ..., n_v) : message;
```

or:

```
[label] B <- A (n_1, ..., n_v) : message;
```

The label has to be a valid, unique identifier and cannot be omitted. By unique we mean that it must not collide with *any* other identifier in the protocol, for example a variable or function name. This label may be used in the output produced by the compiler, so choosing a sensible name can make the output more readable.

Fresh variables are specified in parentheses that are located between receiver and the colon. They may be left out completely if there are no fresh variables like in steps `[asw3]` and `[asw4]` of the ASW protocol in Figure 5 on page 30.

3.4 Declaring Security Goals

The Alice&Bob notation from Section 2 does not provide the possibility to specify the security goals one wants to achieve with a protocol. A&B, however, is designed to be an input language for protocol verifiers and therefore it is sensible if one can state at least some of the most common security goals. For this reason, we have decided to include the **`Goals`** block in A&B where three types of goals can be specified, namely, secrecy, injective agreement and non-injective agreement. Let us start by demonstrating the three types of secrecy goals on the CR example.

3.4.1 Overview

The specification of the CR protocol in A&B, including security goals, can be found in Figure 5 on page 30. In it, C generates the nonce n, asymmetrically encrypts it with the public key of R and sends it to R. Since the adversary is assumed not to know the secret key of R, he cannot decrypt the message and obtain n. Therefore, if C receives the hash of n back from the network he can be sure that it was constructed by R.

Under the usual Dolev-Yao adversary model, there is no way for the adversary to obtain n since he can neither decrypt $\{n\}^{\mathbf{a}}_{\mathbf{pk}(R)}$ nor extract n from $\mathbf{h}(n)$. One of the objectives of the CR protocol is just this, namely that n must not be known by the adversary. This is modelled in the first security goal, `[n_secret]`.

The other major objective of the CR protocol is to ensure that if C terminates a run of the protocol, she can be sure that R knows the nonce produced by her. This is modelled by the second and the third security goal, `[authNonInj]` and `[authInj]`. `[authNonInj]` specifies that C should (non-injectively) agree with R on n. The meaning of this is that if C finishes execution of her protocol, then she is guaranteed that R has received n and that both R and C have a consistent view of n (that is, they have the same value for it).

Goal `[authInj]` requires injective agreement, a stronger version of non-injective agreement. Additionally to `[authNonInj]`, this security goal demands not only that C and R have the same view of the message, but also that every run of the protocol is unique. This rules out replay attacks.

3.4.2 Secrecy

The first type of security goal that we discuss are secrecy goals. A&B uses the following syntax:

```
[label] msg secret of P_1, ..., P_n;
```

Security goals always start with a label. Like in the case of communication exchange steps, this label must be a unique and valid identifier and cannot be omitted. The actual security goal specification starts with the message that we require to be secret followed by the two keywords "**`secret of`**". Finally, a comma-separated list of one or more role names is given. We define secrecy as follows:

Definition 3.1. *Let p be a protocol that contains the roles $P_1, \ldots, P_n$. A message msg is called* a secret of the roles $P_1, \ldots, P_n$*, if whenever there is a run of the protocol in which the honest principals in the roles $P_1, \ldots, P_n$ successfully finish the execution of their protocol run, and have the same view of msg, then the adversary does not possess that value of msg at any time.*

Let us take a closer look at this definition. The most important property of our definition of secrecy that we should notice is that the message only needs to be secret if all of the principals that participate in a run of the protocol actually finish execution and agree on the same value. Secrecy goals therefore are meaningless if any of the principals $P_1, \ldots, P_n$ cannot construct *msg*.

To point out some aspects that require attention, let us return to the Diffie-Hellman key exchange protocol that we introduced earlier when we were talking about Alice&Bob notation (see Figure 2). A similar version in the A&B protocol specification language can be found in Figure 5.

In this protocol, we declare one secrecy goal, namely that the common key $g^{x \odot y}$ is a secret of A and B. If we assume an adversary model where the adversary cannot solve the discrete logarithm problem, this protocol actually satisfies this security goal, since $g^{x \odot y}$ has to be secret only if both A and B are honest principals and both finish protocol execution and agree on the same value.

Note that in this particular protocol, there is no common secret knowledge that honest participants share and consequently there is no way for principals to authenticate one another; the adversary can easily impersonate either of the roles and there is no way for an honest principal to detect this. This means that our definition of secrecy is in some sense orthogonal to authentication.

Consider the goals `key_secretA` and `key_secretB`. Even though they look very similar to goal `key_secret`, neither of them is satisfied. We require the principals in the roles that are mentioned in a security goal to be honest and since only one of the two roles is mentioned, the adversary can take over the other role, in which case the adversary obtains the shared key.

3.4.3 Agreement

The syntax of injective and non-injective agreement security goals is as follows:

```
[label1] P non-injectively agrees with Q on msg_1, ..., msg_n;
[label2] P injectively agrees with Q on msg_1, ..., msg_n;
```

Again, the security goal starts with a label that should be a valid and unique identifier. The two principals that are mentioned, `P` and `Q`, have to be distinct. The list of messages `msg_1`, ..., `msg_n` must contain at least one message, and all of the messages should be synthesizable by both P and Q at the latest at the end of their roles, since otherwise, authentication goals are meaningless.

Our specification of agreement is based on the definition proposed by Lowe [6]. Concretely, we define non-injective agreement as follows:

Definition 3.2. *Let P and Q be two roles of a protocol p and let $m_1, \ldots, m_n$ be messages. We say that P* non-injectively agrees with *Q on the messages $m_1, \ldots, m_n$ if and only if whenever (an honest principal in role) P successfully finishes a run of the protocol, then (an honest principal in role) Q has previously been running the protocol, apparently with P, and both P and Q agree on $m_1, \ldots, m_n$.*

The CR protocol in Figure 5 requests that C non-injectively agrees with R on the nonce n. Let us briefly discuss this. C sends n to R, asymmetrically encrypted with R's public key. The only principal that knows $\mathbf{sk}(R)$ is R and hence, assuming a Dolev-Yao adversary model, only R can decrypt the message and obtain n. If C receives the hash of n, he can thus be sure that R has received n. Consequently, C only successfully finishes his run of the protocol if R has received n and has the same view of it; the CR protocol achieves the goal `[authNonInj]`.

The nonce n is freshly generated in every run of the protocol which rules out replay attacks. In such a case, we speak of *injective agreement*. This is required by the CR protocol in goal `[authInj]` and is defined as follows:

Definition 3.3. *Let P and Q be two roles of a protocol p and let $m_1, \ldots, m_n$ be messages. We say that P* injectively agrees with *Q on the messages $m_1, \ldots, m_n$ if and only if Q agrees non-injectively with P on $m_1, \ldots, m_n$, and each such run of P corresponds to a unique run of Q.*

3.5 Well-formedness Checks

Even if a protocol meets the requirements of the grammar of the A&B input language and can therefore be parsed, this does not mean that the specification is valid. Typical problems are functions that are used in messages but have not been declared or name clashes. We call a protocol that can be parsed but is still not valid *not well formed*.

```
Protocol secrecy:
Declarations:
    public fun/-1;
    private fun/2;
Knowledge:
    A : secr, pk(A), m;
Actions:
    [action] A -> A (secr) : aenc{secr}pk(A);
    [action] A <- A        : fun(secr);
Goals:
    [secrecy] bar(secr) secret of B;
end
```

Figure 6: A parsable A&B specification that is not well-formed.

An example of a protocol that is not well formed can be found in Figure 6. We now provide a list with all the requirements that a well-formed protocol should fulfill.

- A principal must not send a message to himself or herself. In the example protocol, A does just this (`A -> A`).

- No two functions must have the same name, even if they have different arity. In the example, **public** `fun/-1` and **private** `fun/2` collide.
- Every function that is used in a message must be declared. In the example, the function `bar` is used in the **Goals** block but is not defined in the **Declarations** block.
- No label must occur twice. In the example, two message exchange steps are defined with the label `[action]`. Also, the name given to the protocol and the name of the secrecy goal collide (label `[secrecy]`).
- No function declaration must have negative arity. In the example, this is the case for the declaration **public** `fun/-1`.
- All functions must be used with the arity with which they have been declared.
- No role that has no actions must occur in a goal. In the example, the secrecy goal requests that `bar(secr)` be a secret of role B even though B has no actions to perform.
- The messages of the communication steps must be synthesizable by the sender. That is, a protocol must be executable as defined in Definition 2.27 on page 27.
- The messages that occur in the goals must be synthesizable by all roles that are mentioned in the goal the latest after their last action.
- No two identifiers must occur such that they are equal in a case-insensitive way but not equal in a case-sensitive way. In the example, `Secr` and `secr` collide.
- The declaration of a fresh variable must not collide with a variable that is either in the initial knowledge of a role or declared as a fresh variable in another action. In the example, `secr` is declared as a fresh variable even though it is already in the initial knowledge of A.

The translator refuses to translate any protocol that does not meet all the requirements listed above since a protocol that is not well-formed has no clear semantics. Note that the synthesizability requirements above imply that there are no uninitialized variables.

4 The Intermediate Representation Format

One of the goals of this thesis is to provide a tool that translates from A&B protocol specifications to the input language of Tamarin. It is usual in computer science not to translate directly from the input language to the output language, but to first parse the code into an intermediate representation format (IR):

$$\text{A\&B} \xrightarrow{\text{Parsing \& Analysis}} \text{IR} \xrightarrow{\text{Code Generation}} \text{Output Language}$$

The IR format decouples parsing and analysis from code generation and adds additional flexibility; most importantly, it allows us to write translators to any output language without having to reimplement the parsing and analysis stage. In this thesis, we only provide a translation to the input language of Tamarin, but since the IR is independent of the target language, new code generators can be added without much effort. This is not the only advantage; having the protocol in a clean representation simplifies checking if it is well-formed significantly.

We now outline the IR that we use in this thesis. It is, like the whole translator, implemented in Haskell and for this reason we have to introduce some aspects of the implementation already here. The intermediate representation format is defined in package `Rewriter.IR` of the implementation of the translator.

The IR is based on the same notions and ideas as the specification of protocols that we gave when we were talking about Alice&Bob notation (see Definition 2.26). Nevertheless, the IR is designed as a convenient basis for the code generation step and consequently stores more information than is strictly necessary. Actions in the IR, for instance, do not only describe the job that a principal should perform but also contain the knowledge that the principal has after executing the action. This spares the code generator the responsibility to keep track of that knowledge. In other words, the IR is devised in such a manner that it contains all relevant information of the analysis so that code generation from it is as simple as possible. There is a short reference for the intermediate representation format in Appendix C.

4.1 Framework

The IR is embedded into the Haskell code of the translator. Therefore we introduce a few general data structures and type definitions before we start to talk about the IR itself.

The following types are defined in package `Parser.Basic` and help improve the readability of the code:

```
type Label      = String;
type RoleName   = String;
type Identifier = String;
```

The chosen names should be self-explaining. For the canonical representation of multiplication, we use multisets like in our discussion of Alice&Bob notation which is why we introduce the following type definition:

```
type Factors = Data.MultiSet.MultiSet Message
```

Note that `Data.MultiSet.MultiSet` is not included in most Haskell setups and that therefore, it is probably necessary to download and install it before the translator can be compiled. The type `Data.MultiSet.MultiSet` only works with data types that are in class **`Ord`** which is why the messages that we define in a moment are an instance of **`Ord`**.

A data type that lies at the very heart of protocols is `Message`. The corresponding implementation in Haskell can be found in package `Parser.Message`:

```
data Message
    = Gamma Identifier Message
    | Var Identifier
    | Str String
    | Concat Message Message
    | Aenc Message Message
    | Senc Message Message
    | Hash Message
```

```
    | Mul Factors
    | Exp Message Message
    | Pk RoleName
    | Sk RoleName
    | K RoleName RoleName
    | Fun Identifier [Message]
    | One
```

The same messages and the same message model as defined in Section 2.2 are used here. The first constructor, `Gamma` represents ghost abstraction. The identifier is used to give a unique name to each such ghost abstraction; we will ignore this for now, it is discussed later in detail. `Var` represents a variable (variables can be both of type Num and $Agent$) and `Str` a constant number (constant numbers are represented as strings in single quotes in A&B, therefore the name). `Concat` denotes the concatenation of two messages, `Aenc` and `Senc` asymmetric and symmetric encryption of a message (first argument) with a key (second argument). `Hash` stands for the hash of a message and `Fun` for function application, where the arguments are given as a list of messages. `Mul` represents multiplication (the factors are represented as a multiset) and `Exp` the exponentiation of a message with another message. `Sk` and `Pk` represent the secret and the public key of a role, `K` the symmetric key of two roles. `One` represents the message 1.

Finally, knowledge is represented as a set of messages. For reasons of readability, we again define a new type in package `Parser.Message`:

```
type Knowledge = Data.Set.Set Message
```

4.2 Representing a Protocol

Based on this framework, we can now define the intermediate representation format. Its definition is given in package `Rewriter.IR` and follows the same ideas as the notion of a protocol that we specified in Definition 2.26. Let us introduce it in a top-down approach by starting with the definition of a protocol:

```
data Protocol = Protocol Identifier [Function] [Role] [Goal]
```

The first argument of the `Protocol` type constructor is the name that is given to the protocol in the A&B specification. If we return to the specification of the protocols in Figure 5, this would be `"CR"`, `"DIFFIE-HELLMAN"` and `"ASW"`, respectively. The second argument is a list of the functions that are declared in this protocol, the third a list of the roles and the fourth a list of the goals.

Functions. Let us continue with the definition of functions. A predefined function is represented as follows:

```
type Function = (Identifier, Integer, Bool)
```

The first element of the triple is the name of the function, the second the arity and the third specifies if the function is public. The A&B function declarations `public fun/3;` and `private bar/1;` would be represented as `("fun", 3, True)` and `("bar", 1, False)`, respectively.

Roles. We have specified roles in Definition 2.25 and declared that a role consists of the initial knowledge that a principal in this role has as well as the actions that are to be performed. A role in the intermediate representation format, however, only consists of the name of the role and a list of actions (the order of the actions in the list matters, of course); this is because an `Action` does not only represent the action itself but also contains information about the knowledge, as we will see in the next paragraph, and consequently there is no need to store this information in the role itself. A role is defined as follows:

```
data Role = Role RoleName [Action]
```

Actions. When we were talking about Alice&Bob protocol specifications, we introduced two types of actions, namely **Send** and **Recv**. We use the same idea in the intermediate representation format with the type constructors `Send` and `Receive`. However, there is also a third type constructor, `Prepare`, whose purpose is to represent the initial knowledge of a role. In some sense, it represents the action of acquiring the initial knowledge. The first action of a role is therefore always a `Prepare` action. Let us give the corresponding type definition before we discuss the details:

```
data Action
    = Prepare Knowledge
    | Send Bool Label RoleName [Identifier] Message Knowledge
    | Receive Label RoleName Message Knowledge
```

The last argument of all the three type constructors is the knowledge that a principal executing the action is supposed to have *after* successfully performing it. In the actions that we defined in Section 2, we did not include the knowledge at all; here we do it with the code generation in mind since this allows for a cleaner implementation. It would contradict the idea of separating analysis and code generation if the code generator had to keep track of the knowledge of a role during a run of the protocol. As explained above, there is no reason to include the initial knowledge in the specification of a role since all the information that is needed is included in the actions.

Let us now take a closer look at `Send` and `Receive`. Both contain that label that is given to the action in the A&B specification. It is used for reference when checking the well-formedness of the protocol and when generating the output code.

Both action types also contain the name of the partner role. This is the third argument in the case of the `Send` constructor; here, the partner role is the receiving role. In the case of the `Receive` constructor, the partner role is specified in the second argument and corresponds to the role sending the message.

The `Send` type constructor has two more arguments than `Receive`. Its very first argument is a `Bool` that indicates if the action is executable, that is, if the message can be constructed from the knowledge. A sending action that is not executable is undefined according to Definition 2.27, in the IR it is simply marked as not executable. The fourth argument contains the list of the fresh names that should be generated before the message is constructed and sent.

The second-to-last argument in the `Send` and `Receive` type constructors is the message that is exchanged over the network. In both cases, it is represented in its constructive form. In the case of `Send` actions it is an obvious choice to represent the message in this format since it provides a direct description of how the message to be sent can be constructed from the knowledge (this is of course the knowledge after adding the freshly generated numbers).

However, it is less clear why we would use the constructive form for `Receive` actions because receiving and analyzing is at first glance completely different from synthesizing messages. We have shown earlier that there are situations where a principal can analyze a message but not reconstruct it from the extracted sub-messages. For example, if B's knowledge contains only the message $\mathbf{pk}(A)$ and he receives the message $\{n_1^{n_2}\}^{\mathbf{a}}_{\mathbf{sk}(A)}$, he can extract $n_1^{n_2}$ but cannot reconstruct $\{n_1^{n_2}\}^{\mathbf{a}}_{\mathbf{sk}(A)}$ from $n_1^{n_2}$ because he does not know $\mathbf{sk}(A)$ as well. Therefore, he has to store the message $\gamma_{\{n_1^{n_2}\}^{\mathbf{a}}_{\mathbf{sk}(A)}}$ as a whole as well as $\gamma_{n_1^{n_2}}$ in order not to lose any messages. This shows that there are two points of view that we can take when receiving a message. On the one hand, we can focus on the new messages that can be learned by analyzing an incoming message; from that perspective, we could represent the message as $\{\gamma_{n_1^{n_2}}\}^{\mathbf{a}}_{\mathbf{sk}(A)}$ since B learns $\gamma_{n_1^{n_2}}$. On the other hand, we can also focus on the messages we have to store since they are not synthesizable from the sub-messages; if looking at the received message from that angle, the constructive form $\gamma_{\{n_1^{n_2}\}^{\mathbf{a}}_{\mathbf{sk}(A)}}$ is a sensible representation because we have to store $\gamma_{\{n_1^{n_2}\}^{\mathbf{a}}_{\mathbf{sk}(A)}}$ as a whole.

Both representations have their advantages and disadvantages. As previously mentioned, we have decided to use the constructive form for the representation of the exchanged message in `Receive` actions because it simplifies some aspects of the code generation step. We come back to this choice in Section 5 where we discuss the translation to the input language of Tamarin and motivate it in more detail.

For illustration of the representation of actions in the IR, consider the last message exchange step of the Diffie-Hellman protocol in Figure 5. It is perfectly suited to demonstrate some of the points we have discussed above:

```
[dh_3] A -> B (n) : senc{n}(g^(x*y));
```

We now give the corresponding actions of the sender (A) and the receiver (B) in the IR format.

- Let us first look at the sending principal A. After generating the fresh number, the knowledge of the principal in role A is $\chi_A = \{A, \mathbf{pk}(A), \mathbf{sk}(A), \gamma_{g^y}, x, n\}$ and as a consequence, we have $cf_{\chi_A}(\{n\}^{\mathbf{s}}_{g^{\odot\{|x,y|\}}}) = \{n\}^{\mathbf{s}}_{(\gamma_{g^y})^x}$. Hence the constructive form is defined and the action is executable. The corresponding `Send` action is then:

  `Send` **`True`** `"dh_3" "B" ["n"]` $\{n\}^{\mathbf{s}}_{(\gamma_{g^y})^x}$ $\{A, \mathbf{pk}(A), \mathbf{sk}(A), \gamma_{g^y}, x, n\}$

  Note that we have not used Haskell notation for the messages because they would become very long, causing the knowledge to span several lines ($\{n\}^{\mathbf{s}}_{(\gamma_{g^y})^x}$ would be represented as `Senc (Var "n") ((Gamma "alpha" (Exp (Var "g") (Var "y"))) (Var "x"))`). We resort to this notation several times in this thesis.

- On the receiving side, B learns n since he can construct the decryption key $g^{x \odot y}$ as $(\gamma_{g^x})^y$. After receiving, his knowledge therefore is $\chi_B = \{B, \mathbf{pk}(B), \mathbf{sk}(B), \gamma_{g^x}, y, n\}$ and we represent the message in the action as $cf_{\chi_B}(\{n\}^{\mathbf{s}}_{g^{\odot\{|x,y|\}}}) = \{n\}^{\mathbf{s}}_{(\gamma_{g^x})^y}$. The corresponding `Receive` action is then:

  `Receive "dh_3" "A"` $\{n\}^{\mathbf{s}}_{(\gamma_{g^x})^y}$ $\{B, \mathbf{pk}(B), \mathbf{sk}(B), \gamma_{g^x}, y, n\}$

There is one important detail about messages that we have ignored so far, namely the identifier in the type constructor of ghost messages (`Gamma Identifier Message`). It plays a crucial role when it comes to the generation of (TAMARIN) code, especially in relation with actions. However, it makes sense to wait with discussing it until after we have introduced the input language of TAMARIN since this will allow us to support our explanations with examples. Therefore we discuss ghost messages only in Section 5.2.

Security Goals. The representation of the security goals in the IR is straightforward.

