Future AnB: The projected APS Language of FutureID

D42.3

Document Identification	
Date	30/08/2013
Status	Final
Version	1.0

Related SP/WP	SP4/WP42.3	**Document Reference**	LiveLink
Related Deliverable(s)	D42.1	**Dissemination Level**	PU
Lead Participant	DTU	**Lead Author**	Omar Almousa Sebastian Mödersheim
Contributors	Moritz Horsch(TUD)	**Reviewers**	Jaap-Henk Hoepman(RU) Peter Lipp(TUG)

Abstract: We introduce *Future AnB*: the projected Authentication Protocol Specification (APS) language for the *FutureID* project.

This document is issued within the frame and for the purpose of the *FutureID* project. This project has received funding from the European Unions Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 318424.

1 Executive Summary

We introduce *Future AnB*: the projected Authentication Protocol Specification (APS) language for the *FutureID* project. We present the syntax of *Future* AnB in Extended Bakus-Naur Form. We then give a formal semantics for our language by translation to (an extended version of) strands [1] from where we can easily connect to the input languages of various tools like AVISPA [2], [3] and ProVerif [4]. We also show how to translate protocol specifications into Java programs. We illustrate all the different aspects with a running example: the Extended Access Control (EAC) protocol [5], [6].

2 Document information

2.1 Contributors

Name	**Partner**
Omar Almousa	DTU
Moritz Horsch	TUD
Sebastian Mödersheim	DTU

2.2 History

0.1	2013-03-1	Omar And Sebastian	1st Draft
0.2	2013-05-15	Omar And Sebastian	2nd Draft
0.3	2013-07-15	Omar And Sebastian	3rd Draft

2.3 Table of Contents

2.4 List of References

[1] Fábrega, F.J.T., Herzog, J.C. and Guttman, J.D. *Strand spaces: Why is a security protocol correct?* In *Security and Privacy, 1998. Proceedings. 1998 IEEE Symposium on*, (IEEE1998) 160–171.

[2] AVISPA. *AVISPA Project: Automated Validation of Internet Security Protocols and Applications project.*
`http://www.avispa-project.org/`

[3] AVISPA. *Deliverable D2.3: The Intermediate Format*, 2003.

[4] Blanchet, B. and Smyth, B. *ProVerif 1.85: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial*, 2011.

[5] Bundesamt fur Sicherheit in der Informationstechnik (BSI). *Advanced Security Mechanism for Machine Readable Travel Documents Extended Access Control (EAC).* Technical report, Technical Report (BSI-TR-03110) Version 2.02 Release Candidate, 2008.

[6] Dagdelen, Özgür and Fischlin, Marc. *Security analysis of the extended access control protocol for machine readable travel documents.* In *Information Security*, 54–68, (Springer2011).

[7] Mödersheim, S. and Viganò, L. *The Open-source Fixed-point Model Checker for Symbolic Analysis of Security Protocols. Foundations of Security Analysis and Design V*, 2009. 166–194.

[8] Mödersheim, S. *Algebraic Properties in Alice and Bob Notation.* In *Availability, Reliability and Security, 2009. ARES'09. International Conference on*, (IEEE2009) 433–440.

[9] Dolev, Danny and Yao, Andrew. *On the security of public key protocols. Information Theory, IEEE Transactions on*, 1983. 29(2):198–208.

[10] Mödersheim, S. and Viganò, L. *Secure pseudonymous channels. Computer Security–ESORICS 2009*, 2009. 337–354.

[11] Cremers, Cas and Mauw, Sjouke. *Operational Semantics of Security Protocols.* In Stefan Leue and TarjaJohanna Systä, editors, *Scenarios: Models, Transformations and Tools*, volume 3466 of *Lecture Notes in Computer Science*, 66–89, (Springer Berlin Heidelberg2005). ISBN 978-3-540-26189-6. doi:10.1007/11495628_4.
`http://dx.doi.org/10.1007/11495628_4`

[12] Armando, A. and Compagna, L. *SATMC: A SAT-based Model Checker for Security Protocols. Logics in Artificial Intelligence*, 2004. 730–733.

[13] Turuani, M. *The CL-Atse protocol analyser. Term Rewriting and Applications*, 2006. 277–286.

3 Introduction

We define *Future AnB*: the Authentication Protocol Specification language APS for the *FutureID* project. This language meets the need for a simple intuitive specification language that is expressive enough to specify the authentication protocols, especially eID protocols, of our project. It serves to specify authentication protocols and enables automated verification with tools like OFMC [7] and ProVerif [4]. It also enables automated translation from a protocol specification to a protocol implementation in Java for instance. This document describes in detail *Future AnB*, a language based on the popular Alice-and-Bob notation (aka message sequence charts, or protocol narrations) and extended from AnB [8]. Plain AnB is one of the input languages of the OFMC tool.

We believe that this language has several advantages over other languages in protocol verification:

- It is simple and easy to use; it enables intuitive one-go modelling for protocols.
- It is possible to connect its specification to a broader variety of verification tools, because it is less focused on a particular paradigm.
- It allows for checking that a protocol is *executable* and derive implementations.
- It introduces an abstraction from the concrete way of structuring plain text (e.g., xml tags) through *formats* that makes it easier for compositional reasoning in WP 2.4.

In the following we provide a brief introduction to the Extended Access Control (EAC) protocol (used in ePassports) that we use as a running example to explain the details of the *Future AnB* language in the remainder of this report. EAC is a two-party protocol aiming at providing mutual authentication. The protocol takes place in the environment of the German ID card. Through EAC (1) a terminal (Proximity Coupling Device, PCD) authenticates itself to the ID card (Proximity Integrated Circuit Card, PICC) to get access to the personal data stored on the card and (2) the German ID Card, i.e., the PICC, proves its authenticity to the PCD. Protocol narrations shown below, though very abstract, gives a brief idea of EAC.

$$
\begin{aligned}
&PCD \rightarrow PICC : \{PCD, PK_{PCD}\}PK_{ca}^{-1}\\
&PICC \rightarrow PCD : R1\\
&PCD \rightarrow PICC : \{PICC, Random1, hash(g^X)\}PK_{PCD}^{-1}\\
&PICC \rightarrow PCD : R2\\
&PCD \rightarrow PICC : g^X\\
&PICC \rightarrow PCD : mac(h3(g^{X.SK_{PICC}}), R3),\\
&\qquad\qquad\qquad\quad \{C, g^{SK_{PICC}}\}PK_{ca}^{-1}, R3
\end{aligned}
$$

The protocol is *narrated* in the form $A \rightarrow B : M$, meaning that entity A sends the a message M to entity B. PCD and PICC in the first and third messages are identifiers for the respective entities. PK_{PCD} denotes the public key and PK_{PCD}^{-1} denotes the corresponding private key for PCD (the same applies for PICC and the Certificate Authority ca). R1, R2 and R3 are fresh nonces.

g^X, $g^{SK_{PICC}}$ are Diffie-Hellman key parts of PCD and PICC respectively (X is fresh secret of PCD and SK_{PICC} is a private key of PICC and $g^{\bullet}$ are the public values).