```
data Goal
    = Secret Label Message [RoleName]
    | WeakAuth Label RoleName RoleName [Message]
    | StrongAuth Label RoleName RoleName [Message]
```

We refer to non-injective agreement as "weak authentication" (`WeakAuth`) and injective agreement as "strong authentication" (`StrongAuth`) here. The meaning of the three types of goals corresponds one-to-one to the definitions given in Section 3.4 when we were discussing the security goals of A&B. Instead of further explanations, let us give examples to illustrate the relationship between the declaration of security goals and their representation in the IR. The CR protocol in Figure 5 specifies one security goal of each type:

1. `[n_secret] n` **`secret of`** `C, R;`
2. `[authNonInj] C` **`non-injectively agrees with`** `R` **`on`** `n;`
3. `[authInj] C` **`injectively agrees with`** `R` **`on`** `n;`

This translates to the following instantiations of `Goal`:

1. `Secret "n_secret" (Var "n") ["C", "R"]`
2. `WeakAuth "authNonInj" "C" "R" [(Var "n")]`
3. `StrongAuth "authNonInj" "C" "R" [(Var "n")]`

4.3 Example

We used the CR example to illustrate protocols when we discussed Alice&Bob notation in Section 2.8.2. Later, we gave the CR protocol in the A&B language in Figure 5. Now, we would like to end this section by giving an explicit implementation of the corresponding intermediate format. It can be found in Figure 7. Note the common features of the representation of the CR protocol that we gave in Section 2.8.2 on page 27 and the corresponding intermediate representation.

```
import Parser.Message
import Rewriter.IR
import qualified Data.Set as S

protocol = Protocol "CR" []
    [ Role "C"
        [ Prepare (S.fromList [Sk "C", Pk "C", Var "C", Pk "R"])
        , Send True "cr_1" "R" ["n"] (Aenc (Var "n") (Pk "R"))
            (S.fromList [Sk "C", Pk "C", Var "C", Pk "R", Var "n"])
        , Receive "cr_2" "R" (Hash (Var "n"))
            (S.fromList [Sk "C", Pk "C", Var "C", Pk "R", Var "n"])
        ]
    , Role "R"
        [ Prepare (S.fromList [Sk "R", Pk "R", Var "R"])
        , Receive "cr_1" "C" (Aenc (Var "n") (Pk "R"))
            (S.fromList [Sk "R", Pk "R", Var "R", Var "n"])
        , Send True "cr_2" "C" [] (Hash (Var "n"))
            (S.fromList [Sk "R", Pk "R", Var "R", Var "n"])
        ]
     ]
     [ Secret "n_secret" (Var "n") ["C", "R"]
     , WeakAuth "authNonInj" "C" "R" [(Var "n")]
     , StrongAuth "authNonInj" "C" "R" [(Var "n")]
     ]
```

Figure 7: The intermediate representation of the CR protocol from Figure 5.

5 Translation to Tamarin

Before we talk about some aspects of the translation step, we would like to say a few words about the Tamarin prover and, in particular, about its input language. In general, it is not decidable if a protocol fulfills certain security properties and for this reason, an automated verifier can only be applied to a subset of all interesting problems.

The Tamarin prover [13] is the first tool that is based on a verification theory that supports unbounded falsification and verification making use of loops and non-monotonic state. What is more, Tamarin supports bilinear pairing as well as Diffie-Hellman multiplication and exponentiation which significantly widens the class of protocols that can be checked by an automated verifier. Tamarin can be downloaded from the website of the information security group at ETH Zurich [10].

The verification theory of Tamarin is out of scope of this thesis; we give here only a short and incomplete introduction to the input language of Tamarin where we put an emphasis on the aspects that are of interest for our purpose. We refer to the documentation that comes with Tamarin for more details.

5.1 Tamarin and its Input Language

State is modeled as a finite multiset of labeled facts in Tamarin. A protocol is modeled by giving a set of rewriting rules for these facts and security goals are modeled by specifying first-order logic formulas (so-called *lemmas*) that can then be analyzed by the tool, where adversary involvement is considered. A Tamarin file has the extension `.spthy` and the following structure:

```
theory name
begin
/* Body: protocol and security goal specifications */
end
```

where `name` can be any identifier. We can see that Tamarin uses C-style comments. Let us now talk about the body of a Tamarin protocol specification.

5.1.1 Messages and Functions

Tamarin uses a similar notation for messages as the A&B protocol specification language. In particular:

- Constant numbers are represented as strings in single quotes such as `'1'` or `'hello'`.
- Variables are represented by identifiers such as `m`, `a` or `key`.
- The concatenation of two messages `m1` and `m2` is represented as `<m1, m2>` or `m1, m2` if unambiguous. Note that Tamarin uses a comma, not a dot.
- The Diffie-Hellman multiplication of the messages `m1` and `m2` is denoted by `(m1 * m2)`.
- The Diffie-Hellman exponentiation of a message `m1` with a message `m2` is denoted by `(m1^m2)`.
- The application of the n-ary function `fun` with the arguments `m1, ..., mn` is denoted by `fun(m1, ..., mn)`. Hashing, inversion of messages, encryption and further operations are represented as functions in Tamarin. The parentheses can be omitted for nullary functions and there is a special syntax for binary functions. The binary function `aenc` can be applied either as `aenc(m1, m2)`, or as `aenc{m1}m2`. This syntax does not make sense in all cases, but helps improve readability for encryption and therefore we use it when generating Tamarin code.

Functions need to be declared before they can be used. This is done after the **`functions`** keyword:

```
functions: fun/1
```

In this example, the unary function `fun` is declared. Functions in Tamarin can be either public or private, which has exactly the same semantics as in A&B, that is, the adversary can apply public functions while he cannot apply private functions. Functions are public by default, if we want to declare a function such as `fun` as private, we can do this as follows:

```
functions: fun/1 [private]
```

5.1.2 Equations and Built-in Theories

The message model is determined in TAMARIN by specifying equations. For example, if we want to model symmetric encryption, we can do this by first declaring the binary function `senc` for encryption and the binary function `sdec` for decryption. Both functions should of course be public since they can be applied by the adversary. Then, we can specify the symmetric decryption property in an equation. This looks as follows:

```
functions: senc/2, sdec/2
equations: sdec(senc(m, key), key) = m
```

Recall property (2) from page 11, namely $\{\{m\}_k^{\mathbf{s}}\}_k^{\mathbf{s}} =_M m$. There, we did not make a distinction between decryption and encryption but used the same function for both directions. Unfortunately, using `senc(senc(m, key), key) = m` can lead to protocol specifications that cannot be treated due to restrictions of the prover that underlies TAMARIN. As a consequence we use the rule `sdec(senc(m, key), key) = m` from above in the TAMARIN code that is output by the translator. The same restriction applies in the case of asymmetric encryption and therefore we use `aenc` for encryption and `adec` for decryption.

Note that this can lead to collisions with the message model that we use in A&B since the message `senc{senc{m}key}key` equals `m` in our Alice&Bob message model but does not equal `m` in the message model we use in TAMARIN. For this reason, we have to make sure that messages are always kept in canonical form since this requires that we represent $\{\{m\}_k^{\mathbf{s}}\}_k^{\mathbf{s}}$ as m. We never have to use decryption explicitly because TAMARIN can work out by itself how messages can be analyzed when given the appropriate equations.

While we can explicitly declare new functions and add equations, there is a basic setup that is always there implicitly in TAMARIN, namely:

```
functions: fst/1, pair/2, snd/1
equations: fst(<x.1, x.2>) = x.1, snd(<x.1, x.2>) = x.2
```

The rules above express that messages can be paired on the one hand and that concatenation can be decomposed on the other hand (`pair/2` represents concatenation, i.e., `pair(m1, m2) = <m1, m2>`). The other function declarations and equations that we need have to be given explicitly. To spare us the necessity to restate the same equations and functions each time, there is a number of built-in equational theories in TAMARIN. For example, there is the equational theory `symmetric-encryption` which can be included in the specification with the line

```
builtins: symmetric-encryption
```

When the protocol is read by TAMARIN, this line is replaced by the appropriate function and equation declarations, that is, the function declarations of `senc` and `sdec` and the equation we stated above. Let us now quickly go through the other built-in equational theories that we use.

Firstly, there is `hashing`. It declares the public unary function `h`. No new equations are added. In other words, `h` is treated as a normal function.

Secondly, there is `diffie-hellman`. This adds the operators $\star$ and $\hat{}$ for Diffie-Hellman multiplication and exponentiation. Furthermore, a complex message model is added. Recall our equational theory M on page 11 in which multiplication and exponentiation are modeled by properties (5) to (10). The equational theory used by TAMARIN is of course much more involved since the capabilities of both the adversary and honest principals need to be modeled. For example, there is the property $g^a * g^b = g^{(a+b)}$ which we did not model (we did not even define addition). The complete proving theory can be found in the CSF paper about TAMARIN [13]. The `diffie-hellman` theory is a special case since it adds capabilities to TAMARIN internally while other theories are nothing but syntactic sugar for equation and function declarations.

There are further built-in theories, namely `asymmetric-encryption`, `bilinear-pairing` and `signing`. We do not use them in this thesis and will therefore not elaborate on them.

5.1.3 Facts and Rewriting Rules

We have mentioned that TAMARIN is based on multisets of labeled facts and rules that specify how the facts in such a multiset can be rewritten. A fact is denoted by `F(t_1, ..., t_k)` where `F` is the label

and `t_1, ..., t_k` is a list of parameters. There are a few built-in facts with pre-defined semantics. For example, the fact `Out(m)` represents the fact that the message `m` was sent over the network at some time and the fact `In(m)` expresses that `m` can be received. Another important built-in fact is `Fr(n)` that states that `n` is a freshly generated number.

Facts can be either *linear* or *persistent*; a linear fact can only be consumed by one rule and is then removed from the multiset state, persistent facts can be consumed arbitrarily often. In the TAMARIN input language, persistent facts are marked by a prefixed exclamation mark. For instance, the persistent fact that the publicly known principal in role `A` has public key `pkA` can be modeled as the persistent fact `!Pk(A, pkA)`.

How facts can be rewritten is specified by *labeled multiset rewriting rules*. They have the shape

```
rule label:
    [ L ] --[ A ]-> [ R ]
```

where `L`, `A` and `R` are comma-separated lists of facts. The facts in `L` are called the *premises*, the facts in `A` the *actions* and those in `R` the *conclusions*. Note that the name "actions" is somewhat unfortunate since it collides with our notion of an action that is executed by a principal; we therefore refer to TAMARIN's actions as *statements* here. From now on, we refer to labeled multiset rewriting rules simply by "rule".

Certain properties of values are indicated by prefixes or postfixes. In particular, fresh values are prefixed with a ~ or postfixed with `:fresh`. Similarly, public values are prefixed with $ or postfixed with `:pub`. Temporal values are prefixed with a `#` or postfixed with `:temp`.

A rule can only be applied if all of the premises are satisfied, that is, if there is a matching fact in the multiset state for each of the facts in `L`. When the rule is applied, linear facts in the multiset are consumed (removed from the state) while persistent facts are left untouched and the facts in R are added to the multiset state. The statements in `A` do not influence state. Their purpose is to express that "something has happened". These statements can be used in lemmas that describe the properties that the protocol should achieve (i.e., security goals).

Let us illustrate this with the help of a small example. First, we would like to give the predefined rule for the generation of fresh values (the label has been omitted here):

```
[ ] --[ ]-> [ Fr(~x) ]
```

This rule expresses that we can generate a fresh value out of nothing (without any premises), that is, we can always add the fact `Fr(~x)` to the multiset without any preconditions. The value `~x` is prefixed with a tilde to mark it as a fresh value, i.e., it has been randomly chosen and can therefore be assumed to appear nowhere else. The fact `Fr` can then be used by the rule `Asymmetric_key_setup` that models the possibility of any entity to acquire both a public and a private key. Since `Fr` is a linear fact, it is consumed (i.e., removed from the multiset) when rule `Asymmetric_key_setup` is applied (this only makes sense because we want to prevent that a fresh value is reused):

```
rule Asymmetric_key_setup:
    [ Fr(~f) ] --[ ]-> [ !Sk($A, sk(~f)), !Pk($A, pk(~f)) ]
```

The rule states that an entity `A` (that is publicly known, hence the dollar sign) can consume a fresh value `~f` from the multiset state and then generate the two persistent facts `!Sk` and `!Pk` that bind a secret key and the corresponding public key to `A`. The functions `pk` and `sk` represent the generation of the public and the private key from a fresh value `~f`. Of course, they need to be defined in the **`functions`** block.

Furthermore, we can express that the public key can be obtained by any entity by the following rule:

```
rule Publish_public_keys:
    [ !Pk(A, pkA) ] --[ ]-> [ Out(pkA) ]
```

We quite simply state that the public key is sent to the network as often as desired (`!Pk` is persistent). Note that the markers for fresh and public values are used no longer here since we take facts from the state.

5.1.4 Lemmas

Lemmas are used to describe what security goals should be achieved by a security protocol. They are guarded fragments of first order logic. Let us just give a dummy example to demonstrate the idea.

First, suppose we have the following rewriting rule

```
rule Dummy:
    [ P(m) ] --[ A(m) ]-> [ ]
```

The rule expresses that we can consume a linear fact `P(m)` from the multiset state while nothing new is added to the state. Here, we also have a statement `A(m)`. Statements are "made" when the rule is applied but have no effect whatsoever on the multiset state. Statements can be used in lemmas. For example, we can express the property that this rewriting rule can actually be applied by a protocol during execution by stating the following lemma:

```
lemma executable:
    exists-trace
    " Ex m #t . A(m) @ #t "
```

A lemma starts with the keyword **`lemma`**, followed by an identifier that gives a name to the lemma. There are two types of lemmas; the one type requests that the lemma must hold for all traces (keyword `all-traces`), the other type requires that there is at least one trace for which the lemma holds (keyword `exists-trace`). Then, the actual lemma follows. In the example above, we require that there exists a point in time `#t` at which a statement `A(m)` is made for some value `m`. We do not discuss lemmas in more detail here since we only need one lemma template per security goal type and can then simply insert the matching values as we will see later.

5.2 Translation from IR to Tamarin

We would now like to emphasize a few aspects of how we generate Tamarin code. First of all, we have to talk about how we specify our equational theory, especially since we also have to fix the adversary capabilities, something which we did not do for the A&B language itself. Then we can discuss how the actions taken by principals can be expressed in Tamarin and finally, how the security goals can be cast into lemmas.

Recall the CR protocol specified in Figure 5 on page 30 and the corresponding IR that is given in Figure 7 on page 41. We will resort to this protocol repeatedly to illustrate some aspects of the translation.

5.2.1 Modeling the Capabilities of Honest Principals

We have to express the message model and the capabilities of principals (both honest and dishonest) from Section 2 in Tamarin. On the one hand, there are the honest principals. In order to specify their capabilities, we need to express the framework that emerges from the synthesis rules (Definition 2.5 on page 15), the analysis rules (Definition 2.8 on page 16) and equational theory M (Section 2.2.2 on page 11). On the other hand, we also need to specify the capabilities of the adversary, something that is mostly predetermined by Tamarin. Let us not beat around the bush and give the function and equation declarations straight away:

```
functions: adec/2, aenc/2, pk/1, sk/1
builtins:  symmetric-encryption, diffie-hellman, hashing
equations: adec(aenc(x.1, sk(x.2)), pk(x.2)) = x.1,
           adec(aenc(x.1, pk(x.2)), sk(x.2)) = x.1
```

We explained above that all of the built-in theories except for `diffie-hellman` and `bilinear-pairing` are nothing but syntactic sugar for function declarations and equations. If we unfold the built-in theories, this results in the following equivalent specification:

```
builtins:  diffie-hellman
functions: adec/2, aenc/2, fst/1, h/1, pair/2, pk/1, sdec/2, senc/2,
           snd/1
equations: adec(aenc(x.1, sk(x.2)), pk(x.2)) = x.1,
           adec(aenc(x.1, pk(x.2)), sk(x.2)) = x.1,
           sdec(senc(x.1, x.2), x.2) = x.1,
           fst(<x.1, x.2>) = x.1,
           snd(<x.1, x.2>) = x.2
```

Note that we do not use the built-in theory `asymmetric-encryption` but set up asymmetric encryption by declaring the functions `adec`, `aenc`, `sk` and `pk` and the two equations that express their properties. We do this because `asymmetric-encryption` only allows the decryption of messages that are encrypted with the public key, not with the secret key. It is not possible to extend `asymmetric-encryption` to allow decryption of messages that were encrypted with the secret key without adding unwanted capabilities for the adversary. In the setup above, the adversary can construct public-secret key pairs, but cannot construct the public key from a secret key or vice-versa.

In the next proposition, we claim that these definitions actually represent the message model that we introduced in Section 2.

Proposition 5.1. *The capabilities of honest principals that are induced by the function declarations, equations and the built-in theory `diffie-hellman` from above are equivalent to the capabilities that arise from Definitions 2.5 and 2.8 and the properties from Section 2.2.2.*

This implies that the protocols that we specify are actually executable in Tamarin. The capabilities of the adversary follow immediately from the definitions above. In particular, the adversary has all the capabilities that honest principals have (with the exception of applying private functions).

5.2.2 Setting Up the Knowledge

We have seen in the previous section how we can model the capabilities of honest principals. Therefore we can now proceed to the initial knowledge, the last piece that we have to include in a specification of a protocol before we can proceed to the discussion of the actual communication. We can do this by defining multiset rewriting rules.

The first rule that we introduce allows the generation of arbitrarily many secret-public-key pairs:

```
rule Asymmetric_key_setup:
    [ Fr(~f) ] --[]-> [ !Sk($A, sk(~f)), !Pk($A, pk(~f)) ]
```

We represent the secret key of a principal A by the persistent fact `!Sk(A, secK)` and the public key by `!Pk(A, pubK)`. Both keys are generated by applying a function to a fresh value `~f`, i.e., we have `secK = sk(~f)` and `pubK = pk(~f)`. We have chosen this setup since it allows the adversary to generate public-secret key pairs, but at the same time does not allow the construction of the secret key from the public key (or vice-versa) if he does not know the value from which they were generated.

In A&B, public keys are assumed to be publicly known (in particular, to the adversary). We model this by the following rule:

```
rule Publish_public_keys:
    [ !Pk(A, pkA) ] --[]-> [ Out(pkA) ]
```

Now we come to the rule that generates the initial knowledge of all participants. Knowledge can be represented by facts in Tamarin. For example, the implicit knowledge $\{A, \mathbf{pk}(A), \mathbf{sk}(A)\}$ of a principal in role A can be represented by a fact of the form `Knowledge(A, pubK, secK)`. We have required in A&B that every communication step has a unique label which comes in handy now. We represent the knowledge of principal `A` after the execution of the message exchange step with the label `[label]` by `St_label_A(...)`. The fact for representing the initial knowledge of `A` is called `St_init_A(...)`.

If we take a glance at the intermediate representation of the CR protocol, we can see that the initial knowledge of role `C` is `[Sk "C", Pk "C", Var "C", Pk "R"]`, the one of role `R [Sk "R", Pk "R", Var "R"]`. We can use rule `Asymmetric_key_setup` to get the public and private keys of the participants; `C` and `R` are public names. This means that we can set up the initial knowledge with the following rule:

```
rule Init_Knowledge:
        [ !Pk($C, pk(k_C)),
          !Pk($R, pk(k_R)),
          !Sk($C, sk(k_C)),
          !Sk($R, sk(k_R)) ]
        --[  ]->
        [ St_init_C($C, sk(k_C), pk(k_C), pk(k_R)),
          St_init_R($R, sk(k_R), pk(k_R)) ]
```

Here, k_C and k_R are the values from which the public and secret keys are generated. Note that this rule ensures that equal messages actually have equal values in the initial knowledge of different principals and that the public key **pk**(R) that C possesses is authenticated (this is why there is only one rule to set up the initial knowledge for all principals that take part in a run of a protocol).

In the CR protocol, there are only agent names and public and secret keys in the initial knowledge, all of which are either publicly known or generated by rule Asymmetric_key_setup. There is no general variable in the initial knowledge. We have defined that variables in the initial knowledge are equal for all participants of a protocol run and that the value of the variable is fresh each time, i.e., we have to model them as fresh numbers.

If we assume that C additionally has the message **h**(n) in her initial knowledge, we can do this by generating n as a fresh number and adapting the rule above as follows:

```
[ ..., Fr(~n)
--[  ]->
[ St_init_C($C, sk(k_C), pk(k_C), pk(k_R), h(~n)), ... ].
```

We have now defined the message model and how the initial knowledge can be set up in Tamarin. With this, we have now the framework in which we can express the actual communication between honest principals.

5.2.3 Communication Steps

The IR provides us with two types of communication steps, Send and Receive. Both of them require that the previous action of the role has been executed. Assume that the current action has the label [current] and the previous action has the label [previous].

We represent each action of a principal by one rule. Tamarin works with multiset rewriting rules and we have to make sure that the rule that expresses the action with label [current] can only be applied when the action with the label [previous] has been applied ("executed") before. We do this with the knowledge; remember that the knowledge that P has after executing the action with label [previous] is represented by the fact St_previous_P(...). Therefore, we use St_previous_P(...) as a premise for the rule that expresses the action with label [current]. This works since the state of a principal consists of nothing but the current position in the protocol and the knowledge, information which is all expressed by the fact St_previous_P(...). Hence, the pattern of all rules that express a Send or Receive action is as follows:

```
rule current_P:
        [ St_previous_P(P, sk(k_P), pk(k_P), ...), ... ]
        --[ ... ]->
        [ ..., St_current_P(P, sk(k_P), pk(k_P), ...) ]
```

If there is no previous action, then the corresponding label is [init] and the corresponding fact St_init_P.

Let us now look at both sending and receiving actions separately. For a sending action, we have to add the Out fact for expressing that the message has been sent. However, before we can send, we need to generate the fresh numbers. In the general case, a fact representing a sending action of a sending principal S looks as follows, where message is in constructive form (as delivered by the IR); we will shortly discuss this in more detail:

```
rule action_current_S:
        [ St_previous_S(S, sk(k_S), pk(k_S), ...),
          Fr(~n1), ..., Fr(~ni), ...]
        --[ ... ]->
        [ ..., Out(message),
          St_current_S(S, sk(k_S), pk(k_S), ..., ~n1, ..., ~ni, ...)].
```

The receiving of a message that was sent before (an Out fact) is expressed by an In fact in the premises. The receiving action of a principal R has the following general structure:

```
rule action_current_R:
        [ St_previous_R(R, sk(k_R), pk(k_R), ...),
          In(message), ... ]
```

```
        --[ ... ]->
        [ ..., St_current_R(R, n, sk(k_R), pk(k_R), ...) ]
```

Let us demonstrate this with the CR protocol. The `Send` and `Receive` actions of the message exchange step with label `[cr_1]` are expressed in TAMARIN by the following two facts: :

```
rule cr_1_C:
        [ St_init_C(C, sk(k_C), pk(k_C), pk(k_R)),
          Fr(~n) ]
        --[ ... ]->
        [ Out(aenc{~n}pk(k_R)),
          St_cr_1_C(C, ~n, sk(k_C), pk(k_C), pk(k_R))].

rule cr_1_R:
        [ St_init_R(R, sk(k_R), pk(k_R)),
          In(aenc{n}pk(k_R)) ]
        --[ ... ]->
        [ St_cr_1_R(R, n, sk(k_R), pk(k_R)) ]
```

5.2.4 Security Goals

The CR protocol declares one of each of the three security goal types (see Section 3.4 for a discussion of the security goals). Let us restate them here in the intermediate representation format:

```
Secret "n_secret" (Var "n") ["C", "R"]
WeakAuth "authNonInj" "C" "R" [(Var "n")]
StrongAuth "authNonInj" "C" "R" [(Var "n")]
```

Remember that `n` only needs to be secret (i.e., not known by the adversary) if both `C` and `R` successfully finish their runs of the protocol and agree on `n`. Similarly, authentication goals only need to hold if the claiming principal finishes protocol execution. In the CR protocol, `C` claims both non-injective and injective agreement with `R` on `n`. This means that `R` should state that he possesses `n` as soon as he has received it and `C` should claim agreement when she finished her last action. This is where the statements in the rewriting rules come into play. `R` states that he possesses `n` after executing `[cr_1]` since this is the first time when he possesses `n`. This is done with the statements `Running_authNonInj(n)` and `Running_authInj(n)` for non-injective and injective agreement, respectively:

```
rule cr_1_R:
        [ ... ]
        --[ Running_authNonInj(n),
            Running_authInj(n) ]->
        [ ... ]
```

`C`, on the other hand, claims agreement after her last action `[cr_2]` with the statement `Commit_authNonInj(n)` and `Commit_authInj(n)`. Furthermore, since this is her last action, `C` also claims secrecy on `n`:

```
rule cr_2_C:
        [ ... ]
        --[ Secret_n_secret_C(n),
            Commit_authNonInj(n),
            Commit_authInj(n) ]->
        [ ... ]
```

`R` also claims secrecy in his last action which is expressed by the following rule:

```
rule cr_2_R:
        [ ... ]
        --[ Secret_n_secret_R(n) ]->
        [ ... ]
```

Based on these rules and the statements that are made, we can now formulate the lemmas that correspond to the security goals. Let us start with secrecy:

```
lemma n_secret:
    " not(
        Ex msg #i1 #i2 #j .
            Secret_n_secret_C(msg) @ #i1 &
            Secret_n_secret_R(msg) @ #i2 &
            K(msg) @ #j
    )"
```

The lemma states that it must not happen that *both* `C` and `R` claim that `msg` is a secret at any time and that there is another point in time where the adversary learns `msg` (expressed by the built-in fact `K`). Note that the lemma only holds because we assume that `msg` has a different value in every run of the protocol (where it is freshly created in the rule that sets up the initial knowledge fact).

The lemma for non-injective agreement looks as follows:

```
lemma authNonInj:
    " (All m1 #i .
        Commit_authNonInj(m1)@ #i
        ==>
        (Ex #j . Running_authNonInj(m1) @ #j & #j < #i)
    )"
```

It expresses that whenever `C` claims non-injective agreement on a message, then there is an *earlier* point in time where `R` knows the message as well. In particular, `C` and `R` have the same value for the message.

Injective agreement is very similar, the only difference is that we additionally require that every successful run (i.e., the principal that claims agreement terminates his run) of the protocol is unique. We can do this by requiring that there is only one commit statement for the same value:

```
lemma authInj:
    " (All m1 #i .
        Commit_authInj(m1)@ #i
        ==>
        (Ex #j . Running_authInj(m1) @ #j & #j < #i) &
        (not (Ex #j . Commit_authInj(m1) @ #j & not (#i = #j)))
    )"
```

We now have the tools to translate A&B protocol specifications to Tamarin. However, we would like to remark here that Tamarin can not alway handle the code produced by the translator. It is undecidable in general if a protocol fulfills the requested security properties and consequently, there are cases where Tamarin does not terminate. Sometimes, this problem can be solved by slightly rewriting the code. In other cases, one can try to specify typing lemmas to reduce the size of the search space. However, all of the protocols presented in this thesis lead to code that works without any modifications.

There are also protocols that cannot be handled at all because some internal restrictions of the Tamarin prover (for example, multiplication restriction) are violated. In these cases, Tamarin complains that the protocol specification is not well-formed and has to be rewritten manually.

5.3 Ghost Messages

We have promised earlier when we were discussing the intermediate representation format that we would make clearer what the purpose of the identifier in the constructor `Gamma Identifier Message` is. Let us start with an example again, this time the ASW protocol from Figure 5 on page 30. The sub-messages `pk(A)`, `sk(A)`, `m` and `h(n_1)` can be extracted from `aenc{pk(A) . pk(B) . m . h(n_1)}sk(A)` by `B` since he knows `pk(B)`. However, `h(n_1)` cannot be further analyzed and since `B` does not posses `n_1`, he has to store the message `h(n_1)`. It is put under ghost abstraction since it is not atomic. The corresponding representation in the IR is then `Gamma "alpha" (Hash (Var "n_1"))`. Here, the `"alpha"` is nothing but an identifier with which we refer to the ghost of `h(n_1)`. Furthermore, `aenc{pk(A) . pk(B) . m . h(n_1)}sk(A)` cannot be reconstructed from the sub-messages because `B` does not know `sk(A)`. Consequently, `B` has to store this message as a ghost as well; it is represented in the intermediate representation language as `(Gamma "beta" (Aenc ... (Sk "A")))`, where we have abbreviated the message a little.

This kind of representation has a direct application when expressing B's Receive action in Tamarin. The receiving of the message **aenc**{**pk**(A).**pk**(B).m.**h**(n_1)}**sk**(A) can be represented in Tamarin by the fact In(senc{pkA, pkB, m, h(n_1)}skA) (which would be perfectly legal even though not all sub-messages can be decomposed). However, since B cannot reconstruct the message from the sub-messages and therefore has to remember the whole message, we write In(beta) instead.

We lose any information about the structure of the message with this representation. Fortunately, Tamarin provides us with the possibility to clarify by adding a **let** block to the rule in which we can specify beta = senc{pkA, pkB, m, alpha}skA. We do not clarify on alpha since it cannot be analyzed. This looks as follows:

```
rule asw1_B:
    let
        beta = aenc{<pk(k_A), pk(k_B), m, alpha>}sk(k_A)
    in
        [ St_init_B(A, B, sk(k_B), pk(k_A), pk(k_B)),
          In(beta) ]
        --[ Running_weakA(m),
            Running_strongA(m) ]->
        [ St_asw1_B(A, B, m, sk(k_B), pk(k_A), pk(k_B), beta, alpha) ]
```

This is exactly why we represent the messages in constructive form in the IR even in Receive actions. This is not the only possible way to represent how messages are analyzed and what information needs to be stored. However, we find this representation most informative and straightforward for several reasons. First of all, we represent composed messages that we treat as one message by one symbol; ghost symbols in the knowledge are always represented by the representatives (alpha and beta in the above rule). Furthermore, this representation makes clear in what order messages are analyzed. And, finally, the stepwise analysis of the message shows nicely what checks can be performed (Tamarin does the checks we require the principals to do automatically by requiring that the premises are fulfilled).

Representatives also go well with passing messages from one action to the next in the knowledge and with sending. Let us demonstrate this by also giving the rule that expresses the sending action of B in message exchange step [asw2]. Note that the structure of beta is of no importance at all in this particular rule:

```
rule asw2_B:
        [ St_asw1_B(A, B, m, sk(k_B), pk(k_A), pk(k_B), beta, alpha),
          Fr(~n_2) ]
        --[  ]->
        [ Out(aenc{<beta, h(~n_2)>}sk(k_B)),
          St_asw2_B(A, B, m, ~n_2, sk(k_B), pk(k_A), pk(k_B),
                    beta, alpha) ]
```

6 Implementation and Tamarin Output

We would now like to talk about some aspects of the implementation of the translator. It consists of two parts: a program that reads A&B protocol specifications and generates the IR and a program that generates Tamarin code from the IR. The Haskell source code can be found at `http://www.infsec.ethz.ch/research/software/tamarin`. See Appendix D for instructions on compilation and installation.

We do not discuss the implementation in detail here but only give a rough overview of its structure where we pick out some of the more important or interesting pieces of code and have a closer look at them. The program is written completely in Haskell and is Haddock-commented. A structural overview of the packages is given in Figure 8.

6.1 Outline of the Translator

We have explained in the previous section that the translation is split in two steps. First, the A&B code is parsed into the IR format from which Tamarin code is then produced. In fact, there is also a third step since A&B specifications are first parsed into a *parse tree*. The complete picture looks as follows:

$$\text{A\&B} \overset{\text{Parsing}}{\longrightarrow} \text{Parse Tree} \overset{\text{Rewriting}}{\longrightarrow} \text{IR} \overset{\text{Code Generation}}{\longrightarrow} \text{Output Language}$$

This predetermines the structure of this chapter. We first discuss the parsing step and parse trees. Then, we come to the rewriting step which is the most interesting part since most of the analysis happens there. Finally, we say a few words about the translation to the output language, Tamarin in our case. We have already discussed this code generation step in Section 5 and therefore only talk about a few implementation-specific aspects.

6.2 Parsing

The parser is implemented in package `Parser` where the main function is `Parser.ProtocolParser.anbParser`. It parses A&B protocol specifications into parse trees.

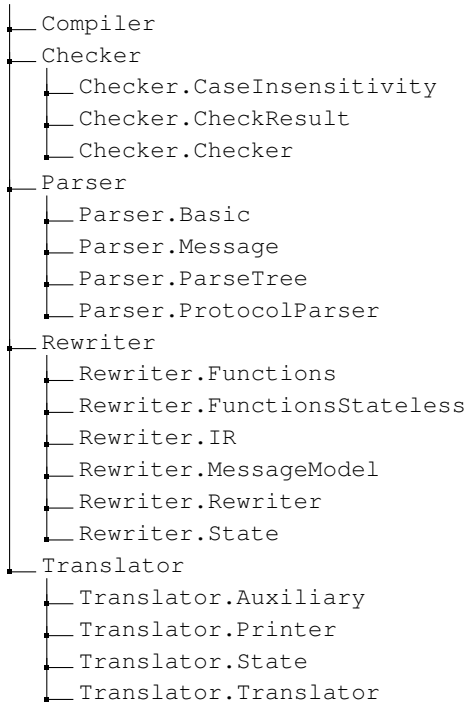

Figure 8: The Haskell packages of the implementation.

6.2.1 Parse Trees

`Parser.ParseTree.ParseTree` is a tree data structure that represents the A&B input. While the intermediate representation format represents a protocol after it has been analyzed and decomposed into roles, the parse tree represents a protocol in its "raw" format. A message exchange step, for example, is represented by the structure

```
data Action
    = Action Label RoleName RoleName [Identifier] M.Message
```

See the implementation for the complete parse tree specification.

The `Parser.ParseTree` package does not only contain the parse tree definition itself but also provides a number of helper functions for working with the parse tree. These are useful during the rewriting stage when we access information from the parse tree frequently.

6.2.2 Parser

The implementation of the parser is based on the monadic `Text.Parsec` package. It is built by composing simple parsers into parsers for complex structures. The following function, for instance, parses a label in brackets such as `[cr_1]` by applying `m_brackets` to the parser `ident`.