The rest of this document is structured as follows: Section 4 gives the structure of a protocol specification in *Future AnB*. Then section 5 provides an overview of the *Future AnB* grammar (the detailed grammar is in Appendix A). Section 6 concludes the syntax part of our language by listing its predefined primitives (reserved words). After specifying the syntax, we define a formal semantics for *Future AnB* in Section 7 by providing a translation to an extended version of strands (we call them *detailed* strands) and define a formal semantics for *detailed* strands. We continue with our running example (EAC protocol) to illustrate different aspects of this translation. From *detailed* strands we illustrate how generated implementation (Java code) generation looks like in Section 8. Finally, we specify the translation to AVISPA IF in Section 9.

4 Protocol specification structure

A protocol specification in *Future AnB* consists of eight sections. Each section describes an aspect of the protocol. We give an overview of each section and then explain them using the EAC protocol.

1. Protocol: Gives name of the specified protocol.
2. Types: Declares protocol entities.
3. Mappings: Relates different protocol object to each other e.g., an agent to its public key.
4. Formats: Specifies plain text data structure for composing and decomposing.
5. Macros: Provides abbreviations.
6. Knowledge: Shows the initial knowledge of each *participant* in the protocol.
7. Actions: Specifies exchanged messages and performed tasks in an ideal run of the protocol.
8. Goals: Specifies what the protocol is supposed to achieve, e.g., secrecy of a message.

4.1 Protocol

This section gives the protocol a name (i.e., an identifier) as can be seen in Listing 1. There is no particular significance for this section rather than “naming".

Listing 1: Protocol

```
Protocol: EAC
```

4.2 Types

This section declares protocol identifiers (*constants* or *variables*). Constants (start with a lower-case letter) and variables (start with an upper-case letter) form atomic message components in a protocol and referred as *terms*. A *term* is either an atomic term (constant or variable) or a composed term. Composed term is the application of a signature symbol, e.g., *crypt* or *sign* on a set of arguments that are terms. Signature symbols include mappings, functions, formats, or macros will be discussed later in more details.

In the `Types` section we require the declaration of all protocol variables and constants. As one can see in Listing 2 this section starts with a header `Types:` followed by a set of declaration statements.

Listing 2: Types section

```
Types:
    Agent   PCD, PICC,ca;
    Nonce   IDpicc, Rpicc, Rmac;
    Number X, g;
    ImpData CertDesc, RCHAT, OCHAT, AuxData, RC,
            CHAT, CAR, EFC, NoCert;
```

A declaration statement has the form of:

datatype identifiers;

Where *identifiers* can be either constants (uncapitalized) such as ca and g and variables (capitalized) such as PCD, PICC etc. Variables are instantiated with concrete values during the protocol execution. Furthermore *datatype* is any of the following predefined data types:

1. `Agent`: Declares protocol agents. The `Agent` type is the key data type of specifications and it is special in the sense that constants of this type (called *honest agents* hereafter) represent honest agents whereas variables of this type (called *roles* hereafter) can be either honest or not; since they represent the different *roles* of the protocol and can be instantiated arbitrarily with concrete agent names including the intruder i, i.e., the intruder i can only impersonate roles. i is a predefined constant identifies the intruder. For convention, by agents we mean roles, honest agents, and the intruder i. And by *participants* we mean agents that directly participate in the protocol.

2. `PublicKey`: Declares public keys, although usually we use the mapping *pk()* (discussed later in Section 4.3) to relate agents to their public keys.

3. `PrivateKey`: Declares private keys. We encourage its use as a data type in declaring mappings or formats only, but not in declaring identifiers (in `Types` section) to avoid having unrealistic specification. So we forbid to declare identifiers of type `PrivateKey` in `Types` section.

4. `Number`: Declares identifiers for number.

5. `Nonce`: Declares random numbers typically used as nonces in protocols.

6. `SymmetricKey`: Declares symmetric keys identifiers although *shk()* mapping is more convenient for this purpose as later discussed.

7. `Msg`: A generic data type used to represent the resulting type of applying cryptographic primitives. For example the data type of `sign`(*Key,Message*) is `Msg` (for simplicity especially in declaring formats).

8. `ImpData`: Defines identifiers that are relevant for protocol implementation but not for the abstract model.

Note that variables of data types with the exception of `Agent` represent fresh values that are created by the agent that first uses them. So we forbid the existence of any variables of other data type (other than `Agent`) in `Knowledge` section.

4.3 Mappings

Mappings are meant to relate different objects in the abstract model, i.e., we relate an agent *A* to its public key using a mapping *pk(A)*. We also relate a private key to a public key by another mapping *inv()*. *inv()* works also for many keys (not only those modelled by *pk()*). Those are abstract functions that exist only in the abstract model and have no corresponding implementation in the real/concrete world. Mappings is an optional section that starts with the heading `Mappings :` as shown in Listing 3.

Listing 3: Mappings section

```
Mappings:
    pk  :  Agent  -> PublicKey;
    inv:  PublicKey-> PrivateKey;
    shk  :  Agent  ,  Agent  -> SymmetricKey;
```

A mapping has the form of:

$$identifier : datatypes \to datatype;$$

Where *identifier* is the name of our mapping, e.g., *shk*. *datatypes* is a data type or comma-separated list of them such as `Agent, Agent`. This list reflects one side of the function of the mapping. The other side of the relation is given by a *datatype* (`SymmetricKey` in *shk* case). *shk* relates two agents to a symmetric key (which models the shared key of two agents). Note that for the three of shk(,), pk() and inv() there is no such functions or procedures in reality, but are very helpful for modelling. Moreover, all of them are predefined mappings in *Future AnB* so one can directly use them and will not see them in the full EAC specification in Appendix B. Of course one can define new mappings as needed in `Mapping` section.

Mappings are *private* in a sense that a participant can not calculate a mapping (knowing the public key of another participant does not result in knowing its private key using *inv()*). A participant is given a mapping in his initial knowledge or he get it from a message that he receives.

4.4 Formats

This section declares the structure of plain text data and describes how the contents of messages are composed or parsed. It relates abstract syntax to concrete messages of the protocol by modelling the fact that concrete messages are of different formats and that they are meant to be different. It gives an abstraction of the concrete plain text messages structure (e.g., xml tags). On another hand, the modelling of message distinction (on basis of composing and decomposing) that formats serves makes it easier for compositional reasoning in WP 2.4. We encourage the use of formats for plain text structure since concatenating (or pairing) of plain messages is too abstract to generate implementations from.

As seen in Listing 4, we have eight formats that are used later as messages exchanged between the two participants of EAC.

Listing 4: Formats section

```
Formats:
  eac1input(Msg, ImpData, ImpData, ImpData, ImpData);
  eac1output(ImpData, ImpData, ImpData, ImpData, Agent,
    Nonce);
  eac2input(ImpData, Number);
  eac2output(Nonce);
  eac22output(Msg, Msg, Nonce);
  eac2additionalinput(Msg);
  certForm(Agent, PublicKey);
  x59d(Agent, Nonce, PublicKey);
```

The form of each of the *formats* is:

$$FormatName(data\ types)$$

FormatName, e.g., *eac1input*, is a format identifier followed by a list of data types (Msg, ImpData ...).

Every format (once declared) is considered as a new data type that can be used in other formats declaration e.g., having the formats of Listing 4, we can declare a new format (say *anotherFormat*) as follows:

```
anotherFormat(eac1input, ImpData, Nonce);
```

in which we use *eac1input* as a data type to declare *anotherFormat*.