```
label :: Parser Label
label = m_brackets ident
```

The monadic nature of Parsec allows for a concise and readable way of composing parsers. For example, the `secret` function parses a secrecy goal (which starts with a label):

```
secret :: Parser Goal
secret = do l <- label
            m <- msg
            m_reserved "secret" >> m_reserved "of"
            ls <- sepBy1 ident (m_symbol ",")
            return (Secret l m ls)
```

The `msg` function parses a message, `m_reserved` parses a reserved keyword, and `sepBy1 ident (m_symbol ",")` parses a comma separated list of identifiers (where at least one identifier must occur). In the last line, the parse tree of the secret is composed and returned.

One aspect of the parser that we would like to emphasize is how precedence and associativity of the operators (see Section 3.2) have been implemented. Operator precedence needs to be implemented with particular care since a left-recursive grammar can lead to non-terminating loops. We use the following grammar for our message parser (note that this is no Haskell code):

```
msg  ::= comT
comT ::= mulT '.' comT | mulT
mulT ::= xorT '*' mulT | xorT
xorT ::= bscT '^' xorT | bscT
bscM ::= '(' comT ')' | '<' comT '>' | builtinFun
       | generalFun | str | var
```

The parser to the left of a `|` is applied first; the one on the right-hand side is applied only if the left-hand side fails (this behavior is implemented in Parsec by the function `<|>`). In `comT`, for example, the parser first tries to parse something of the form `mulT '.' comT` and only resorts to parsing something of the form `mulT` if this fails. In this sense, concatenation is evaluated at the top level. The parser tries to parse a multiplicative term only if no concatenation can be parsed. Similarly, it will only try exponentiation if no multiplication can be parsed. Basic messages as specified by `bscM` are parsed only as "last resort".

The order of the patterns in `bscM` matters, too. First, the parser to the very left is applied. If the parsing is successful, the parsed value is returned. Otherwise, the parser on the right-hand side is applied. Therefore, if we applied `var` before `generalFun` (such as `var | generalFun`), the message `foo(a)` would be parsed as the variable `foo` even though it is clearly a function application, eventually leading to a failure since the rest of the string would make no sense to the parser.

6.3 Rewriter

The rewriting stage is more interesting since it interprets the meaning of an A&B protocol specification. It takes a parse tree as input, analyzes it and then converts it into the intermediate representation format. The rewriting stage is implemented in the package `Rewriter`, the main function being `translateToIR :: Parser.ParseTree.ParseTree -> Rewriter.IR.Protocol` in package `Rewriter.Rewriter`.

6.3.1 Representing State

We have seen that a protocol can be described as a collection of roles that can be analyzed more or less independently when we were discussing the semantics of Alice&Bob notation. For this reason, the rewriting step handles one role after another. At the core of a role lies the knowledge that the principal in it has at any point in time which means that we have to carry it with us during the complete rewriting step. The nicest way to handle this is by using a state monad.

The state does not only contain the current knowledge but also other relevant information such as the name of the role and the parse tree. The state is defined in the package `Rewriter.State`:

```
type RoleState a
    = Control.Monad.State ( RoleName,
                          , Parser.ParseTree.ParseTree
                          , Rewriter.IR.Knowledge
                          , [Basic.Identifier],
                          , Data.Set.Set Basic.Message.Message) a
```

The last two elements of the tuple require some explanation. Recall the definition of `Parser.Message.Message` (we introduced it in Section 4.1 on page 37) where the type constructor for a message under ghost abstraction is defined as `Gamma Identifier Message`. The identifier provides a representative for the message such as `alpha` or `beta`. We want to make sure that all ghost messages get a unique representative in the IR such that we can directly use them when we generate the code for the output.

We do not know in advance how many representatives we need. But we can use the lazy evaluation property of Haskell and work with an infinite list of unique identifiers. For this purpose, the function `Rewriter.Functions.symbolStore` returns the infinite list with the elements `"alpha"`, `"beta"`, ..., `"omega"`, `"alpha1"`, `"beta1"`, ... The fourth element of the `RoleState` tuple is the list of all the representatives that are not yet being used as representatives. Whenever a new name is needed, we can simply remove the head of the list.

During rewriting, we have to remember which representative was assigned to which ghost message. This is what the fifth element of the tuple is for; whenever a new ghost message is produced, it is assigned a unique representative from the infinite list and the corresponding ghost message (including its representative) is added to the set. When we have to generate another ghost message (for example when computing the constructive form of a message), we should first check if this message already has a ghost and only use a new representative if this is not the case. In doing so, we make sure that the same ghost message is alway represented by the same representative.

The package `Rewriter.State` does not only provide the definition of `RoleState` but also provides some functions for working with it which are used frequently in the rewriter.

6.3.2 From Parse Tree to Intermediate Representation Format

Based on the `RoleState` monad introduced above and our analysis of Alice&Bob notation in Section 2, we can now approach the rewriting step itself.

For some parts of a protocol specification, such as the declaration of functions or security goals, the rewriting from parse tree to IR is straightforward since there is no need to perform any analysis. What is a little bit more involved is analyzing the actions that a principal in a role has to take. We have indicated above that we translate one role after another. It is easy to extract the names of all the roles that appear in a protocol by analyzing the parse tree. For each role, we then construct the initial knowledge and build the initial `RoleState`. Further analysis then happens under this `RoleState` until finally the `Rewriter.IR.Role` is returned.

It is here where it comes in handy that we have defined the constructive form cf_χ that tells us how a message can be constructed from knowledge and the algorithm $analyzeOnce_\chi$ that tells us what sub-messages can be extracted from a message using the analysis rules. They are implemented in Haskell

by the functions `Rewriter.Functions.cf` and `Rewriter.Functions.analyzeOnce` that both operate under the `RoleState` monad since both are dependent on the current knowledge and may need to generate new ghost messages (for which we need the infinite list of unused representatives and the list of representatives that are already being used).

Calculating the constructive form of multiplicative and exponential terms is based on division and left and right reduction like we have explained in Section 2.5. The corresponding functions are implemented in the package `Rewriter.MessageModel` and are called `divide`, `reducel` and `reducer`. In order to check if a message is divisible, left or right reducible by another message, the functions `divisible`, `reduciblel` and `reducibler` are provided. These functions are not under the `RoleState` monad since they are not dependent on state. The functions $constrMultTerm$ and $constrExpTerm$ are implemented by `Rewriter.Functions.cfMulTerm` and `Rewriter.Functions.cfExpTerm`, respectively.

We have explained that new messages are analyzed by the algorithm *addKnowledge* in Section 2.6. The corresponding function is `Rewriter.Rewriter.addKnowledge`. It is implemented in a similar fashion as the algorithm $addKnowledge_\chi$:

```
addKnowledge :: Message -> RoleState ()
addKnowledge m =
    do ana1 <- Rewriter.State.analyzeOnce m
       Rewriter.State.addToChi ana1
       Rewriter.Rewriter.analyzeKnowledge
       Rewriter.Rewriter.removeSynth
```

The new message is first analyzed once by `analyzeOnce` before it is added to the knowledge (by function `addToChi`). Function `analyzeKnowledge` implements the semantics of the `while` loop of $addKnowledge_\chi$. The function `removeSynth` finally restores the basic set property. Consult the code for the corresponding function definitions. It is easy to see that `addKnowledge` indeed implements $addKnowledge_\chi$.

Some of the functions that are under the `RoleState` monad have an additional stateless version in package `Rewriter.FunctionsStateless`, for example there is the function `Rewriter.FunctionsStateless.cf`. The most obvious difference here is that we have to give the knowledge explicitly as a parameter. But what is more important is the fact that functions without state cannot generate new ghost symbols since they do not have access to the list of the remaining representatives. Consequently, `Rewriter.FunctionsStateless.cf` can only be applied if the ghost messages that are needed are already in the knowledge. This is the case *after* the IR has been constructed. When generating TAMARIN code, there are situations where we need to compute the constructive form of a message. However, since the complete IR already exists at that time there is no need to generate new ghost symbols and consequently we can use the stateless version in this case. In other words, the stateful functions are used during the construction of the IR, the stateless functions afterwards.

6.4 Well-formedness Checks

The intermediate representation format is perfectly suited for ensuring that the protocol is well-formed like we required in Section 3.5. The translator does not produce any code unless all well-formedness checks are passed. The checks are implemented in package `Checker` where the function `doAllChecks :: Rewriter.IR.Protocol -> Checker.CheckResult.CheckResult` in package `Checker.Checker` performs all checks.

In order to inform about failed well-formedness checks in a sensible way, we introduce the following data structure in the package `Checker.CheckResult`:

```
data CheckResult = Success | Failure [Fail]
```

A check is either successful, or it fails. If it fails, then the error is specified by an instance of `Fail` (it will become clear in a moment why we use a list of `Fail` in `Failure [Fail]` instead of a single `Fail`):

```
data Fail
    = DuplicateLabels (Label)
    | NotExecutable IR.Action
    | EqualSenderReceiver Label
    | NameCollision (Identifier, Label)
    | DuplicateFunctionDec Identifier
```

```
    | NegativeArity Identifier
    | WrongArity (Identifier, Label)
    | UnknownFunction (Identifier, Label)
    | UsedFunAsVar (Identifier, Label)
    | InconsistentCases (Label, Identifier, Label, Identifier)
    | GoalNotExecutable (Label, Message, RoleName)
    | RoleInGoalWithNoAction (Label, RoleName)
```

`DuplicateLabels` expresses that a label occurs more than once, `NotExecutable` that a specific action is not executable, `EqualSenderReceiver` that sender and receiver are equal in an action, `NameCollision` that there is a fresh variable declaration that collides with a variable that is declared elsewhere, `DuplicateFunctionDec` that there are multiple declarations of functions with the same name, `NegativeArity` that a function was declared with negative arity, `WrongArity` that a function was used with an arity that differs from the arity with which the function was declared, `UnknownFunction` that a function was used that was not previously declared, `UsedFunAsVar` that a function was used as if it were a variable, that is, without parentheses, `InconsistentCases` that there were identifiers that are equal in a case-insensitive sense but not equal in a case-sensitive sense, `GoalNotExecutable` that there is a security goal where not all roles can construct all messages of the goal, `RoleInGoalWithNoAction` that a role is mentioned in a goal even though this role does not have any actions in the protocol. For details and the meaning of the arguments, see the comments of the definition of `Checker.CheckResult.Fail`.

`CheckResult` is a monoid since the results of checks can be appended to each other. Two successful checks result in a successful check; if either of the check fails, the result is a list of the errors. The instance declaration of the monoid therefore looks as follows:

```
instance Monoid CheckResult where
    mempty = Success
    mappend Success cr = cr
    mappend cr Success = cr
    mappend (Failure f1) (Failure f2) = Failure (f1 ++ f2)
```

If more than one check fails, then the corresponding instances of `Fail` are accumulated in the list of `Failure [Fail]`. This is why we use a list there and not a single element. This enables us to print readable error messages.

Some checks only make sense if the previous checks have succeeded. For example, checking if all functions are applied with the correct number of arguments has no meaning if the function was declared twice with different arity or was not declared at all. Therefore, the checks are assigned to levels. Only if all of the checks of one level have passed is the next higher level of checks performed as well. This helps prevent error messages that are caused by other errors that should be resolved first. The levels are as follows:

- Level 1: `DuplicateFunctionDec`, `NegativeArity`, `DuplicateLabels`, `InconsistentCases`.
- Level 2: `EqualSenderReceiver`.
- Level 3: `RoleInGoalWithNoAction`.
- Level 4: `GoalsNotExecutable`, `WrongArity`, `UsedFunAsVar`, `UnknownFunction`, `NameCollision`.

Recall the protocol from Figure 6 on page 35 that is not well-formed. When calling `Checker.Checker.doAllChecks` on the IR of the protocol, the following errors are output:

```
There are multiple declarations for a function with name 'fun'.

The function with name 'fun' is declared with negative arity.

Label 'action' is declared multiple times.
```

In this example, there are failures with the types `DuplicateFunctionDec`, `NevativeArity` and `DuplicateLabels`, all of which are from level 1.

6.5 Generation of Tamarin Code

We have discussed in Section 5 how a protocol in its intermediate representation can be translated into Tamarin code. We therefore only talk about some implementation-specific aspects here. The code generator is implemented in the package `Translator` where we use the state monad again for storing relevant information during the code generation process. The state `TranslationState` is defined as follows:

```
import Parser.Basic
import Rewriter.IR
import Control.Monad.State

type OutputState = (String, Int, Int)

type RoleState = (RoleName, [Action], [Identifier], [Identifier])

type TranslationState a = State (Protocol, RoleState, OutputState) a
```

We can see that `TranslationState` is defined as a triple. The first element is the IR of the complete protocol that is being translated to Tamarin and allows access to any information of the protocol that is needed. The second element of the tuple is of type `RoleState`, the third of type `OutputState`. Let us discuss those last two elements in a little more detail.

Recall that when generating Tamarin code, we first output the setup. This involves the message model and the rewriting rules for setting up keys and the initial knowledge. After that, the rewriting rules that describe the roles are output where we can treat the different roles independently. Therefore, we first generate code for one role, then for another until all roles have been translated. Generating code for a role means to generate code for its `Receive` and `Send` actions. An action depends on its preceding action and therefore they have to be treated in the order in which they are executed by a principal. The second element of the `RoleState` tuple therefore represents the remaining actions of the role for which code is currently being generated.

For the second and third element of `RoleState`, recall that variables can be prefixed with a $\sim$ to indicate that they represent a fresh value or a \$ to indicate that they contain a public value. The third element of `RoleState` is a list of the identifiers that should be prefixed with a $\sim$, the fourth a list of those identifiers that should be prefixed with a \$ in the rewriting rule that expresses the action for which code is currently being generated.

In order to achieve nicely formatted code, we have implemented a few functions for generating Tamarin code in package `Translator.Printer`. These functions make use of the information in the tuple `OutputState`. Namely, the first element is the output that has been written so far. The second parameter stores the current indentation depth and the third the current cursor position. Whenever Tamarin code is appended, the cursor position is updated and if necessary, a newline character is inserted. The value `Translator.Printer.maxOffset` determines the maximum length of a line. Whenever this length would be exceeded, the printer functions try to insert a newline character in order to prevent the line from becoming too long. The indentation level determines how many blank spaces are inserted on the new line. Restricting our output to a reasonable line length and using indentation significantly improves the readability of the generated Tamarin code.

Based on the state defined above, we have implemented the functions that generate code from the IR in the packages `Translator.Translator` and `Translator.Auxiliary`.

7 Case Study

We have now completely specified the A&B protocol specification language, the intermediate representation format and explained how the translation works and how it is implemented. Throughout this thesis, we used the CR protocol to point out some of the problems that occur in the context of Alice&Bob protocol specifications and the ideas behind A&B. In this section, we demonstrate the translator on a somewhat more involved protocol, namely the ASW protocol that was given in Figure 5 on page 30. In particular, we show how the TAMARIN prover can be used to verify or disprove the security goals that are declared in the A&B specification.

Recall that the translator first converts A&B input into the intermediate representation from which it then produces TAMARIN code. We therefore first take a short look at the intermediate representation. However, the main focus will be on the TAMARIN code that is produced. Finally, we show how the TAMARIN prover can be applied to the output to verify (or disprove) the security goals.

7.1 Intermediate Representation

The translator provides the option `-v` that produces verbose output, that is, a pretty-printed version of the A&B protocol specification, the IR and the produced TAMARIN code are output. Therefore, if we want to get a human-readable representation of the IR of the ASW protocol, we can call

```
>> ./anb asw.anb -v
```

and then read it off the screen (see Appendix D on page 78 for a short manual for the translator). The IR is given below in a nicely formatted way. The actual representation in Haskell code is quite long, therefore we resort to shorter and more readable way of denoting messages by using the same syntax for them as in Section 2. To make sure that we do not lose the representative identifiers of ghost messages, we adapt our syntax a little. For example, `Gamma "alpha" (Hash (Var "a"))` is represented as γ[alpha]($\mathbf{h}(a)$). The IR then looks as follows:

```
Protocol "ASW" []
  [ Role "A"
        [ Prepare [A, B, m, sk(A), pk(A), pk(B)]
        , Send True "asw1" "B" ["n_1"] {⟨pk(A) . pk(B) . m . h(n1)⟩}a_sk(A)
              [A, B, m, n1, sk(A), pk(A), pk(B)]
        , Receive "asw2" "B" γ[beta]({{⟨pk(A) . pk(B) . m . h(n1)⟩}a_sk(A) . h(n2)⟩}a_sk(B))
              [A, B, m, n1, sk(A), pk(A), pk(B), γ[alpha](h(n2)),
               γ[beta]({{⟨pk(A) . pk(B) . m . h(n1)⟩}a_sk(A) . h(n2)⟩}a_sk(B))]
        , Send True "asw3" "B" [] n1
              [A, B, m, n1, sk(A), pk(A), pk(B), γ[alpha](h(n2)),
               γ[beta]({{⟨pk(A) . pk(B) . m . h(n1)⟩}a_sk(A) . h(n2)⟩}a_sk(B)]
        , Receive "asw4" "B" n2
              [A, B, m, n1, n2, sk(A), pk(A), pk(B),
               γ[beta]({{⟨pk(A) . pk(B) . m . h(n1)⟩}a_sk(A) . h(n2)⟩}a_sk(B))]
        ]
  , Role "B"
        [ Prepare [A, B, sk(B), pk(A), pk(B)]
        , Receive "asw1" "A"  γ[beta]({⟨pk(A) . pk(B) . m . h(n1)⟩}a_sk(A))
              [A, B, m, sk(B), pk(A), pk(B), (γ[alpha](h(n1)),
               γ[beta]({⟨pk(A) . pk(B) . m . h(n1)⟩}a_sk(A))]
        , Send True "asw2" "A" ["n_2"]
               {γ[beta]({⟨pk(A) . pk(B) . m . h(n1)⟩}a_sk(A)) . h(n2)}a_sk(B),
              [A, B, m, n2, sk(B), pk(A), pk(B),
               γ[beta]({⟨pk(A) . pk(B) . m . h(n1)⟩}a_sk(A)), γ[alpha](h(n1))]
        , Receive "asw3" "A" n1
              [A, B, m, n1, n2, sk(B), pk(A), pk(B),
               γ[beta]({⟨pk(A) . pk(B) . m . h(n1)⟩}a_sk(A))]
        , Send True "asw4" "A" [] n2
              [A, B, m, n1, n2, sk(B), pk(A), pk(B),
               γ[beta]({⟨pk(A) . pk(B) . m . h(n1)⟩}a_sk(A)]
```