Formats are *public* in the sense that all participants can construct them or parse their content. They are also considered disjoint in the sense that no format will be handled as another one.

4.5 Macros

This section provides abbreviations to improve readability and to prevent repetition that happens often in protocol specifications. Macros can be simply replaced by unfolding them in the specification wherever they occur.

Listing 5: Macros section

```
Macros:
```

```
cert(A,K,CA)=sign(inv(pk(CA)),certForm(A,K))
kdf(SEC, PUB, NONCE)=hash(exp(PUB, SEC), NONCE)
```

Notice that a macro is formed as:

$$MacroName(identifiers) = message$$

Referring to our running example, the cert(A,K,CA) macro in Listing 5 is applying the cryptographic primitive `sign` on a *format certFrom*. In macro declaration, variables on the right hand side must appear on the left had side of the equal sign (macro variables must be bound).

We also support `let` statements as another abbreviation possibility that can be used in the `Actions` section. We discuss `let` statements in more detail when we reach that section.

4.6 Knowledge

In this section we describe the initial knowledge of all *participants*. A *participant* is an agent that directly participates in the protocol (i.e., exchange messages or perform tasks). There is no need to specify the initial knowledge of an agent who does not directly participate in the protocol, such as a *certificate authority (ca)* in EAC because issuing and distribution of certificates is not part of EAC, and *ca* it is not involved in direct interactions with other *participants*. Listing 6 shows the initial knowledge for the two participants PCD and PICC.

Listing 6: Knowledge section

```
Knowledge:
  PCD:  PICC, PCD,cert(PCD,pk(PCD),ca), pk(PCD),
        pk(ca), inv(pk(PCD)), g;
  PICC: PCD, PICC,cert(PICC,exp(g,sk(PICC)),ca),
        pk(ca), sk(PICC), g;

  where PCD!=ca, PICC!=ca
```

As one can see in Listing 6, the initial knowledge of a participant can be any *message* (atomic or composed term). For each participant we give its initial knowledge in the form:

$$participant : knowledge;$$

Where *knowledge* is a message or comma-separated list of them. We only allow variables of type `Agent`, i.e., roles. Variables of other types do not occur in the initial knowledge since they represent values that are freshly created by the agent who first uses them. Note that only the variables PCD and PICC occur in the initial knowledge and both are roles. A participant can not have any fresh knowledge initially since fresh values is what an agent creates during message exchange.

At the end of this section we can add constraints to *roles* instantiation in the form of:

$$\texttt{where}\ A \mathrel{!=} B$$

Where A and B are any participants (including the *intruder i*). Notice that we have two constraints at the end of the `Knowledge` section of the EAC protocol in Listing 6. Having PCD!=ca prevents PCD to be instantiated to ca in an execution.

4.7 Actions

The actions section is the core section of the protocol specification. It models how different participants exchange messages in an ideal run of a protocol run defined by a set of actions.

Listing 7: Actions section

```
Actions (Main):
 [PCD]*->*[PICC]: eac1input(cert(PCD,pk(PCD),ca),
          CertDesc, RCHAT, OCHAT, AuxData)

 [PICC]*->*[PCD]:eac1output(RC, CHAT, CAR,
          EFC, PICC, IDpicc)

let PK_PCD=exp(g,X)

 [PCD]*->*[PICC]:eac2input(NoCert, PK_PCD)

 [PICC]*->*[PCD]: eac2output(Rpicc)

 [PCD]*->*[PICC]:eac2additionalinput(sign(inv(pk(PCD))
          ,x59d(PICC, Rpicc, PK_PCD)))

 let PK_PICC=exp(g,sk(PICC))
 let K=kdf(sk(PICC),PK_PCD, Rmac)
 let Tpicc=mac(K, PK_PCD)

 [PICC]*->*[PCD]: eac22output(cert(PICC,PK_PICC,ca),
          Tpicc, Rmac)
```

Actions are mainly message exchanging as shown in Listing 7. We also consider further actions that we call *tasks* including selecting structures and sub-protocol calling. We give further explanation for different types of actions in the following:

- Message exchange: A participant sending a *Message* to another participant. A *Message* can be any term (discussed earlier in Types section). Message exchange form is:

  $$A\ Channel\ B:\ Messages$$

  Both *A* and *B* are participants. We distinguish between four different types of channels:

1. Insecure channel: $A \to B : M$ represents the default insecure channel from A to B, controlled by the intruder. Intruder can read, send under any sender's name, and intercept messages.
2. Authentic channel: $A \bullet\!\!\to B : M$ represents an authentic channel from A to B. Here B is guaranteed that the message is sent from A (and meant for B). However, the intruder can see the message M.
3. Confidential channel: $A \to\!\!\bullet B : M$ means that A has the guarantee that only the intended receiver B can see the message M. However, B has no guarantee of authenticity.
4. Secure Channel: $A \bullet\!\!\to\!\!\bullet B : M$ represents an authentic and confidential channel.
5. Pseudonymous channels: $[A]_\psi \bullet\!\!\to\!\!\bullet B : M$ and $B \bullet\!\!\to\!\!\bullet [A]_\psi : M$. This represents a secure channel, but with an unauthenticated party A that acts under pseudonym ψ. This is different from a channel where the end-point A is simply not secured, i.e., $A \to\!\!\bullet B$ or $B \bullet\!\!\to A$, because the channel is bound to pseudonym ψ. The idea is to model channels like the ones we get from TLS without client authentication: we have a secure line between a client and a server, but the identity of the client is not proved. However, an intruder cannot hack into this line any more. This is crucial when the client uses the channel for a login to authenticate itself. We also allow to drop the notation ψ of the pseudonym if not relevant for the protocol: it then means that at the beginning of each protocol run, each pseudonymous users picks a fresh pseudonym to use throughout the session.
6. Mutual pseudonymous channel $[A]_\psi \bullet\!\!\to\!\!\bullet [B]\varphi : M$ represents a secure channel with both parties unauthenticated to each other. Each participant is under a pseudonym (A under ψ and B under φ that can be both dropped as in the previous channel type). Note that we use this channel type in our example shown in Listing 7 for all channels between PICC and PCD. We have a card and a card reader that can be faked by the intruder but still the intruder can not sit between them.

- Tasks: Tasks are mainly control structure actions that break the linear structure of protocols in classic *AnB* and database manipulation actions. Actions of this type have the form of:

  $$A : Task$$

  Where A is a role and *Task* can be any of the following (in order of priority and significance):

  – Selection structures: We have two selection structures to change the sequential message interchange of a protocol. The first one is the non-deterministic `OR` in which the participant can select any of the two options unconditionally. The second type of selection is the conditional/ deterministic if-else statements in which an agent acts according to a condition.

  – Sub-protocol call: The sub-protocol notion emphasizes specification flexibility, reusability as well as the ability to provide non-linear control flow of protocols. Accordingly, we will allow several `Actions` sections each per sub-protocol and one as `Main`. Sub-protocols are supposed to behave like methods in programming languages. The termination of sub-protocols can be done either explicitly using `return` to return a value or `exit` without a return value.

  – Database manipulation: Including add, delete and query a database.