```
        ]
  ]
  [ Secret "secrecy" m ["A", "B"]
  , WeakAuth "weakA" "A" "B" [m]
  , WeakAuth "weakB" "B" "A" [m]
  , StrongAuth "strongA" "A" "B" [m]
  , StrongAuth "strongB" "B" "A" [m]
  ]
```

The overall structure of the IR can be seen very well from this example, in particular the lists with the roles of the protocols and the security goals. One can easily see how the knowledge of the roles evolves over time and also that messages get removed from the knowledge sometimes. If we take a look at the role of `A` for example, we see that she has $\mathbf{h}(n_1)$ in her knowledge after the `Receive` action `asw2` and that this message is not kept in the knowledge any longer once she has received n_2 in action `asw4`.

7.2 Tamarin Output

The translation from the IR to Tamarin is more interesting. The following Tamarin code is generated by the translator for the ASW protocol (we have re-formatted it slightly to save space):

```
theory ASW
begin

functions: pk/1, sk/1, aenc/2, adec/2
builtins: hashing
equations: adec(aenc(x.1, sk(x.2)), pk(x.2)) = x.1,
           adec(aenc(x.1, pk(x.2)), sk(x.2)) = x.1

rule Asymmetric_key_setup:
    [ Fr(~f) ] --> [ !Sk($A, sk(~f)), !Pk($A, pk(~f)) ]

rule Publish_public_keys:
    [ !Pk(A, pkA) ] --> [ Out(pkA) ]

rule Init_Knowledge:
        [ !Pk($A, pk(k_A)), !Pk($B, pk(k_B)),
          !Sk($A, sk(k_A)), !Sk($B, sk(k_B)), Fr(~m) ]
        --[  ]->
        [ St_init_A($A, $B, ~m, sk(k_A), pk(k_A), pk(k_B)),
          St_init_B($A, $B, sk(k_B), pk(k_A), pk(k_B)) ]

// ROLE A
rule asw1_A:
        [ St_init_A(A, B, m, sk(k_A), pk(k_A), pk(k_B)),
          Fr(~n_1) ]
        --[ Running_weakB(m), Running_strongB(m) ]->
        [ Out(aenc{<pk(k_A), pk(k_B), m, h(~n_1)>}sk(k_A)),
          St_asw1_A(A, B, m, ~n_1, sk(k_A), pk(k_A), pk(k_B)) ]

rule asw2_A:
    let
        beta = aenc{<aenc{<pk(k_A), pk(k_B), m, h(n_1)>}sk(k_A), alpha>}sk(k_B)
    in
        [ St_asw1_A(A, B, m, n_1, sk(k_A), pk(k_A), pk(k_B)),
          In(beta) ]
        --[  ]->
        [ St_asw2_A(A, B, m, n_1, sk(k_A), pk(k_A), pk(k_B), beta, alpha) ]

rule asw3_A:
        [ St_asw2_A(A, B, m, n_1, sk(k_A), pk(k_A), pk(k_B), beta, alpha) ]
        --[  ]->
```

```
        [ Out(n_1),
          St_asw3_A(A, B, m, n_1, sk(k_A), pk(k_A), pk(k_B), beta, alpha) ]

rule asw4_A:
    let
        beta = aenc{<aenc{<pk(k_A), pk(k_B), m, h(n_1)>}sk(k_A), h(n_2)>}sk(k_B)
        alpha = h(n_2)
    in
        [ St_asw3_A(A, B, m, n_1, sk(k_A), pk(k_A), pk(k_B), beta, alpha),
          In(n_2) ]
        --[ Secret_secrecy_A(m), Commit_weakA(m), Commit_strongA(m) ]->
        [ St_asw4_A(A, B, m, n_1, n_2, sk(k_A), pk(k_A), pk(k_B), beta) ]

// ROLE B
rule asw1_B:
    let
        beta = aenc{<pk(k_A), pk(k_B), m, alpha>}sk(k_A)
    in
        [ St_init_B(A, B, sk(k_B), pk(k_A), pk(k_B)),
          In(beta) ]
        --[ Running_weakA(m), Running_strongA(m) ]->
        [ St_asw1_B(A, B, m, sk(k_B), pk(k_A), pk(k_B), beta, alpha) ]

rule asw2_B:
        [ St_asw1_B(A, B, m, sk(k_B), pk(k_A), pk(k_B), beta, alpha),
          Fr(~n_2) ]
        --[  ]->
        [ Out(aenc{<beta, h(~n_2)>}sk(k_B)),
          St_asw2_B(A, B, m, ~n_2, sk(k_B), pk(k_A), pk(k_B), beta, alpha) ]

rule asw3_B:
    let
        beta = aenc{<pk(k_A), pk(k_B), m, h(n_1)>}sk(k_A)
        alpha = h(n_1)
    in
        [ St_asw2_B(A, B, m, n_2, sk(k_B), pk(k_A), pk(k_B), beta, alpha),
          In(n_1) ]
        --[  ]->
        [ St_asw3_B(A, B, m, n_1, n_2, sk(k_B), pk(k_A), pk(k_B), beta) ]

rule asw4_B:
        [ St_asw3_B(A, B, m, n_1, n_2, sk(k_B), pk(k_A), pk(k_B), beta) ]
        --[ Secret_secrecy_B(m), Commit_weakB(m), Commit_strongB(m) ]->
        [ Out(n_2),
          St_asw4_B(A, B, m, n_1, n_2, sk(k_B), pk(k_A), pk(k_B), beta) ]

lemma secrecy:
    " not( Ex msg #i1 #i2 #j .
             Secret_secrecy_A(msg) @ #i1 &
             Secret_secrecy_B(msg) @ #i2 &
             K(msg) @ #j )"

lemma weakA:
    " (All m1 #i . Commit_weakA(m1)@ #i
        ==>
        (Ex #j . Running_weakA(m1) @ #j & #j < #i) )"

lemma weakB:
    " (All m1 #i . Commit_weakB(m1)@ #i
        ==>
        (Ex #j . Running_weakB(m1) @ #j & #j < #i) )"
```

```
lemma strongA:
    " (All m1 #i . Commit_strongA(m1)@ #i
        ==>
        (Ex #j . Running_strongA(m1) @ #j & #j < #i) &
        (not (Ex #j . Commit_strongA(m1) @ #j & not (#i = #j))) )"

lemma strongB:
    " (All m1 #i . Commit_strongB(m1)@ #i
        ==>
        (Ex #j . Running_strongB(m1) @ #j & #j < #i) &
        (not (Ex #j . Commit_strongB(m1) @ #j & not (#i = #j))) )"
end
```

The structure of the file should be quite clear. At the top, we first declare our message model (note that we only included the theories that actually matter for the ASW protocol). Then follow the rules that generate public-secret key pairs, the rule that enables the adversary to get hold of the public keys and the rule that expresses the initial knowledge that the roles have. This is succeeded by the rules that express the actual communication that happens in the protocol and finally, there is a lemma for each of the security goals.

It can be seen very clearly that the order in which a principal executes his role is induced by the knowledge. For example, the principal in role `A` first obtains her initial knowledge from fact `St_init_A` and produces fact `St_asw1_A` as a conclusion in her first rule `asw1_A`. This fact is then consumed in rule `asw2_A`, where fact `St_asw2_A` is produced. This also illustrates that the labels that we required in the A&B input are essential for the readability of the TAMARIN code.

Rule `asw3_B` illustrates well how messages in the knowledge that become synthesizable are treated. The message `n_1` is not known to `B` until communication step `asw3` and therefore `h(n_1)` can neither be analyzed nor synthesized from other messages in the knowledge. Consequently, it is stored as a ghost message and assigned the representative `alpha`. Now that `n_1` is received, `alpha` can be removed from the knowledge and therefore, `alpha` is not contained in the conclusion fact `St_asw3_B`. At first glance, there is no reason for expressing that `alpha = h(n_1)` in the **`let`** block. However, we need to tell TAMARIN what shape the message `alpha` has because otherwise TAMARIN cannot perform the appropriate checks whether the two messages actually match. The same applies for `beta` that represents the complete encrypted message. We mention it in the **`let`** block even though it still cannot be synthesized from other messages. Note that this is not strictly necessary for `beta` since this check is covered by the check involving `alpha`, but listing both simplifies matters in the implementation.

The ASW protocol was declared with five security goals and we can see well how these are expressed in TAMARIN. For the secrecy goals, `A` and `B` make the statements `Secret_secrecy_A` and `Secret_secrecy_B`, respectively (recall that we refer to TAMARIN's action facts as statements). These statements are then used in lemma `secrecy`. For agreement goals, the agreement partner states that he possesses the message as soon as possible by a statement like `A` does in rule `asw1_A` with her statement `Running_weakB`. The principal in the other role, `B` in this case, claims agreement with the statement `Commit_weakB` in rule `asw4_B`. These statements are then used in lemma `weakB`. Note again that the label-based naming of the rules makes quite clear how the different rules, lemmas and statements are related.

7.3 Protocol Verification

Consult Appendix D for instructions on the compilation and the usage of the translator. When calling

```
>> ./anb asw.anb,
```

the verifier produces a file `asw.spthy` with the TAMARIN output. In a standard installation, the TAMARIN prover can then be invoked on it by calling

```
>> tamarin-prover asw.spthy --prove
```

The TAMARIN prover first outputs a pretty-printed version of the protocol specification. In particular, all the built-in rules are spelled out in full. Then, TAMARIN tries to verify the lemmas. Protocol verification is undecidable and consequently there are protocols that simply cannot be proved by the TAMARIN

prover. In such a case the verification does not terminate. For the ASW protocol, however, TAMARIN almost immediately returns:

```
summary of summaries:

analyzed: asw.spthy

  secrecy (all-traces): falsified - found trace (10 steps)
  weakA (all-traces): verified (6 steps)
  weakB (all-traces): verified (6 steps)
  strongA (all-traces): verified (14 steps)
  strongB (all-traces): falsified - found trace (10 steps)
```

We can see that the security goals `secrecy` and `strongB` are disproved by TAMARIN while there are no attacks for the other security goals. The TAMARIN prover provides a graphical user interface that visualizes the attacks that were found. The traces of the attacks for `secrecy` and `strongB` are very long and therefore we demonstrate this on on the following short and very simple example:

```
Protocol Asym:
Knowledge:
    B : pk(A);
Actions:
    [a1] A -> B (n) : aenc{n}sk(A);
Goals:
    [sec]  n secret of A, B;
end
```

The adversary knows all public keys and therefore it is obvious that the secrecy goal `sec` does not hold since the message **aenc**`{n}`**sk**`(A)` can be decrypted by him and TAMARIN consequently finds a trace that disproves lemma `sec`. The graphical mode of TAMARIN can be started by invoking

```
>> tamarin-prover interactive asym.spthy --prove
```

and then visiting `http://localhost:3001/` in a browser. The graphical representation of the attack that is found by TAMARIN for the `Asym` protocol is depicted in Figure 9. We will not provide the TAMARIN code here because it is possible to follow a trace only with the A&B protocol specification.

Let us briefly discuss the trace of the attack. The secrecy goal `sec` requires that `n` must not be known by the adversary whenever both `A` and `B` finish the execution of their roles (compare to Definition 3.1 on page 34). Recall that `A` and `B` make the statements `Secret_sec_A` and `Secret_sec_B` in TAMARIN in their last rules because they are both mentioned in the goal `sec` and note that both `A` and `B` perform their last (any only) action in message exchange step `a1`.

The first three rows of the graphical representation of the trace represent the setup of the initial knowledge of `A` and `B`. In the fourth row, `A` executes her first and only action, makes her statement `Secret_sec_A` and sends `aenc{n}sk(A)`. The message is then intercepted by the adversary (rule `irecv`). The rewriting chain on the right-hand side describes how the adversary first obtains the public key of `A` via the rule `Publish_public_keys` and then the derivation chain where the message `n` is extracted step by step from `aenc{n}sk(A)`, where the fact `!KU(~n)` has the meaning that `n` can be constructed by the adversary. Finally, the adversary forwards `aenc{n}sk(A)` to the network (rule `isend`) where `B` receives the message and makes his statement `Secret_sec_B`. Therefore, both the statements `Secret_sec_A(n)` and `Secret_sec_B(n)` are made while the adversary knows `n`. Consequently, the lemma `sec` is violated.

This example illustrates that the generated code is designed in such a way that the output of the TAMARIN prover should be understandable without looking at the TAMARIN file. However, it still makes sense to double check if the TAMARIN code really expresses the protocol that was intended.

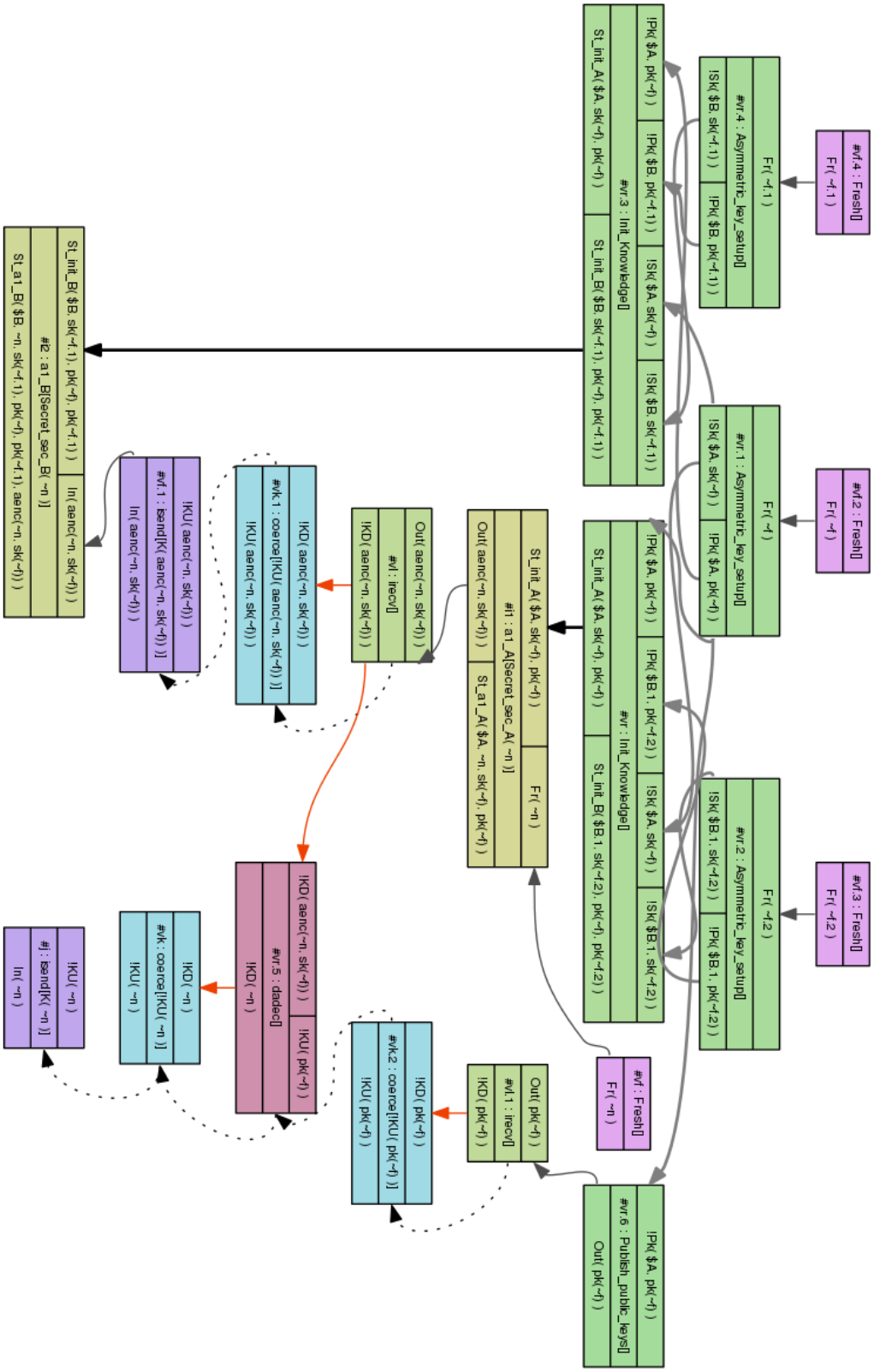

Figure 9: Graphical representation of the attack on the security goal `sec` of the `Asym` protocol.

8 Conclusion

Let us first briefly recapitulate what we have done in this thesis. We first discussed the challenges that have to be met when formalizing the semantics of Alice&Bob notation where we could resort at large to previous work. Based on our semantics of Alice&Bob notation, we then specified the syntax and the semantics of the A&B protocol specification language. Finally, we implemented the translator where we argued the correctness of the most important pieces of code.

The examples that we gave throughout the thesis show that the A&B language is much more readable and shorter than the input languages of common protocol verification tools such as Tamarin while it still allows to express a large class of protocols. The case study at the end of the thesis demonstrated that the generated Tamarin code is well-structured and readable.

However, there are also some limitations to the A&B language. There is no way to specify explicitly what principals should do if they suspect that there might have been adversary involvement; principals simply abort protocol execution in such a case. What is more, the rigid structure of message exchange steps with exactly one receiver limits the A&B language to a certain class of protocols. There is also no immediate way to specify *stateful* protocols. These restrictions are acceptable though since we designed the A&B language to be slim and expressive and as close as possible to Alice&Bob notation. An aspect of A&B that leaves room for improvement, on the other hand, is its message model, in particular with respect to Diffie-Hellman multiplication and exponentiation. Only the most important properties were included in equational theory M. For example, the message $a^c \odot b^c$ is not equal to $(a \odot b)^c$ in our current message model.

At the moment, Tamarin is the only target language of the translator. The most obvious extension is therefore adding support for other output languages where the structure of the intermediate representation format is a good basis. However, the fact that A&B is independent of a specific verification tool also has its downsides since this prevents us from taking the properties of specific provers into account. Sometimes, Tamarin requires the specification of typing lemmas since otherwise, the verification step does not terminate, something that the translator cannot do for us. This makes clear that the A&B protocol specification language must not be understood as a direct input language for automated verifiers but rather as a tool that helps on the path to the final (Tamarin) specification. The A&B language gives a head-start by allowing one to express a protocol in a simple and readable way where the translator then produces a well-structured basis code from which work can continue.

In this sense, the main contribution of this thesis is the specification of the syntax and the semantics of the A&B language as well as a translator from it to Tamarin. A&B allows one to specify a protocol in a highly readable way while the translator produces valid Tamarin code that can then be used for further refinement. In summary, we can conclude that the A&B protocol specification language and its translator to Tamarin can be a valuable and work-saving tool when it comes to specifying a protocol in Tamarin.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Converting Alice&Bob Protocol Specifications to Tamarin

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Keller	Michel

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.

References

[1] N. Asokan, Victor Shoup, and Michael Waidner. Asynchronous Protocols for Optimistic Fair Exchange. In *IEEE Symposium on Security and Privacy*, pages 86–99. IEEE Computer Society, 1998.

[2] Bruno Blanchet and Ben Smyth. *ProVerif 1.85: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial*, 2011.

[3] Carlos Caleiro, Luca Viganò, and David A. Basin. On the Semantics of Alice&Bob Specifications of Security Protocols. *Theor. Comput. Sci.*, 367(1-2):88–122, 2006.

[4] Cas J.F. Cremers. Unbounded Verification, Falsification, and Characterization of Security Protocols by Pattern Refinement. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 119–128, New York, NY, USA, 2008. ACM.

[5] Santiago Escobar, Catherine Meadows, and José Meseguer. Maude-NPA: Cryptographic Protocol Analysis Modulo Equational Properties. In *FOSAD*, pages 1–50, 2007.

[6] Gavin Lowe. A Hierarchy of Authentication Specifications. In *Proceedings of the 10th IEEE Workshop on Computer Security Foundations*, CSFW '97, pages 31–, Washington, DC, USA, 1997. IEEE Computer Society.

[7] Sreekanth Malladi and Bruno Blanchet. Online Demo for ProVerif. `http://proverif.rocq.inria.fr/`, 2014.

[8] Simon Meier. GitHub Repository of scyther-proof Project. `https://github.com/meiersi/scyther-proof`.

[9] Simon Meier, Cas J. F. Cremers, and David A. Basin. Strong Invariants for the Efficient Construction of Machine-Checked Protocol Security Proofs. In *CSF*, pages 231–245. IEEE Computer Society, 2010.

[10] Simon Meier, Benedikt Schmidt, Cas J. F. Cremers, and Cedric Staub. Tamarin Prover. `http://www.infsec.ethz.ch/research/software/tamarin`, May 2014.

[11] Sebastian Mödersheim. Algebraic Properties in Alice and Bob Notation (extended version). Technical Report RZ3709, IBM Zurich Research Lab, 2008.

[12] Sebastian Mödersheim. Algebraic Properties in Alice and Bob Notation. In *ARES*, pages 433–440. IEEE Computer Society, 2009.

[13] Benedikt Schmidt, Simon Meier, Cas J. F. Cremers, and David A. Basin. Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties. In Stephen Chong, editor, *CSF*, pages 78–94. IEEE, 2012.

A Proofs

Proposition 2.7. *Let χ_1 and χ_2 be sets of messages and let χ_1' and χ_2' be basic sets with $synth^*(\chi_1) =_M synth^*(\chi_1')$ and $synth^*(\chi_2) =_M synth^*(\chi_2')$. Then $synth^*(\chi_1) =_M synth^*(\chi_2)$ if and only if $\chi_1' =_M \chi_2'$.*

Proof. Let χ_1 and χ_2 be sets of messages and let χ_1' and χ_2' be basic sets with $synth^*(\chi_1) =_M synth^*(\chi_1')$ and $synth^*(\chi_2) =_M synth^*(\chi_2')$. We prove the two directions separately.

- $synth^*(\chi_1) =_M synth^*(\chi_2) \Rightarrow \chi_1' =_M \chi_2'$.
  The proof is by contradiction. Assume that $synth^*(\chi_1) =_M synth^*(\chi_2)$ but $\chi_1' \neq_M \chi_2'$. From $\chi_1' \neq_M \chi_2'$ we can conclude that there is a message m with $m \in_S \chi_1'$ and $m \notin_M \chi_2'$ (without loss of generality; otherwise simply exchange χ_1' and χ_2'). Since χ_1' and χ_2' are basic sets, we have $m \notin_M synth^*(\chi_1' \backslash \{m\})$ since otherwise χ_1' would not be a basic set. Furthermore, we have $m \in_M synth^*(\chi_2') =_M synth^*(\chi_2' \backslash \{m\})$ since otherwise $synth^*(\chi_1') \neq_M synth^*(\chi_2')$ and therefore $synth^*(\chi_1) =_M synth^*(\chi_1') \neq_M synth^*(\chi_2') =_M synth^*(\chi_2)$, contradicting our assumption.

  Since $m \in_M synth^*(\chi_2')$ and $m \notin_M \chi_2'$, we know that there must be messages $m_1, \ldots, m_i$ in χ_2' that are not equal to m from which m can be synthesized. At least one of the messages $m_1, \ldots, m_i$ cannot be in $synth^*(\chi_1')$ since otherwise m would be synthesizable from χ_1', i.e., $m \in synth^*(\chi_1' \backslash \{m\})$. This however implies $synth^*(\chi_1') \neq_M synth^*(\chi_2')$ and hence $synth^*(\chi_1) =_M synth^*(\chi_1') \neq_M synth^*(\chi_2') =_M synth^*(\chi_2)$, a contradiction to our assumption.

- $\chi_1' =_M \chi_2' \Rightarrow synth^*(\chi_1) =_M synth^*(\chi_2)$.
  From $\chi_1' =_M \chi_2'$ we obviously have $synth^*(\chi_1') =_M synth^*(\chi_2')$ and with $synth^*(\chi_1) =_M synth^*(\chi_1')$ and $synth^*(\chi_2) =_M synth^*(\chi_2')$ we have $synth^*(\chi_1) =_M synth^*(\chi_1') =_M synth^*(\chi_2') = synth^*(\chi_2)$.

□

Lemma 2.11. *Let m be an arbitrary message. Then $canonical(m)$ is in canonical form and $canonical(m) =_M m$.*

Proof Sketch. First of all, note that $canonical(m)$ is defined for all messages. The proof is by structural induction on the message m. Let us do a case distinction on the shape of m.

- m is atomic.
  In this case, $canonical(m) =_S m$. Consequently, $canonical(m) =_M m$. Furthermore, m is in canonical form by the definition of the canonical form.

- $m =_S \langle m_1 \,.\, m_2 \rangle$.

  - Case (i). Message $m_1 =_S \langle m_1' \,.\, m_1'' \rangle$, i.e., m_1 is a concatenation message itself.
    In this case, we have $canonical(\langle m_1 \,.\, m_2 \rangle) =_S canonical(\langle m_1' \,.\, \langle m_1'' \,.\, m_2 \rangle \rangle)$, i.e., the shape of the concatenation is rewritten into a right-associated form as required by the definition of the canonical form and *canonical* is applied recursively. Eventually, a situation will be reached where the left-hand side of the concatenation is not a concatenation itself, i.e., we are in case (ii) where the top-level concatenation is in its right-associated form.
  - Case (ii). Message m_1 is not a concatenation itself.
    In this case, we can apply our induction hypothesis and conclude that $canonical(m_1)$ and $canonical(m_2)$ are both in canonical form and $canonical(m_1) =_M m_1$ as well as $canonical(m_2) =_M m_2$. Consequently, $canonical(m)$ is in canonical form and from property (1) on page 11, we can conclude that $canonical(m) =_M m$.

- $m =_S \{m'\}_{k'}^{\mathbf{a}}$.

  - Case (i). $m' =_S \{m''\}_{k''}^{\mathbf{a}}$ for some messages m'' and k'' and $(k')^{-1} =_M k''$.
    In this case, we have $canonical(\{m'\}_{k'}^{\mathbf{a}}) =_S canonical(m'')$. By our induction hypothesis, $canonical(m'')$ is in canonical form and $canonical(m'') =_M m''$. From property (3) on page 11, we can conclude that $\{\{m''\}_{k''}^{\mathbf{a}}\}_{k'}^{\mathbf{a}}) =_M m''$ and consequently that $canonical(m) =_M canonical(m'') =_M m'' =_M m$.

- Case (ii). Whenever case (i) does not apply.
  In this case, we have $canonical(\{m'\}^{\mathbf{a}}_{k'}) =_S \{canonical(m')\}^{\mathbf{a}}_{canonical(k')}$. By our induction hypothesis, $canonical(m')$ and $canonical(k')$ are both in canonical form and from the definition of the canonical form we can see that $canonical(m)$ is in canonical form. From our induction hypothesis, we can also derive that $canonical(m') =_M m'$ as well as $canonical(k') =_M k'$ and consequently $canonical(m) =_M m$.

- $m =_S \{m'\}^{\mathbf{s}}_{k'}$.
  - Case (i). $m' =_S \{m''\}^{\mathbf{s}}_{k''}$ for some messages m'' and k'' and $k' =_M k''$.
    In this case, we have $canonical(\{m'\}^{\mathbf{s}}_{k'}) =_S canonical(m'')$. By our induction hypothesis, $canonical(m'')$ is in canonical form and $canonical(m'') =_M m''$. From property (2) on page 11, we can conclude that $\{\{m''\}^{\mathbf{s}}_{k''}\}^{\mathbf{s}}_{k'} =_M m''$ and consequently that $canonical(m) =_M canonical(m'') =_M m'' =_M m$.
  - Case (ii). Whenever case (i) does not apply.
    In this case, we have $canonical(\{m'\}^{\mathbf{s}}_{k'}) =_S \{canonical(m')\}^{\mathbf{s}}_{canonical(k')}$. By our induction hypothesis, $canonical(m')$ and $canonical(k')$ are both in canonical form and from the definition of the canonical form we can see that $canonical(m)$ is in canonical form. From our induction hypothesis, we can also derive that $canonical(m') =_M m'$ as well as $canonical(k') =_M k'$ and consequently $canonical(m) =_M m$.

- $m =_S \mathbf{h}(m')$.
  In this case, we have $canonical(\mathbf{h}(m')) =_S \mathbf{h}(canonical(m'))$. By our induction hypothesis we have $canonical(m') =_M m'$ and that $canonical(m')$ is in canonical form. From this and the definition of the canonical form, we can conclude that $canonical(m) =_M m$ and that $canonical(m)$ is in canonical form.

- $m =_S foo(m_1, \ldots, m_i)$.
  In this case, we have $canonical(foo(m_1, \ldots, m_i)) =_S foo(canonical(m_1), \ldots, canonical(m_i))$. By our induction hypothesis we have $canonical(m_k)$ is in canonical form and $canonical(m_k) =_M m_k$ for all $k \in \{1, \ldots, i\}$. From this and the definition of the canonical form, we can conclude that $canonical(m) =_M m$ and that $canonical(m)$ is in canonical form.

- m is a multiplication.
  - Case (i). $m =_S \odot\{\!|m'|\!\}$.
    In this case, we have $canonical(\odot\{\!|m'|\!\}) =_S canonical(m')$. By our induction hypothesis we have $canonical(m') =_M m'$ and with $\odot\{\!|m'|\!\} =_M m'$, we get that $canonical(m) =_M canonical(m') =_M m' =_M m$. Also by our induction hypothesis, we know that $canonical(m')$ is in canonical form and hence $canonical(m)$ is in canonical form.
  - Case (ii). $m =_S \odot\{\!|m_1, \ldots, m_{j-1}, m_j, m_{j+1}, \ldots, m_i|\!\}$ for $i \geq 2$ and $m_j =_M 1$.

    In this case, we have $canonical(m) =_S canonical(\odot\{\!|m_1, \ldots, m_{j-1}, m_{j+1}, \ldots, m_i|\!\})$. From our induction hypothesis, we know that $canonical(\odot\{\!|m_1, \ldots, m_{j-1}, m_{j+1}, \ldots, m_i|\!\})$ is in canonical form and hence $canonical(m)$ is in canonical form. From properties (5), (6), and (8) on page 11, we get $m = \odot\{\!|m_1, \ldots, m_{j-1}, m_j, m_{j+1}, \ldots, m_i|\!\} =_M \odot\{\!|m_1, \ldots, m_{j-1}, m_{j+1}, \ldots, m_i|\!\}$ and from our induction hypothesis we also know that $canonical(\odot\{\!|m_1, \ldots, , m_{j-1}, m_{j+1}, \ldots, m_i|\!\}) =_M \odot\{\!|m_1, \ldots, m_{j-1}, m_{j+1}, \ldots, m_i|\!\}$ and consequently $canonical(m) =_M m$.
  - Case (iii). $m =_S \odot\{\!|m_1, \ldots, m_{j-1}, m_j, m_{j+1}, \ldots, m_i|\!\}$ and $m_j =_S \odot\{\!|m^1_j, \ldots, m^k_j|\!\}$ and $i \geq 2$.
    In this case we have $canonical(m) =_S canonical(\odot\{\!|m_1, \ldots, m_{j-1}, m^1_j, \ldots, m^k_j, m_{j+1}, \ldots, m_i|\!\})$. This is only a different representation of multiplication on which *canonical* is recursively called. Since the depth of multiplication is gradually reduced, a state will eventually be reached where we are in state (i), (ii) or (iv) where none of $m_1, \ldots, m_i$ are themselves multiplicative messages.
  - Case (iv). Whenever none of the above applies.
    In this case $canonical(\odot\{\!|m_1, \ldots, m_i|\!\}) =_S \odot\{\!|canonical(m_1), \ldots, canonical(m_i)|\!\}$. From our

induction hypothesis, we can conclude that $\mathit{canonical}(m_k) =_M m_k$ and that $\mathit{canonical}(m_k)$ is in canonical form for all $k \in \{1, \ldots, i\}$. We consequently get that $\mathit{canonical}(m) =_M m$ and with the definition of the canonical form we get that $\mathit{canonical}(m)$ is in canonical form.

- $m =_S m_1^{m_2}$.
  - Case (i). $m_2 =_M 1$.
    In this case, we have $\mathit{canonical}(m) =_S \mathit{canonical}(m_1)$. From our induction hypothesis, we get that $\mathit{canonical}(m_1) =_M m_1$ and that $\mathit{canonical}(m_1)$ is in canonical form. Furthermore, we get from property (9) on page 11 that $m =_M m_1$ and consequently $\mathit{canonical}(m) =_M \mathit{canonical}(m_1) =_M m_1 =_M m$ and that $\mathit{canonical}(m)$ is in canonical form.
  - Case (ii). $m_1 =_M 1$ and $m_2 \neq_M 1$.
    In this case, we have $\mathit{canonical}(m) =_S 1$. We know from property (10) on page 11 that $m = 1$, hence $\mathit{canonical}(m) =_M \mathit{canonical}(1) =_M 1$ which is obviously in canonical form.
  - Case (iii). $m_1 =_S (m_1')^{m_1''}$.
    In this case, we have $\mathit{canonical}(m_1^{m_2}) =_S \mathit{canonical}((m_1')^{\odot\{\!|m_1'', m_2|\!\}})$. Then canonical is applied recursively. Eventually, a situation will be reached where message m_1 is not an exponentiation message itself and we are in one of the cases (i), (ii) or (iv), where m_1 is not an exponential message.
  - Case (iv). Whenever none of the above applies.
    In this case, we have $\mathit{canonical}(m_1^{m_2}) =_S \mathit{canonical}(m_1)^{\mathit{canonical}(m_2)}$. From our induction hypothesis, we know that $\mathit{canonical}(m_1) =_M m_1$ and $\mathit{canonical}(m_2) =_M m_2$ and that $\mathit{canonical}(m_1)$ and $\mathit{canonical}(m_2)$ are both in canonical form. Hence $\mathit{canonical}(m) =_M m$. From the definition of the canonical form, we can further see that $\mathit{canonical}(m)$ is in canonical form.
- $m =_S (m_1)^{-1}$.
  - Case (i). $m_1 =_S (m_2)^{-1}$.
    Here we have $\mathit{canonical}(m) =_S \mathit{canonical}(m_2)$. In this case, we can derive with property (4) from page 11 that $m =_M m_2$. From our induction hypothesis, we have $\mathit{canonical}(m_2) =_M m_2$ and that $\mathit{canonical}(m_2)$ is in canonical form. Hence $\mathit{canonical}(m) =_M \mathit{canonical}(m_2) =_M m_2 =_M m$ and $\mathit{canonical}(m)$ is in canonical form.
  - Case (ii). Whenever case (i) does not apply.
    In this case, we have $\mathit{canonical}(m) =_S (\mathit{canonical}(m_1))^{-1}$. By our induction hypothesis we have $\mathit{canonical}(m_1) =_M m_1$ and that $\mathit{canonical}(m_1)$ is in canonical form. We get $\mathit{canonical}(m) =_M (\mathit{canonical}(m_1))^{-1} =_M (m_1)^{-1}$. From the definition of the canonical form we can derive that $\mathit{canonical}(m)$ is in canonical form.

We can see that the required property holds in all cases. □

Lemma 2.12. *Let m be an arbitrary message. Then the canonical form of m exists and is unique, i.e., there is exactly one message m' in canonical form such that $m =_M m'$.*

Proof Sketch. Let m be an arbitrary message. From Lemma 2.11, we know that $\mathit{canonical}(m) =_M m$ and that $\mathit{canonical}(m)$ is in canonical form. Consequently, we can conclude that there is a canonical form for every message.

What remains to be shown is that the canonical form is unique. To this end, suppose that there are two different canonical forms for m, i.e., assume that there are messages m' and m'' in canonical form for which $m =_M m'$ and $m =_M m''$ but $m' \neq_S m''$. Since the messages m' and m'' do not have the same shape but are still equal in equational theory M, we need to check the properties (1) to (10) on page 11:

- Property (1): $\langle m_1 \,.\, \langle m_2 \,.\, m_3 \rangle \rangle =_M \langle \langle m_1 \,.\, m_2 \rangle \,.\, m_3 \rangle$
  Here, only the left-hand side can be in canonical form.
- Property (2): $\{\{m\}_k^{\mathbf{s}}\}_k^{\mathbf{s}} =_M m$
  Here, only the right-hand side can be in canonical form.

- Property (3): $\{\{m\}_k^{\mathbf{a}}\}_{k^{-1}}^{\mathbf{a}} =_M m$
  Here, only the right-hand side can be in canonical form.
- Property (4): $(k^{-1})^{-1} =_M k$
  Here, only the right-hand side can be in canonical form.
- Property (5): $m_1 \odot m_2 =_M m_2 \odot m_1$
  Here, both sides are represented as $\odot\{\!|m_1, m_2|\!\}$ in canonical representation, i.e., both $m_1 \odot m_2$ and $m_2 \odot m_1$ have the same canonical form (same shape).
- Property (6): $(m_1 \odot m_2) \odot m_3 =_M m_1 \odot (m_2 \odot m_3)$
  Here, both sides are represented as $\odot\{\!|m_1, m_2, m_3|\!\}$ in canonical representation, i.e., both $(m_1 \odot m_2) \odot m_3$ and $m_1 \odot (m_2 \odot m_3)$ have the same canonical form (same shape).
- Property (7): $(m_1^{m_2})^{m_3} =_M m_1^{m_2 \odot m_3}$
  Here, only the right-hand side can be in canonical form.
- Property (8): $m_1 \odot 1 =_M m_1$
  Here, only the right-hand side can be in canonical form.
- Property (9): $(m_1)^1 =_M m_1$
  Here, only the right-hand side can be in canonical form.
- Property (10): $1^{m_1} =_M 1$
  Here, only the right-hand side is in canonical form.

We can see that there is no case in which the two messages m' and m'' in canonical form have the same value because of one of the properties (1) to (10) and different shape. Consequently, the canonical form of a message is unique. □

Proposition 2.13. *Let m_1 and m_2 be messages. We have $m_1 =_M m_2$ if and only if $canonical(m_1) =_S canonical(m_2)$.*

Proof Sketch. This is a direct corollary of Lemmas 2.11 and 2.12:

- $m_1 =_M m_2 \Rightarrow canonical(m_1) =_S canonical(m_2)$.
  We know from Proposition 2.12 that messages that are equal in equational theory M have the same canonical form. Therefore m_1 and m_2 have the same unique canonical form. From Proposition 2.11 we know that this canonical form is $canonical(m_1) =_S canonical(m_2)$.
- $canonical(m_1) =_S canonical(m_2) \Rightarrow m_1 =_M m_2$.
  We know from Proposition 2.11 that $canonical(m_1) =_M m_1$ and $canonical(m_2) =_M m_2$. From $canonical(m_1) =_S canonical(m_2)$ we can conclude that $canonical(m_1) =_M canonical(m_2)$ and consequently $m_1 =_M canonical(m_1) =_M canonical(m_2) =_M m_2$.

□

Lemma 2.18. *Let m and r be two messages in canonical form. Then we have:*

(i) If m is divisible by r, then we have $m = r \odot (m \div r)$ and $m \div r$ is in canonical form.

(ii) If m is left reducible by r, then we have $m = r^{(m \triangleleft r)}$ and $m \triangleleft r$ is in canonical form.

(iii) If m is right reducible by r, then we have $m = (m \triangleright r)^r$ and $m \triangleright r$ is in canonical form.

Proof Sketch. Let us prove the three points separately.

(i)
- Case $m =_S r$. Then $m \div r =_S 1$ and therefore we have $r \odot (m \div r) =_S m \odot 1 =_M m$ where we have used property (8) from page 11 in the last step and $m \div r$ is in canonical form since 1 is in canonical form.
- Case $r =_S 1$. Then $m \div r =_S m$ and therefore we have $r \odot (m \div r) =_M 1 \odot m =_M m \odot 1 =_M m$ where we have used properties (8) and (5) from page 11 and $m \div r =_S m$ is in canonical since m is in canonical form.

- Case $m =_S \odot M$ and $r =_S \odot R$ and $R \subset_S M$. Then $m \div r =_S \mathit{canonical}(\odot(M \backslash R))$. We have $\odot(M \backslash R) =_M \mathit{canonical}(\odot(M \backslash R))$ and $m \div r$ is in canonical form from Lemma 2.11 on page 18. In canonical form, we represent multiplication by a multiset where we have $(\odot A) \odot (\odot B) =_M \odot(A \cup B)$ for any multisets A and B. It is clear that $M = R \cup (M \backslash R)$ and hence that $m =_M \odot M =_M \odot(R \cup (M \backslash R)) =_M (\odot R) \odot (\odot(M \backslash R)) =_M r \odot (m \div r)$.