Tasks are not considered essential for our purposes for the time being; we plan to integrate it to *Future AnB* but with less priority in implementation.

We also define `let` statements as one can see in 7 and earlier mentioned. A `let` statement is a non-parametrized abbreviation method (with no arguments as opposed to macros). It is placed in the `Actions` section and provides a local abbreviation i.e., applicable on statements below it.

4.8 Goals

This section specifies the goals that the protocol is supposed to achieve and that the verification tools should check. We have three types of goals:

- M `secret of` *list*: specifying a *list* of agents who are cleared to know the secret M. It counts as an attack if the intruder i finds out M and he is not in the *list*.
- B `weakly authenticates` A `on` M: If B receives the message M, he can rely on the fact that it was really sent by A, and was meant for him.
- B `authenticates` A `on` M: Same as the weak authentication variant but a replay counts as an attack.

Even though this list of goals is basic, there is a surprisingly large number of security problems that can be specified with just these goals.

Listing 8: Goals section

```
Goals:
    PICC    authenticates PCD on Tpicc
    PCD     authenticates PICC on Tpicc
    Rpicc   secret of PICC, PCD
    K   secret of PICC, PCD
```

Note that the first two goals represent the mutual authentication between PCD and PICC that EAC is supposed to achieve.

5 *Future AnB* Grammar

This section gives an overview of *Future AnB* Grammar. The entire EBNF grammar of *Future AnB* syntax is in Appendix A.

```
⟨Protocol⟩ ::= [⟨ProtocolName⟩]⟨Types⟩[⟨Mappings⟩][⟨Formats⟩]
      [⟨Macros⟩]⟨Knowledge⟩ ⟨Actions⟩⟨Goals⟩

⟨ProtocolName⟩ ::= 'Protocol''':' ⟨Ident⟩';'

⟨Types⟩ ::= 'Types''':'
      (⟨Type⟩ ⟨Ident⟩ ( ',' ⟨Ident⟩)* ';')+

⟨Type⟩ ::= 'Agent' | 'Nonce' | 'PublicKey'
      | 'Number' | 'SymmetricKey' | 'ImpData'
      | 'PrivateKey'| 'Msg'

⟨Mappings⟩ ::= 'Mappings''':'
      (⟨Ident⟩ ':'⟨Type⟩ (',' ⟨Type⟩)* '->' ⟨Type⟩';')+

⟨Formats⟩ ::= 'Formats''':'
      (⟨Ident⟩ '(' ⟨Ident⟩(','⟨Ident⟩)* ')' = ⟨Msg⟩ ';')+

⟨Macros⟩ ::= 'Macros''':'
      (⟨Ident⟩ '(' ⟨Ident⟩(','⟨Ident⟩)* ')' = ⟨Msg⟩ ';')+

⟨Knowledge⟩ ::= 'Knowledge''':'
      (⟨Agent⟩ ':' ⟨Msg⟩;)+
      ['where' ⟨Agent⟩ '!=' ⟨Agent⟩ (','⟨Agent⟩)*
      (,⟨Agent⟩ '!=' ⟨Agent⟩ (','⟨Agent⟩)*)* ]

⟨Actions⟩ ::= 'Actions''('Main'')'':' (S)+
      ('Actions('⟨Type⟩ ⟨Ident⟩
      '('⟨Type⟩ ⟨Ident⟩ (','⟨Type⟩ ⟨Ident⟩)* '):' (S)+)*

⟨S⟩ ::= (⟨Agent⟩ ⟨Channel⟩ ⟨Agent⟩ : ⟨Msg⟩
      | ⟨Agent⟩ ':' ⟨Task⟩ | 'let' ⟨Ident⟩ '=' ⟨Msg⟩)';'
```

$\langle \mathit{Task} \rangle$ ::= ‘`if`’‘`(`’$\langle \mathit{Condition} \rangle$‘`)`’‘`then`’ $\langle S \rangle$ ‘`else`’ $\langle S \rangle$ ‘`fi`’
| $\langle S \rangle$ ‘`OR`’ $\langle S \rangle$

$\langle \mathit{Goals} \rangle$ ::= ‘`Goals`’‘`:`’ $\langle \mathit{Goal} \rangle^{+}$

$\langle \mathit{Goal} \rangle$ ::= $\langle \mathit{Agent} \rangle$ ‘`authenticates`’ $\langle \mathit{Agent} \rangle$ ‘`on`’ $\langle \mathit{Msg} \rangle$
| $\langle \mathit{Agent} \rangle$ ‘`weakly authenticates`’ $\langle \mathit{Agent} \rangle$ ‘`on`’ $\langle \mathit{Msg} \rangle$
| $\langle \mathit{Msg} \rangle$ ‘`secret of`’ $\langle \mathit{Agent} \rangle$ (‘`,`’$\langle \mathit{Agent} \rangle$)*

6 Predefined primitives

In this section we list the predefined primitives that are considered as keywords in *Future AnB*:

1. Data types: As discussed earlier in Section 4.2, we have these data types: `Agent`, `PublicKey`, `PrivateKey`, `Number`, `Nonce`, `SymmetricKey`, `ImpData`, `Msg`. We will refer to this set as T={`Agent`,`PublicKey`,...} later in this document.

2. Functions (public):

   - Symmetric key encryption: `scrypt(K,M)`
   - Asymmetric key encryption: `crypt(K,M)`
   - Signing: `sign(K, M)`
   - Exponentiation: `exp(A,B)`
   - Hashing: `hash(M)`
   - Message authentication code: `mac(A,B)`
   - Multiplication: `mult(A,B)`
   - Decryption and verifying of signatures are also public but they are not supposed to appear in specifications, but are implicit in the *Future AnB* description. This is shown in the semantics where the decryption operation (as well as) becomes explicit in the execution model.

3. Mappings (private):

   - pk: Maps an agent to a public key. For example pk(A) gives the public key of the agent A.
   - inv: Maps a public key to a private key. inv(pk(A)) gives the private key of A.
   - shk: Maps two agents to a relation of both, it can be used to represent a key shared between them.

7 Semantics

Protocol specification in AnB, as well as *Future AnB*, describes how different participants behave in an ideal protocol execution. This behaviour is defined by an initial state and a transition relation on states forming a finite-state transition system. Reachable states represent what a participant can do, and goals are achieved only if their corresponding attack states are not be reachable. The formal semantics of *Future AnB* is defined by translating its specification into an extension of strands that we define a formal semantic for.

7.1 Message Model

The core of a protocol specification in *Future AnB* consists of messages exchanging and more specifically messages handling (parsing and composing).

A *Future AnB* specification does not give explicit details about how participants compose or decompose messages. In general, a participant can receive any message whether he can parse it or not (checking messages will be discussed later). However, he can not send a message *Msg* unless he is able to construct that message by these means:

- *Msg* is in his initial knowledge.
- He received *Msg* so he can forward it (we can say that *Msg* is in his *updated* knowledge).
- He derives *Msg* by composing or decomposing it using deduction rules based on Dolev-Yao intruder model [9] and algebraic properties of public functions, mappings, and formats.

We formally define *Future AnB* message model based on [8] as follows.