- We have $\odot(M \backslash \{r\}) =_M \mathit{canonical}(\odot(M \backslash \{r\}))$ and $m \div r$ is in canonical form (Lemma 2.11 on page 18). Obviously, $M = \{r\} \cup (M \backslash \{r\})$ and hence $m =_M \odot M =_M \odot(\{r\} \cup (M \backslash \{r\})) =_M r \odot (\odot(M \backslash \{r\})) =_M r \odot (m \div r)$.
- In all other cases, $m \div r$ is not defined.

(ii)
- Case $m =_S m_1^{m_2}$ and $r =_S r_1^{r_2}$ and $m_1 =_S r_1$ and m_2 is divisible by r_2. Then $m \triangleleft r =_S m_2 \div r_2$. We have $r^{m \triangleleft r} =_S (r_1^{r_2})^{m_2 \div r_2} =_M r_1^{r_2 \odot (m_2 \div r_2)} =_M r_1^{m_2} \overset{r_1 =_S m_1}{=_M} m_1^{m_2} =_S m$ where we have used property (7) from page 11 and the properties of division that we have shown above. $m \triangleleft r$ is in canonical form since $m \div r$ is in canonical form.
- Case $m = m_1^{m_2}$ and r is not an exponentiation message and $m_1 =_S r$. Then $m \triangleleft r =_S m_2$. We have $r^{m \triangleleft r} =_S r^{m_2} \overset{m_1 =_S r}{=_S} m_1^{m_2} =_S m$. $m \triangleleft r$ is in canonical form since m_2 is in canonical form.
- In all other cases, $m \triangleleft r$ is not defined.

(iii)
- Case $m =_S m_1^{m_2}$ and m_2 is divisible by r. Then we have $m \triangleright r =_S m_1^{m_2 \div r}$. We have $(m \triangleright r)^r =_S (m_1{}^{m_2 \div r})^r =_M m_1{}^{(m_2 \div r) \odot r} =_M m_1^{m_2} =_S m$. From the definition of division, we can see that it returns a non-exponential message whenever it is defined and can conclude that $m_2 \div r$ is non-exponential. We know that m_1 is in canonical form and that $m_2 \div r$ is in canonical form (we have just shown this) and can consequently conclude that $m_1^{m_2 \div r}$ is in canonical form as well.
- In all other cases, $m \triangleright r$ is not defined.

□

Proposition 2.19. *Let m be a message in canonical form and let χ be a basic set with $\mathit{synth}^*(\chi) = \mathit{close}(\chi)$. Then $\mathit{cf}_\chi(m)$ is defined if and only if $m \in \mathit{synth}^*(\chi)$. If defined, then $\mathit{cf}_\chi(m)$ is composed of messages that are in χ and describes a valid way of constructing m from χ.*

Proof Sketch. Let us prove the directions separately.

- $\mathit{cf}_\chi(m)$ is defined $\Rightarrow m \in_M \mathit{synth}^*(\chi)$.
  This proof is by structural induction on messages. Let us do a case distinction based on the shape of the message m.
  - m is an atomic message.
    In this case, $\mathit{cf}_\chi(m)$ is defined if and only if $m \in_S \chi$. We know that since m is in canonical form that $m \in_S \chi \Leftrightarrow m \in_M \chi$ (Lemma 2.11 on page 18 and Proposition 2.13 on page 19) and consequently $m \in_M \mathit{synth}^*(\chi)$ whenever $\mathit{cf}_\chi(m)$ is defined.
  - $m =_S \langle m_1 \,.\, m_2 \rangle$.
    In this case, $\mathit{cf}_\chi(m)$ is defined if and only if $\mathit{cf}_\chi(m_1)$ and $\mathit{cf}_\chi(m_2)$ are both defined. By our induction hypothesis, we know that $m_1 \in_M \mathit{synth}^*(\chi)$ and $m_2 \in_M \mathit{synth}^*(\chi)$. By applying synthesis rule SCONCAT on m_1 and m_2, we can conclude that $m =_M \langle m_1 \,.\, m_2 \rangle \in_M \mathit{synth}^*(\chi)$.
  - $m =_S \{m_1\}^{\mathbf{a}}_{m_2}$.
    In this case, $\mathit{cf}_\chi(m)$ is defined if either (1) both $\mathit{cf}_\chi(m_1)$ and $\mathit{cf}_\chi(m_2)$ are defined, or (2) $m \in_S \chi$. Otherwise, $\mathit{cf}_\chi(m)$ is undefined. In case (1), we know by our induction hypothesis that $m_1 \in_M \mathit{synth}^*(\chi)$ and $m_2 \in_M \mathit{synth}^*(\chi)$. By applying synthesis rule SAENC on m_1 and m_2 we get that $m =_S \{m_1\}^{\mathbf{a}}_{m_2} \in_M \mathit{synth}^*(\chi)$. In case (2) we can conclude $m =_S \{m_1\}^{\mathbf{a}}_{m_2} \in_M \mathit{synth}^*(\chi)$ directly from $m \in_S \chi$ (Proposition 2.13).
  - $m =_S \{m_1\}^{\mathbf{s}}_{m_2}$.
    In this case, $\mathit{cf}_\chi(m)$ is defined if either (1) both $\mathit{cf}_\chi(m_1)$ and $\mathit{cf}_\chi(m_2)$ are defined, or (2) $m \in_S \chi$. Otherwise, $\mathit{cf}_\chi(m)$ is undefined. In case (1), we know by our induction hypothesis

that $m_1 \in_M synth^*(\chi)$ and $m_2 \in_M synth^*(\chi)$. By applying synthesis rule SSENC on m_1 and m_2 we get that $m =_S \{m_1\}^{\mathbf{s}}_{m_2} \in_M synth^*(\chi)$. In case (2) we can conclude $m =_S \{m_1\}^{\mathbf{s}}_{m_2} \in_M synth^*(\chi)$ directly from $m \in_S \chi$ (Proposition 2.13).

- $m =_S \mathbf{h}(m_1)$.
  In this case, $cf_\chi(m)$ is defined if (1) $cf_\chi(m_1)$ is defined, or (2) $m \in_S \chi$. Otherwise, $cf_\chi(m)$ is undefined. In case (1), we can conclude by our induction hypothesis that $m_1 \in_M synth^*(\chi)$. By applying synthesis rule SHASH on m_1 we get that $m =_S \mathbf{h}(m_1) \in_M synth^*(\chi)$. In case (2) we can conclude $m =_S \mathbf{h}(m_1) \in_M synth^*(\chi)$ directly from $m \in_S \chi$ (Proposition 2.13).
- $m =_S fun(m_1, \ldots, m_i)$.
  In this case, $cf_\chi(m)$ is defined if (1) $cf_\chi(m_1), \ldots, cf_\chi(m_i)$ are defined, or (2) $m \in_S \chi$. Otherwise, $cf_\chi(m)$ is undefined. In case (1), we can conclude by our induction hypothesis that $m_k \in_M synth^*(\chi)$ for $k = 1, \ldots, i$. By applying synthesis rule SAPPL on $m_1, \ldots, m_i$, we get that $m =_S fun(m_1, \ldots, m_i) \in_M synth^*(\chi)$. In case (2), we can conclude $m =_S fun(m_1, \ldots, m_i) \in_M synth^*(\chi)$ directly from $m \in_S \chi$ (Proposition 2.13).
- $m =_S \odot\{\!|m_1, \ldots, m_i|\!\}$.
  In this case, $cf_\chi(m) = constrMultTerm_\chi(m)$ which is defined if (1) $m \in_S \chi$, (2) there is a message $r \in_S \chi$ by which m is divisible and $m \div r$ is synthesizable from χ, or (3) there is a factor m_k, $k \in \{1, \ldots, i\}$ that is synthesizable from χ and for which $m \div m_k$ is synthesizable. In all other cases, $cf_\chi(m)$ is undefined.

  In case (1), we can conclude $m \in_M synth^*(\chi)$ directly from $m \in_S \chi$ (Proposition 2.13).

  In case (2), we know that $m \div r$ is synthesizable, i.e., $cf_\chi(m \div r)$ is defined. We know that $m =_M r \odot (m \div r)$ from Lemma 2.18 on page 21. Since $(m \div r)$ is a sub-message of $r \odot (m \div r)$, we can apply our induction hypothesis and conclude that $m \div r \in_M synth^*(\chi)$. We can then apply synthesis rule SMUL on r and $m \div r$ and conclude that $m =_M r \odot (m \div r) \in_M synth^*(\chi)$.

  In case (3) we know that both m_k and $m \div m_k$ are synthesizable (note that m is always divisible by m_k since m_k is a factor of m), that is, $cf_\chi(m_k)$ and $cf_\chi(m \div m_k)$ are both defined. We can now apply our induction hypothesis and conclude that $m_k \in_M synth^*(\chi)$ and $m \div m_k \in_M synth^*(\chi)$. We can then apply synthesis rule SMUL on m_k and $m \div m_k$ and conclude that $m =_M m_k \odot (m \div m_k) \in_M synth^*(\chi)$ with the help of Lemma 2.18 on page 21.
- $m =_S {m_1}^{m_2}$. In this case, $cf_\chi(m) = constrExpTerm_\chi(m)$ which is defined if (1) $m \in_S \chi$, (2) $cf_\chi(m_1)$ and $cf_\chi(m_2)$ are defined, (3) there is a message $r \in_S \chi$ such that m is left reducible by r and $m \triangleleft r$ is synthesizable from χ, or (4) there is a message $r \in_S \chi$ such that m is right reducible by r and $m \triangleright r$ is synthesizable from χ.

  In case (1), we can conclude $m \in_M synth^*(\chi)$ directly from $m \in_S \chi$ (Proposition 2.13).

  In case (2), we can apply our induction hypothesis and get that if $cf_\chi(m_1)$ is defined then $m_1 \in synth^*(\chi)$ and similarly that if $cf_\chi(m_2)$ is defined then $m_2 \in synth^*(\chi)$. By applying synthesis rule SEXP, we then get $m =_S m_1^{m_2} \in_M synth^*(\chi)$.

  In cases (3) and (4), we can again apply our induction hypothesis and get that whenever $cf_\chi(m \triangleleft r)$ is defined then $m \triangleleft r \in_M synth^*(\chi)$ and whenever $cf_\chi(m \triangleright r)$ is defined then $m \triangleright r \in_M synth^*(\chi)$. Furthermore, we have $r \in_M synth^*(\chi)$ from $r \in_S \chi$. With Lemma 2.18, we get $m \in_M synth^*(\chi)$ with $m =_M r^{m \triangleleft r}$ in case (3) and $m =_M (m \triangleright r)^r$ in case (4).

We see that the implication holds in all cases.

- $m \in_M synth^*(\chi) \Rightarrow cf_\chi(m)$ is defined.
  First recall that the implication only needs to hold if m is in canonical form (Definition 2.10 on page 17). We have $m \in_S \chi \Rightarrow m \in_M synth^*(\chi)$, i.e., $cf_\chi(m)$ has to be defined whenever $m \in_S \chi$. This is always true except in the case of concatenation. What remains are the other cases where $m \in_M synth^*(\chi)$ (and the case of concatenation). The proof here is again by structural induction. Let us make a case distinction based on the shape that a message in canonical form can assume:

  - m is an atomic message.
    In this case, we have $m \in_M synth^*(\chi) \Leftrightarrow m \in_S \chi$ since atomic messages cannot be constructed from other messages. The implication holds since $cf_\chi(m)$ is defined if and only if $m \in_S \chi$.

– $m =_S \langle m_1 \,.\, m_2 \rangle$.
From $\chi = analyze^*(\chi)$ and the analysis rules ACONCAT1 and ACONCAT2, we can conclude that concatenation is always taken apart in χ and since χ is a basic set, we have no concatenation messages in χ (for example, the message $\gamma_{\langle m_1 \,.\, m_2 \rangle}$ cannot be in χ since there is also m_1 and m_2, making the ghost message synthesizable). Therefore, $m \in_M synth^*(\chi)$ if and only if $m_1 \in_M synth^*(\chi)$ and $m_2 \in_M synth^*(\chi)$. From our induction hypothesis, we know that $m_1 \in_M synth^*(\chi)$ implies that $cf_\chi(m_1)$ is defined and, similarly, that $m_2 \in_M synth^*(\chi)$ implies that $cf_\chi(m_2)$ is defined. From the definition of the constructive form we can read off that it is only defined if both $cf_\chi(m_1)$ and $cf_\chi(m_2)$ are defined. Note that this is compatible with property (1) from page 11 because of $\chi = analyze^*(\chi)$ (χ is completely analyzed).

– $m =_S \{m_1\}^{\mathbf{a}}_{m_2}$.
We know that $m_1 \neq_S \{m_1'\}^{\mathbf{a}}_{m_2^{-1}}$ since m is in canonical form. Consequently, SAENC is the only rule we can apply to construct a message that equals m in equational theory M, i.e., m is synthesizable only if $m_1 \in_M synth^*(\chi)$ and $m_2 \in_M synth^*(\chi)$. From our induction hypothesis we know that $m_1 \in_M synth^*(\chi) \Rightarrow cf_\chi(m_1)$ is defined and $m_2 \in_M synth^*(\chi) \Rightarrow cf_\chi(m_2)$ is defined and consequently $m \in_M synth^*(\chi) \Rightarrow cf_\chi(m_1)$ is defined and $cf_\chi(m_2)$ is defined. This is exactly what the constructive form requires, i.e., $m \in_M synth^*(\chi) \Rightarrow cf_\chi(m)$.

– $m =_S \{m_1\}^{\mathbf{s}}_{m_2}$.
We know that $m_1 \neq_S \{m_1'\}^{\mathbf{s}}_{m_2}$ since m is in canonical form. Consequently, SSENC is the only rule we can apply to construct a message that equals m in equational theory M, i.e., m is synthesizable only if $m_1 \in_M synth^*(\chi)$ and $m_2 \in_M synth^*(\chi)$. From our induction hypothesis we know that $m_1 \in_M synth^*(\chi) \Rightarrow cf_\chi(m_1)$ is defined and $m_2 \in_M synth^*(\chi) \Rightarrow cf_\chi(m_2)$ is defined and consequently $m \in_M synth^*(\chi) \Rightarrow cf_\chi(m_1)$ is defined and $cf_\chi(m_2)$ is defined. This is exactly what the constructive form requires, i.e., $m \in_M synth^*(\chi) \Rightarrow cf_\chi(m)$.

– $m =_S \mathbf{h}(m_1)$.
There is no equational property that includes hashing in equational theory M. Hence, m is synthesizable only if rule SHASH can be applied, i.e., m is synthesizable only if $m_1 \in_M synth^*(\chi)$. From our induction hypothesis, we know that $m_1 \in_M synth^*(\chi) \Rightarrow cf_\chi(m_1)$. This is exactly what the constructive form requires and hence $m \in_M synth^*(\chi) \Rightarrow cf_\chi(m)$.

– $m =_S foo(m_1, \ldots, m_i)$.
There is no equational property that includes function application in equational theory M. Hence, m is synthesizable only if rule SAPPL can be applied, i.e., m is synthesizable only if $m_1, \ldots, m_i \in_M synth^*(\chi)$. From our induction hypothesis, we know that $m_k \in_M synth^*(\chi) \Rightarrow cf_\chi(m_k)$ for $k \in \{1, \ldots, i\}$. This is exactly what the constructive form requires and hence $m \in_M synth^*(\chi) \Rightarrow cf_\chi(m)$.

– $m =_S \odot\{\!|m_1, \ldots, m_i|\!\}$.
In this case, we have $cf_\chi(m) = constrMultTerm_\chi(m)$.
The message m is in canonical form and is therefore represented as a multiset of factors. Furthermore, the canonical form requires that none of $m_1, \ldots, m_i$ are themselves multiplicative messages. This means that the multiplication only has the factors $m_1, \ldots, m_i$. The multiset representation also covers the properties (5) and (6) of equational theory M (that is, it solves the problems of commutativity and associativity). Therefore, $m \in synth^*(\chi)$ only if either (1) $m \in_S \chi$ or if (2) there are two messages $a, b \in_M synth^*(\chi)$ to which SMUL can be applied and for which $m =_M a \odot b$.

(1) This case is handled by the condition $m \in_S \chi$ at the beginning of $constrMultTerm_\chi$.

(2) Suppose that there are $a, b \in_M synth^*(\chi)$ such that $m =_M a \odot b$. The messages a and b are either single factors of m, or they are multiplications of several factors of m. Let us first take a look at the second loop of $constrMultTerm_\chi$. It iterates over all factors f of m and checks if they are synthesizable from χ. Since f is a sub-message of m, we can apply our induction hypothesis and get $f \in_M synth^*(\chi) \Rightarrow cf_\chi(f)$ is defined. That is, if there is a factor that is synthesizable from χ, then it is detected by $constrMultTerm_\chi$. Then the algorithm checks if $m \div f$ is constructible. We can again apply our induction hypothesis and get $m \div f \in_M synth^*(\chi) \Rightarrow cf_\chi(m \div f)$ is defined . From Lemma 2.18 on page 21, we get that $m =_M f \odot (m \div f)$.

What remains are the cases where m can be constructed as $m =_M a \odot b$ but neither a nor b is a single factor of m. This can only be the case if there are multiplicative ghost messages in χ. This is handled by the first loop of $constrMultTerm_\chi$. It checks if there is a message $r \in_S \chi$ such that m is divisible by r and $m \div r$ is synthesizable from χ. By our induction hypothesis, we get that $r \in_M synth^*(\chi) \Rightarrow cf_\chi(r)$ is defined and $m \div r \in_M synth^*(\chi) \Rightarrow cf_\chi(m \div r)$ is defined. From Lemma 2.18 on page 21, we get that $m =_M f \odot (m \div f)$.
Summarizing, we can conclude that the implication holds.

- $m =_S m_1^{m_2}$.
  In this case, we have $cf_\chi(m) = constrExpTerm_\chi(m)$.
  The message m is in canonical form and therefore m_1 is not an exponentiation message itself and m_2 is not the message 1.
  By applying rule SEXP, we can construct m directly from the messages m_1 and m_2. Due to property (7) we can also construct m as an exponentiation of two messages a and b, where a has the shape m_1^c such that $a^b =_S (m_1^c)^b =_M m_1^{b \odot c} =_M m$.
  Let us now show that whenever $m \in_M synth^*(\chi)$, we have that $constrExpTerm_\chi(m)$ is defined.
  The first condition of the algorithm checks if m is directly in χ. Therefore, we have $m \in_M synth^*(\chi) \Rightarrow cf_\chi(m)$ in this case.
  Next, the algorithm checks if m_1 and m_2 can be synthesized from χ. By our induction hypothesis, we have $m_1 \in_M synth^*(\chi) \Rightarrow cf_\chi(m_1)$ is defined and $m_2 \in_M synth^*(\chi) \Rightarrow cf_\chi(m_2)$ is defined. Consequently, we have $m \in_M synth^*(\chi) \Rightarrow cf_\chi(m)$ in this case.
  In the loop, we check for each $r \in_S \chi$ whether it can be used for right or left exponentiation. Note that from our induction hypothesis, we have $m \triangleleft r \in_M synth^*(\chi) \Rightarrow cf_\chi(m \triangleleft r)$ is defined and $m \triangleright r \in_M synth^*(\chi) \Rightarrow cf_\chi(m \triangleright r)$ is defined and therefore in both cases $m \in_M synth^*(\chi) \Rightarrow cf_\chi(m)$ (Lemma 2.18).
  Note that there are no further cases. Either, we can construct m_1, or m_1 is contained in an exponential ghost term. Consequently, all cases are covered and $m \in_M synth^*(\chi) \Rightarrow cf_\chi(m)$ in all cases.
- $m =_S m_1^{-1}$.
  Since our principals do not have the capability of constructing the inverse of a message, they only can synthesize it if they directly possess it in their knowledge which is exactly what the constructive form requires with $m \in_S \chi$.

Finally, we request that the canonical form describes a valid way of synthesizing m from χ. This can be seen very easily from the explanations above. □

Lemma 2.21. *Let S be a set of messages. Then $\chi := basicSet(S)$ is a basic set and $synth^*(\chi) = synth^*(S)$.*

Proof Sketch. According to Definition 2.6 on page 16, χ is a basic set if for all messages $m \in_S \chi$ we have $m \notin_M synth^*(\chi \backslash \{m\})$. According to Proposition 2.19 on page 22, $m \notin_M synth^*(\chi \backslash \{m\})$ holds if and only if m is not synthesizable from $\chi \backslash \{m\}$, that is, if and only if $cf_{\chi \backslash \{m\}}(m)$ is not defined. This is exactly the termination criterion of the loop in algorithm *basicSet*, in other words, whenever *basicSet* terminates then χ is a basic set.

On every iteration, one message is removed from the set of messages and since the algorithm terminates at the latest when there are no more messages left in this set, we can conclude that *basicSet* always terminates. Consequently, *basicSet* always terminates and returns a basic set.

What remains to be shown is $synth^*(S) =_M synth^*(basicSet(S))$ for any set of messages S. Let us consider an arbitrary iteration of the loop and denote by S_a the set of messages before the loop execution. According to the specification of *basicSet*, the loop is only executed if there is a message m that can be synthesized from $S_a \backslash \{m\}$. The set of messages after the loop iteration is $S_b := S_a \backslash \{m\}$. We know that m can be synthesized from S_b, that is, $cf_{S_b}(m)$ is defined, and consequently we get $m \in_M synth^*(S_b)$ by Proposition 2.19. This in turn implies that $synth^*(S_b) =_M synth^*(S_b \cup \{m\}) =_M synth^*(S_a)$. We can conclude by induction that $synth^*(basicSet(S)) =_M synth^*(S)$ which concludes the proof. □

Proposition 2.22. *Let Q and R be two sets of messages. Then there are unique basic sets Q' and R' with $synth^*(Q) =_M synth^*(Q')$ and $synth^*(R) =_M synth^*(R')$. For these, we have $synth^*(Q) =_M synth^*(R)$ if and only if $Q' =_M R'$.*

Proof Sketch. This is a direct corollary of Lemma 2.21 and Proposition 2.7. From Lemma 2.21 we know that there is a basic set Q' with $synth^*(Q') =_M synth^*(Q)$. Similarly, there is a basic set R' with $synth^*(R') =_M synth^*(R)$. From Proposition 2.7, we know $synth^*(Q) =_M synth^*(R) \Leftrightarrow Q' =_M R'$ which also implies that the basic sets Q' and R' are unique (in equational theory M, not necessarily by shape). □

Proposition 2.23. *Let m be a message and χ a basic set. Then $addKnowledge_\chi(m)$ is a basic set and $synth^*(addKnowledge_\chi(m)) =_M close(\chi \cup \{m\})$.*

Proof Sketch. Let us denote the initial knowledge with which the algorithm is called by χ^{start}.

The (gradually growing) knowledge in the algorithm is represented by the variable χ which is a finite set of messages. The set of messages that we can construct from it is exactly the set $synth^*(\chi)$. Recall that according to Proposition 2.19, we have $cf_\chi(m)$ is defined $\Leftrightarrow m \in_M synth^*(\chi)$.

Let us now take a look at $analyzeOnce_\chi$. We can see that $analyzeOnce_\chi$ implements all of the analysis rules from Definition 2.8 on page 16. In particular, $analyzeOnce_\chi(m)$ contains all the messages that can be extracted from $synth^*(\chi)$ by applying one analysis rule since in the case of the rules that take two messages, it checks if the second message is in $synth^*(\chi)$ by computing $cf_\chi(m)$. Consequently, the `while` loop terminates exactly when no more information can be analyzed from $synth^*(\chi)$, that is, when $synth^*(\chi) =_M analyze(synth^*(\chi))$ which implies $synth^*(\chi) =_M close(\chi)$.

The first instruction of the algorithm adds m to χ^{start}. After that, messages are only added to χ, never removed (at least as long as the loop is not left). Also, only messages that can be analyzed from χ are added. Combined, this implies that $close(\chi) =_M close(\chi^{start} \cup \{m\})$.

Summarizing, the loop is left exactly when $synth^*(\chi) =_M close(\chi)$. Together with the fact $close(\chi) =_M close(\chi^{start} \cup \{m\})$ we can conclude that the loop is left exactly when $synth^*(\chi) =_M close(\chi^{start} \cup \{m\})$.

The return value of $addKnowledge_{\chi^{start}}(m)$ is then $basicSet(\chi)$. From Proposition 2.21 on page 24, we know that $addKnowledge_{\chi^{start}}(m) =_M basicSet(\chi)$ is a basic set and $synth^*(\chi) = synth^*(basicSet(\chi))$. Hence, we have $synth^*(addKnowledge_{\chi^{start}}(m)) =_M synth^*(basicSet(\chi)) =_M synth^*(\chi) =_M close(\chi^{start} \cup \{m\})$.

□

Proposition 5.1. *The capabilities of honest principals that are induced by the function declarations, equations and the built-in theory `diffie-hellman` from above are equivalent to the capabilities that arise from Definitions 2.5 and 2.8 and the properties from Section 2.2.2.*

Proof Sketch. We see immediately that the messages used in Tamarin satisfy all the properties that we defined in Section 2.2.2; associativity of concatenation is implemented in Tamarin by default, the properties of exponentiation and multiplication are added with the built-in `diffie-hellman` and the other properties arise from the equations. What remains to be shown is that the synthesis capabilities from Definition 2.5 on page 15 and the analysis capabilities from Definition 2.8 on page 16 follow from the equations specified in Tamarin.

Let us first discuss the synthesis capabilities. We note that concatenation is expressed by the function `pair`, symmetric encryption by `senc`, asymmetric encryption by the function `aenc` and hashing by the function `h`. Since function application is there in Tamarin by default, we can conclude that the rules Sappl, Sconcat, Saenc, Ssenc, and Shash are implemented. Finally Sexp and Smul are added with the operators `*` and `^` by the built-in theory `diffie-hellman`.

The analysis capabilities are expressed by the equations. In particular, Asenc is implemented by `sdec(senc(x.1, x.2), x.2) = x.1`, Aaenc1 by `adec(aenc(x.1, sk(x.2)), pk(x.2)) = x.1`, Aaenc2 by `adec(aenc(x.1, pk(x.2)), sk(x.2)) = x.1` (since we do not work with the inverses of general messages in A&B), Aconcat1 by `fst(<x.1, x.2>) = x.1`, and, Aconcat2 by `snd(<x.1, x.2>) = x.2`.

□

B Grammar of the A&B Input Language

The grammar of the A&B protocol specification language is defined in this section.

Identifiers and Strings. Identifiers must start with a letter, followed by any number of letters, digits, and underscores ('_'). A&B as such is not case sensitive, but special rules apply for identifiers: there must not be two identifiers that differ only in case. For example, the two identifiers "`Message`" and "`message`" must not appear in the same protocol. String literals are identifiers that are enclosed in single quotes.