Definition 1. *A message model* (Σ_c, Σ_d, Σ_f, Σ_m, Σ_i, V, T, Σ, $\mathcal{V}$, $\approx$) *consists of:*

- Σ_c: *a finite set of predefined public constructor symbols with arities that are protocol independent.*

$$\Sigma_c = \{\texttt{scrypt}/2, \texttt{crypt}/2, \texttt{sign}/2, \texttt{mult}/2, \texttt{exp}/2, \texttt{hash}/1, \texttt{mac}/2\}$$

- Σ_d: *a finite set of predefined public destructor symbols with arities that are protocol independent. The modeller can not use any element of Σ_d in the APS specification. They are instead introduced later by a translation into an executable program formal model used for message parsing and checking.*

$$\Sigma_d \supseteq \{\texttt{sdecrypt}/2, \texttt{vscrypt}/2, \texttt{decrypt}/2, \texttt{vcrypt}/2, \texttt{vsign}/2\}$$

- Σ_i: *a countable set of protocol constants. Mainly user-defined constants but also contains special reserved constants such as the intruder name i as well as $\top$.*

$$\Sigma_i \supseteq \{\mathsf{i}, \top\}$$

- *V: a countable set of protocol variables (user-defined).*
- *Σ_m: a finite set of predefined and user-defined mapping symbols with arities.*

$$\Sigma_m \supseteq \{\texttt{inv}/1, \texttt{pk}/1, \texttt{shk}/2\}$$

- *Σ_f: a finite set of user-defined protocol formats symbols with arities. For every $f \in \Sigma_f$ with arity $n \geq i$, we have:*
  - $\texttt{get}_{i,f}(t_1, ..., t_n) \in \Sigma_d$
  - $\texttt{verify}_f(t_1, ..., t_n) \in \Sigma_d$
- *Σ: a countable set of all identifiers (pre-defined and user-defined, i.e., protocol dependent and independent).*

$$\Sigma = \Sigma_c \cup \Sigma_d \cup \Sigma_i \cup \Sigma_m \cup \Sigma_f$$

- *T: a finite set of predefined data types.*

$$T = \{\texttt{Agent}, \texttt{Nonce}, \texttt{Number}, \texttt{ImpData}, \texttt{PublicKey}, \texttt{SymmetricKey}, \texttt{Msg}, \texttt{PrivateKey}\}$$

- *$\mathcal{V}$: a countable set of variable symbols disjoint from Σ.*

$$\mathcal{V} = \{\mathcal{X}_0,\ \mathcal{X}_1, ...\}$$

- *$\approx$: a congruence relation over ground terms over Σ defined below by a set of properties.*

$\Sigma_c, \Sigma_d, \Sigma_i, \Sigma_m, \Sigma_f, V, \mathcal{V}$ are all pairwise disjoint. A term t over Σ is denoted by $t \in \mathcal{T}_\Sigma(V)$. $\mathcal{T}_\Sigma(V)$ is defined as follows:

- $t \in V \Rightarrow t \in \mathcal{T}_\Sigma(V)$
- *$t_1, .., t_n \in \mathcal{T}_\Sigma(V), f \in \Sigma, f$ is of arity $n \ \Rightarrow f(t_1, .., t_n) \in \mathcal{T}_\Sigma(V)$, note that for arity 0 we omit braces.*
- *Nothing else is in $\mathcal{T}_\Sigma(V)$*

A term *t over Σ is a variable $t \in V$ or an expression in the form $f(t_1, t_2, .., t_n)$ where $f \in \Sigma$ and the t_i are terms. A* ground term *is a term t without variables denoted by $t \in \mathcal{T}_\Sigma$, i.e., $V = \emptyset$.*

A labelled message *t^l is a term $t \in \mathcal{T}_\Sigma(V)$ labelled by a term $l \in \ \mathcal{T}_\Sigma(\mathcal{V})$. For a set of labelled messages M, we define the following deduction rules:*

$$\frac{}{M \vdash m^l}\ m^l \in M \qquad (1)$$

$$\frac{M \vdash t_1^{l_1}\ \ ...\ M \vdash t_n^{l_n}}{M \vdash f(t_1, ..., t_n)^{f(l_1, ..., l_n)}}\ f \in \Sigma_c \cup \Sigma_d \cup \Sigma_f \qquad (2)$$

The first rule expresses that terms given in the initial knowledge can be deduced directly. The second rule expresses that the deduction of messages is closed under the application of public operations (constructors Σ_c and destructors Σ_d) as well as formats.

Finally, the we define $\approx$ as the least congruence relation that satisfies these properties:

$$\mathtt{sdecrypt}(k,\ \mathtt{scrypt}(k,m)) \approx m \tag{3}$$
$$\mathtt{vscrypt}(k,\mathtt{scrypt}(k,m)) \approx \top \tag{4}$$
$$\mathtt{decrypt}(\mathtt{inv}(k),\ \mathtt{crypt}(k,m)) \approx m \tag{5}$$
$$\mathtt{vcrypt}(\mathtt{inv}(k),\mathtt{crypt}(k,m)) \approx \top \tag{6}$$
$$\mathtt{open}(\mathtt{sign}(k,m)) \approx m \tag{7}$$
$$\mathtt{vsign}(k,\mathtt{sign}(\mathtt{inv}(k),m)) \approx \top \tag{8}$$
$$\textit{For every } f \in \Sigma_f \textit{ with arity } n \geq i \textit{ we have } (9) \textit{ and } (10):$$
$$\mathtt{get}_{i,f}(f(t_1,...,t_n)) \approx t_i \tag{9}$$
$$\mathtt{verify}_f(f(t_1,...,t_n)) \approx \top \tag{10}$$
$$\mathtt{proj_i}(\ \mathtt{m_1},\ \mathtt{m_2},..,\ \mathtt{m_n}) \approx m_i \tag{11}$$
$$\mathtt{exp}(\mathtt{exp}(\mathtt{A},\ \mathtt{B}),\ \mathtt{C})) \approx \mathtt{exp}(\mathtt{exp}(\mathtt{A},\ \mathtt{C}),\ \mathtt{B})) \tag{12}$$
$$\mathtt{mult}(\mathtt{A},\ \mathtt{B}) \approx \mathtt{mult}(\mathtt{B},\ \mathtt{A}) \tag{13}$$
$$\mathtt{mult}(\mathtt{A},\ \mathtt{mult}(\mathtt{B},\mathtt{C})) \approx \mathtt{mult}(\mathtt{mult}(\mathtt{A},\mathtt{B}),\ \mathtt{C}) \tag{14}$$

For protocol specification, the modeller can not use any of Σ_d since they are implicitly used in analysis but not explicitly in specification.

7.2 Translation preprocessing

Before we perform translation form *Future AnB* to *detailed strands*, we perform the following preprocessing steps:

1. Knowledge packing per participant: We specify per participant its initial knowledge. The main part of participant knowledge is given directly from `Knowledge` section in the specification. The only allowed variables in the initial knowledge are *roles.*

2. Fresh variables: We run a pass through the specification to determine fresh variables within it and add them to the initial knowledge of its creator participant(owner) with distinction from other initial knowledge items. The creator/owner participant is the participant that first uses that variable and thus a fresh variable has a unique owner.

3. Unfold macros and `let` statements: As macros and `let` statements are just for abbreviation and have no semantic significance, we perform a preprocessing step of unfolding macros and `let` statements. i.e., we simply replace them wherever they occur in the specification with what they abbreviate.

4. Channel preprocessing: Since we support different channel types in protocol specification, we perform a channel preprocessing step in which we model channels via cryptographic operations, e.g., signing for authentic channels, see [10].

7.3 *Detailed strands*

We use an extension of strands to represent the semantics. It is well suited for translating to the input languages of many tools like AVISPA and ProVerif as well as for generating implementation. Detailed strands are based on strands of [1] and [11] with introduction to new strand structures and extra details within strands. Detailed strands are meant to:

- Give a separate program per participant.
- Pack all required details of a protocol run from each peer view as one piece including reasoning for message composing and parsing.
- Provide a comprehensive version for each participant for further translation to targeted low-level languages or implementation.

In this section we give the syntax and semantics of the *detailed* strands based on [1], and [11].

7.3.1 *Detailed* strands syntax

We start with *detailed* strands syntax and give the structure of strands followed by a graphical syntax. Strand textual representation is as following:

$$Strand0 = Knowledge\ Strand$$

$$Strand = \ Snd\ Msg\ Strand$$

$$|\ Rcv\ Msg\ Strand$$

$$|\ If\ Fact\ Strand\ Strand\ Strand$$

$$|\ If\ not\ Fact\ Strand\ Strand\ Strand$$

$$|\ Or\ Strand\ Strand\ Strand$$

$$|\ Check\ Msg\ Msg\ Strand$$

$$|\ Add\ Fact\ Strand$$

$$|\ Delete\ Fact\ Strand$$

$$|\ Query\ Fact\ Strand$$

$$|\ EOS\ Owner\ ID$$

$$\begin{aligned} Fact &= FactSymbol(Msg) \\ FactSymbol &= \texttt{witness} \mid \texttt{contains} \mid \texttt{request} \mid \\ &\quad \texttt{secret} \mid \texttt{wrequest} \mid \texttt{iknows} \end{aligned}$$

Where *Knowledge* is declared by $Knowledge : \mathcal{V} \mapsto \mathcal{T}_{\Sigma}(V)$ and represents the initial knowledge of the strand owner. *Msg* is constructed as in the EBNF grammar in Appendix A. *Owner* denotes unique identifier for strand owner and *ID* denotes session identifier at execution time. We have a restriction on messages (*Msg*) in *Query* and *Rcv* that they may contain free variables that will be bound. Whereas messages in *Delete* and *Snd* can not have free variables (in a sense that one can not delete or send something that is not specified), they can only use parameters and previously bound variables. We formally define $fv(\bullet) : Strand \mapsto 2^V$ and $fv_m(\bullet) : Msg \mapsto 2^V$. As follows (we here give the most interesting cases, the others as as expected):

$$\begin{aligned} fv_m(c) &= \emptyset \quad c \textit{ is constant} \\ fv_m(X) &= \{X\} \quad x \in V \\ fv_m(f(t_1, ..t_n)) &= \bigcup^i fv_m(t_i) \end{aligned}$$

$$fv(Snd\ Msg\ Rest) = fv_m(Msg) \cup fv(Rest)$$

$$fv(Rcv\ Msg\ Rest) = fv(Rest) \setminus fv(Msg)$$

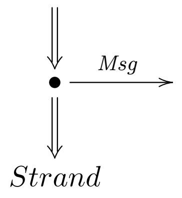
$$fv(Query\ Fact\ Rest) = fv(Rest) \setminus fv_m(Fact)$$

$$fv(Delete\ Fact\ Rest) = fv_m(Fact) \cup fv(Rest)$$

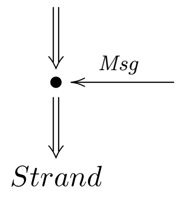
In all other cases it is just as in *Snd* case.

We use this graphical syntax for strands for clarity in coming steps:

- Snd *Msg* Strand:



- Rcv *Msg* Strand:



- If Fact Strand Strand Strand:

$$\begin{tikzcd}
 & {} \arrow[d, Rightarrow] & \\
 & \boxed{Fact} \arrow[dl, Rightarrow] \arrow[dr, Rightarrow] & \\
Strand \arrow[dr, Rightarrow] & & Strand \arrow[dl, Rightarrow] \\
 & Strand \arrow[d, Rightarrow] & \\
 & {} &
\end{tikzcd}$$

- If not Fact Strand Strand Strand:

$$\begin{tikzcd}
 & {} \arrow[d, Rightarrow] & \\
 & \boxed{not(Fact)} \arrow[dl, Rightarrow] \arrow[dr, Rightarrow] & \\
Strand \arrow[dr, Rightarrow] & & Strand \arrow[dl, Rightarrow] \\
 & Strand \arrow[d, Rightarrow] & \\
 & {} &
\end{tikzcd}$$

- Or Strand Strand Strand:

$$\begin{tikzcd}
 & {} \arrow[d, Rightarrow] & \\
 & \bullet \arrow[dl, Rightarrow] \arrow[dr, Rightarrow] & \\
Strand \arrow[dr, Rightarrow] & & Strand \arrow[dl, Rightarrow] \\
 & Strand \arrow[d, Rightarrow] & \\
 & {} &
\end{tikzcd}$$

- Check Msg Msg Strand:

$$\begin{tikzcd}
{} \arrow[d, Rightarrow] \\
\boxed{Msg == Msg} \arrow[d, Rightarrow] \\
{}
\end{tikzcd}$$

- Add Fact Strand:

$$\begin{tikzcd}
{} \arrow[d, Rightarrow] \\
\boxed{+\ Fact} \arrow[d, Rightarrow] \\
{}
\end{tikzcd}$$

- Delete Fact Strand:

$$\begin{tikzcd}
{} \arrow[d, Rightarrow] \\
\boxed{-\ Fact} \arrow[d, Rightarrow] \\
{}
\end{tikzcd}$$

- Query Fact Strand:

$$\begin{tikzcd}
{} \arrow[d, Rightarrow] \\
\boxed{?\ Fact} \arrow[d, Rightarrow] \\
{}
\end{tikzcd}$$

- EOS Owner SessionID:

$$\begin{tikzcd}
{} \arrow[d, Rightarrow] \\
\underline{EOS:\ Owner:\ SessionID}
\end{tikzcd}$$

Note that this strand denotes the end of a strand.