```
IDENT    ::= Letter { Letter | Digit | "_" }*
STRING   ::= "'" IDENT "'"
```

Messages. There are 5 types of atomic messages, namely strings, variables, secret keys, public keys and symmetric keys. All other messages are composed from atomic messages.

```
MESSAGE  ::= IDENT                              // variable name or role name
           | STRING                             // a string in single quotes
           | "pk" "(" IDENT ")"                 // public key of an agent
           | "sk" "(" IDENT ")"                 // secret key of an agent
           | "k" "(" IDENT "," IDENT ")"        // shared symmetric key
                                                // of two agents
           | MESSAGE "." MESSAGE                // concatenation
           | "<" MESSAGE {"." MESSAGE}+ ">"     // concatenation in angles
           | "(" MESSAGE ")"                    // parentheses for
                                                // determining precedence
           | "aenc" "{" MESSAGE "}" MESSAGE     // asymmetric encryption
           | "senc" "{" MESSAGE "}" MESSAGE     // symmetric encryption
           | MESSAGE "*" MESSAGE                // multiplication
           | MESSAGE "^" MESSAGE                // exponentiation
           | "h" "(" MESSAGE ")"                // hash of a message
           | IDENT "(" [MESSAGE {"," MESSAGE}*] ")" // function application
```

Protocol Specification. A protocol specification consists of four blocks, namely a **`Declarations`** block where functions are declared, a **`Knowledge`** block where the initial knowledge of the roles is specified, an **`Actions`** block describing the protocol and a **`Goals`** block where the security goals are specified. Any of these blocks can be left away if they are not needed:

```
PROTOCOL     ::= "Protocol" IDENT ":"
                 [ DECLARATIONS ] [ KNOWLEDGE ] [ ACTIONS ] [ GOALS ]
                 "end"

DECLARATIONS ::= "Declarations" ":" { DECLARATION }*

DECLARATION  ::= ["public" | "private"] IDENT "/" INTEGER ";"

KNOWLEDGE    ::= "Knowledge" ":" { KNOWS }*

KNOWS        ::= IDENT ":" [ MESSAGE ] {"," MESSAGE}* ";"

ACTIONS      ::= "Actions" ":" { ACTION }*

ACTION       ::= "[" IDENT "]" IDENT "->" IDENT
                 ["(" IDENT { "," IDENT }* ")"] ":" MESSAGE ";"
               | "[" IDENT "]" IDENT "<-" IDENT
                 ["(" IDENT { "," IDENT }* ")"] ":" MESSAGE ";"
```

```
GOALS        ::= "Goals" ":" { GOAL }*

GOAL         ::= "[" IDENT "]" IDENT "non-injectively" "agrees" "with"
                 IDENT "on" MESSAGE ";"
               | "[" IDENT "]" IDENT "injectively" "agrees" "with"
                 IDENT "on" MESSAGE ";"
               | "[" IDENT "]" MESSAGE "secret" "of" IDENT { "," IDENT } ";"
```

Reserved Identifiers. The following identifiers are either reserved keywords of A&B or could lead to naming conflicts in TAMARIN and must therefore not be used:

`Actions`, `adec`, `aenc`, `agrees`, `Declarations`, `em`, `end`, `fst`, `Goals`, `h`, `in`, `init`, `inv`, `injectively`, `k`, `Knowledge`, `let`, `non-injectively`, `of`, `on`, `pk`, `pair`, `pmult`, `Protocol`, `private`, `public`, `rule`, `secret`, `sdec`, `senc`, `sign`, `snd`, `true` `sk`, `verify`, `with`.

C Intermediate Representation Format

In the following, we give a listing of all data structures that are part of the intermediate representation format and the packages in which they are defined.

Package `Parser.Basic`:

```
type Label      = String;
type RoleName   = String;
type Identifier = String;
```

Package `Parser.Message`:

```
type Factors = Data.MultiSet.MultiSet Message

data Message
    = Gamma Identifier Message
    | Var Identifier
    | Str String
    | Concat Message Message
    | Aenc Message Message
    | Senc Message Message
    | Hash Message
    | Mul Factors
    | Exp Message Message
    | Pk RoleName
    | Sk RoleName
    | K RoleName RoleName
    | Fun Identifier [Message]
    | One

type Knowledge = Data.Set.Set Message
```

Package `Rewriter.IR`:

```
data Protocol = Protocol Identifier [Function] [Role] [Goal]

type Function = (Identifier, Integer, Bool)

data Role = Role RoleName [Action]

data Action
    = Prepare Knowledge
    | Send Bool Label RoleName [Identifier] Message Knowledge
    | Receive Label RoleName Message Knowledge

data Goal
    = Secret Label Message [RoleName]
    | WeakAuth Label RoleName RoleName [Message]
    | StrongAuth Label RoleName RoleName [Message]
```

D Short Manual for the Translator

Compilation The Haskell source code of the translator can be downloaded from `http://www.infsec.ethz.ch/research/software/tamarin`. The package `Data.MultiSet` needs to be installed before it can be compiled. It can be installed via Cabal or alternatively downloaded from `https://hackage.haskell.org/package/multiset`. The main file of the translator is `code/anb.hs`. Using Cabal and the Glasgow Compiler, the translator can be compiled with the following instructions:

```
>> cd ./code
>> cabal install MultiSet
>> ghc anb.hs
```

The program can now be invoked with the command `./anb`.

Usage Assume that the executable of the translator is called `anb`. In this case, the translator is invoked according to the following pattern:

```
>> ./anb [input-file and options]
```

A short explanation of usage and possible options is displayed when called with the option `--help` (or `-h`). For instance:

```
>> ./anb --help
```

Only one file can be translated at a time and only input files with the file extension `.anb` are accepted. A file `protocol.anb` can be translated by invoking:

```
>> ./anb protocol.anb
```

This will read the A&B specification and write the corresponding Tamarin code in the file `protocol.spthy`. If the output should be written to file `out.spthy` instead, one can do this with the `-o` option:

```
>> ./anb protocol.anb -o out.spthy
```

Note that the translator does not produce any code if the A&B input does not pass all well-formedness checks. For verbose output, use the option `-v`:

```
>> ./anb protocol.anb -v
```

Verbose output contains the A&B specification, a representation of the IR and the generated Tamarin code.