7.3.2 *Detailed* strands semantics

As introduced in [1], a strand is a sequence of actions that a participant makes during protocol execution. A strand reflects the protocol from a participant view. Strands define the behaviour of participants of a protocol by means of an infinite-state transition system defined by an infinite state and a transition relation on states. Reachable states represent what the system can do. The initial state is represented by the set of strands for all participants and the initial knowledge instantiated with concrete values for parameters (roles) and fresh values for the required number of sessions. The transition relation is defined by these transition rules:

$$\bullet\{Snd\ \ Msg.Rest\} \cup\ \ Strands; M; Facts \Longrightarrow \{Rest\} \cup\ \ Strands; M \cup \{Msg\}; Facts$$

$$\bullet\{Rcv\ \ Msg.Rest\} \cup\ \ Strands; M; Facts$$
$$\Longrightarrow \{\sigma(Rest)\} \cup\ \ Strands; M; Facts\ \ if\ \ exists\ \ a\ \ substitution\ \ \sigma\ \ s.t.\ \ \sigma(Msg) \in M$$

$$\bullet\{OR\ \ Rest1\ \ Rest2\} \cup\ \ Strands; M; Facts \Longrightarrow \{Rest1\} \cup\ \ Strands; M; Facts$$

$$\bullet\{OR\ \ Rest1\ \ Rest2\} \cup\ \ Strands; M; Facts \Longrightarrow \{Rest2\} \cup\ \ Strands; M; Facts$$

$$\bullet IF\ \ Fact\ \ RT\ \ RF.Rest\} \cup\ \ Strands; M; Facts$$
$$\Longrightarrow \{\sigma(RT.Rest)\} \cup\ \ Strands; M; Facts\ \ \ if\ \ exists\ \ \sigma.\ \sigma(Fact) \in Facts$$

$$\bullet\{IF\ \ Fact\ \ RT\ \ RF.Rest\} \cup\ \ Strands; M; Facts$$
$$\Longrightarrow \{RF.Rest\} \cup\ \ Strands; M; Facts\ \ if\ \ for\ \ all\ \ \sigma.\ \sigma(Fact) \notin Facts$$

$$\bullet\{Add\ \ Fact.Rest\} \cup\ \ Strands; M; Facts \Longrightarrow \{Rest\} \cup\ \ Strands; M; Facts \cup \{Fact\}$$

$$\bullet\{Delete\ \ Fact.Rest\} \cup Strands; M; Facts \Longrightarrow \{Rest\} \cup Strands; M; Facts = Facts \setminus \{Fact\}$$

$$\bullet\{Query\ \ Fact.Rest\} \cup\ \ Strands; M; Facts$$
$$\Longrightarrow \{\sigma(Rest)\} \cup\ \ Strands; M; Facts\ \ \ if\ \ exists\ \ \sigma.\ \sigma(Fact) \in Facts$$

$$\bullet\{Check\ \ s\ \ t.Rest\} \cup\ \ Strands; M; Facts \Longrightarrow \{Rest\} \cup\ \ Strands; M; Facts$$

Goals are added to strands as facts and attack rules are appended to the transition rules accordingly. Since we have three different types of goals that can be specified in the `Goals` section we describe each one and what to append in strands and in transition rules:

1. `A authenticates B on Msg`: At the end of the A's strand we add the fact `request(`A`,` B`,` Msg`,` $GoalID$`,` $SessionID$`)` s.t., *A,B, and Msg* are derived directly from the goal. *GoalID* is a unique constant identifies the goal message to avoid confusing it with other goals. And finally *SessionID* is a variable for session identifier in execution time.On B's side we add the fact `witness(`$\mathcal{X}_B, \mathcal{X}_A, Msg, \mathcal{X}_{Msg}$`)`. This fact must be added once B is able to construct *Msg*, but we add it at the beginning of B's strand and later check the strand to relocate it properly. For transition rules, we add two rules: The first for authentication and the other for detecting replay. The first rule is as follows:

   request(A,B,Msg,G,S).not(witness(B,A,Msg,G,S)).A$\not\approx$ i$\Longrightarrow$ attack

   The reply detection rule depends on S as follows:

   request(A,B,Msg,G,S).request(A,B,Msg,G,S'). A$\not\approx$i.S$\not\approx$S'$\Longrightarrow$ attack

2. `A weakly authenticates B on Msg`: The same as previous one, but without the replay detection rule.

3. `Msg secret of` A_i: Having that A_i is a list of agents, we add at the end of each agent in the list A_i the facts:

   secret(Msg, A_i).contains(A_i, A_1)....contains(A_i, A_n) where Msg is the secret, $A_1...A_n$ are the agents in the list A_i.

   We add this transition rule to describe attack:

   secret(Msg, A_i). iknows(Msg). not(contains(A_i,i)) $\Longrightarrow$ attack

7.3.3 Translation *Future AnB* to *detailed* strands

We define the function $tr_{Str} : Agent, Agent, Action \rightarrow Strand$, it translates from *Future AnB* specification to strands. A typical application of tr_{Str} is of the form $tr_{Str}(A, B, Action)$ where A is the participant that the strand is being generated for (we call it strand owner since each participant has a strand), B is the active participant (initially the participant sending the first message in `Actions` section and then the one receiving last message sent), and *Action* is the *Future AnB* sequence of actions to be translated. tr_{Str} is recursively defined as follows:

1. In this case we are generating the strand of A and A is active and sending message *msg* to B we have:

$$tr_{Str}\left(A, A, \begin{array}{c} A \rightarrow B : msg \\ Rest \end{array}\right) = \begin{array}{c} Snd\ \ msg \\ \\ tr_{Str}(A, B,\ \ Rest) \end{array}$$

2. This is a generic case in which we are generating the strand of A. B is active, and C sends a message *msg* to D as follows.

$$tr_{Str}\left(A, B, \begin{matrix} C \to D : msg \\ Rest \end{matrix}\right) = \qquad error$$

Unless B $\neq$ C (regardless A and D) then this causes an error; since the participant is taking the action (C) is not the active participant (B). This case can be extended to tasks as well (having the same constraint of B$\neq$ C) as follows:

$$tr_{Str}\left(A, B, \begin{matrix} C : task \\ Rest \end{matrix}\right) = \qquad error$$

3. We are translating for A, B is the active participant and sending *msg* to A.

$$tr_{Str}\left(A, B, \begin{matrix} B \to A : msg \\ Rest \end{matrix}\right) = \qquad \begin{matrix} Rcv\ \ msg \\ \\ tr_{Str}(A, A, Rest) \end{matrix}$$

4. We are translating for A, B is the active participant sending *msg* for another participant C.

$$tr_{Str}\left(A, B, \begin{matrix} B \to C : msg \\ Rest \end{matrix}\right) = \qquad tr_{Str}(A, C, Rest)$$

5. If-else statement, we translate for A that is active and making a conditional selection (if-else).

$$tr_{Str}\left(A, A, \begin{matrix} A : if(fact) \\ then\ RT \\ else\ RF\ fi \\ Rest \end{matrix}\right) = \qquad \begin{matrix} IF\ \ (fact)\ tr_{Str}(A, A, RT)\ tr_{Str}(A, A, RF) \\ \\ tr_{Str}(A, A, Rest) \end{matrix}$$

6. If-else statement where we translate for the participant that is not making the decision, i.e., A is active but we generate the strand of B. Note that it is translated to an OR selection denoting that since A is making the selection, B is open to any choice A may

choose.

$$tr_{Str}\left(B, A, \begin{array}{c} A : if(fact) \\ then\ RT \\ else\ RF\ fi \\ Rest \end{array}\right) = \quad OR\ tr_{Str}(B, A, RT)\ tr_{Str}(B, A, RF)$$

$$tr_{Str}(B, A, Rest)$$

7. OR selection statement

$$tr_{Str}\left(A, A, \begin{array}{c} A : ChoiceL \\ OR \\ ChoiceR \\ Rest \end{array}\right) = OR\quad tr_{Str}(A, A, ChoiseL)\quad tr_{Str}(A, A, ChoiseR)$$

$$tr_{Str}(A, A, Rest)$$

8. OR selection statement

$$tr_{Str}\left(A, A, \begin{array}{c} A : ChoiceL \\ OR \\ ChoiceR \\ Rest \end{array}\right) = OR\quad tr_{Str}(B, A, ChoiseL)\quad tr_{Str}(B, A, ChoiseR)$$

$$tr_{Str}(B, A, Rest)$$

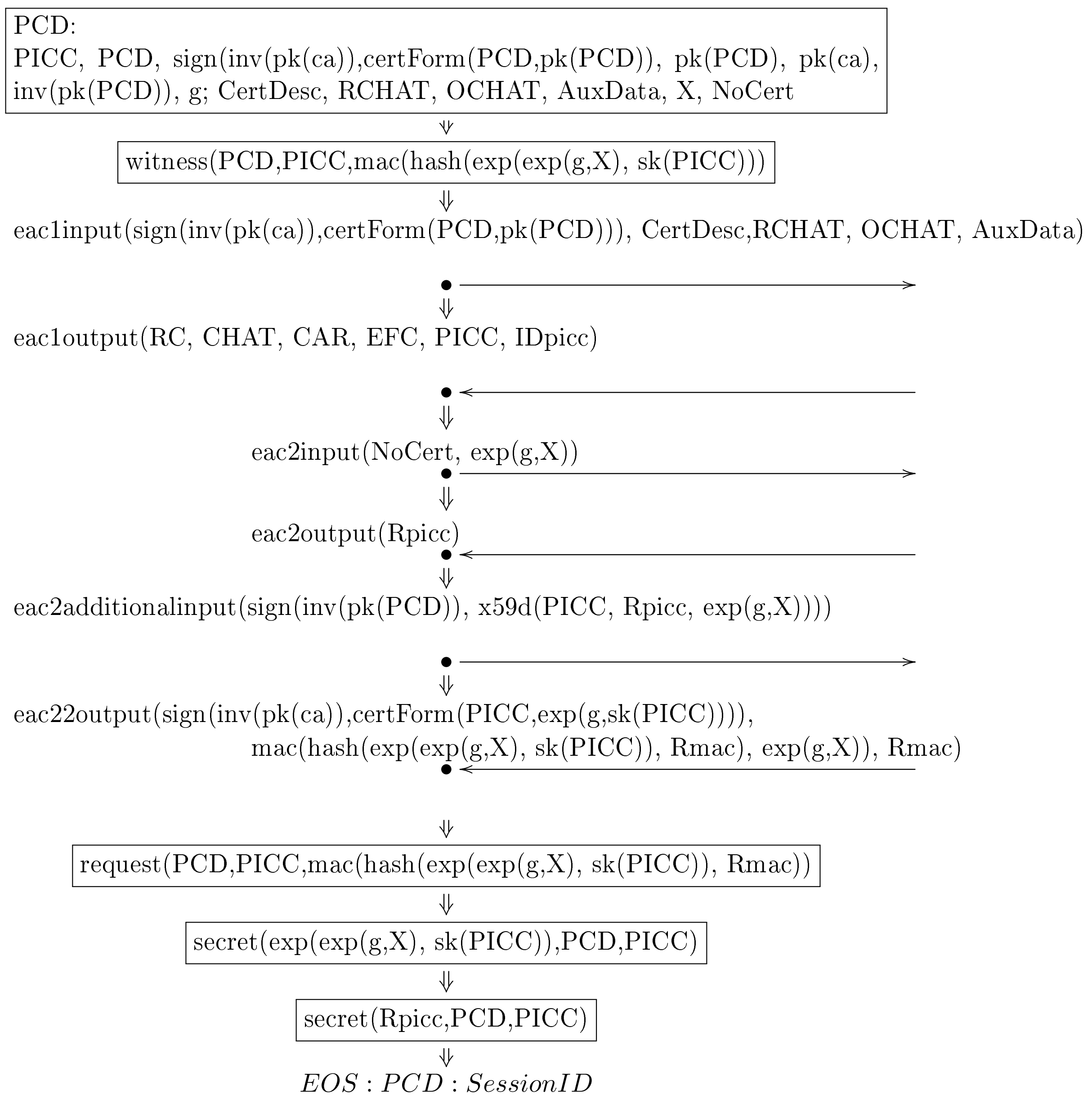
9. In case of no more actions to translate

$$tr_{Str}(A, B, \epsilon) = EOS\ A\ SessionID$$

Where SessionID is a unique identifier for the session.

7.3.4 EAC in strands

Follows a preliminary translation form *Future AnB* specifications to plain strands (non-detailed as one can see shortly), The first block gives the initial knowledge of the strand owner and we use semi-colon to separate fresh knowledge created by the owner.


PCD:
PICC, PCD, sign(inv(pk(ca)),certForm(PCD,pk(PCD)), pk(PCD), pk(ca), inv(pk(PCD)), g; CertDesc, RCHAT, OCHAT, AuxData, X, NoCert
witness(PCD,PICC,mac(hash(exp(exp(g,X), sk(PICC)))
eac1input(sign(inv(pk(ca)),certForm(PCD,pk(PCD))), CertDesc,RCHAT, OCHAT, AuxData)
eac1output(RC, CHAT, CAR, EFC, PICC, IDpicc)
eac2input(NoCert, exp(g,X))
eac2output(Rpicc)
eac2additionalinput(sign(inv(pk(PCD)), x59d(PICC, Rpicc, exp(g,X))))
eac22output(sign(inv(pk(ca)),certForm(PICC,exp(g,sk(PICC)))), mac(hash(exp(exp(g,X), sk(PICC)), Rmac), exp(g,X)), Rmac)
request(PCD,PICC,mac(hash(exp(exp(g,X), sk(PICC)), Rmac))
secret(exp(exp(g,X), sk(PICC)),PCD,PICC)
secret(Rpicc,PCD,PICC)
EOS : PCD : SessionID

7.3.5 Making strands *detailed*

This translation aims to find how participants construct or destruct messages and how to check messages. To show how different messages are dealt with we translate a transition relation tr_{ex}: *strands* $\rightarrow$ *strands* defined for message exchanging (send and receive) as follows:

- Sending a message : We first introduce $compose_K$ that maps from terms to terms, more precisely

$$compose_K : \mathcal{T}_{\Sigma(V)} \mapsto \mathcal{T}_{\Sigma(\mathcal{V})}$$

Then we define tr_{ex} in the case of sending a message:

$$tr_{ex}(Snd\ m\ S) = Snd(compose_K(m)); tr_{ex}(S)$$

Before sending a message, a function $compose_K$ is applied on that message to *explain* how this message is composed according to agent's knowledge K; the function $compose_K$ is defined as follows:

$$compose_K(t) = \begin{cases} x_i & if[x_i \mapsto t] \in K \\ f(compose_K(t_1), ..., compose_K(t_n)) & if\ t \approx\ f(t_1, ..., t_n) \\ \perp & otherwise \end{cases}$$

- Receiving a message:

$$tr_{ex}(Rcv\ m\ S) = Rcv\ \mathcal{X}_i; \varphi; tr_{ex}(S)$$

Once a message is received it is bound to $\mathcal{X}_i \notin K$. Then a procedure ANA(K) is run to update K to $\bar{K}$ and produce φ (new checks, e.g., $vsign(k, sign(inv(k), M))== \top$). ANA(K) is repeated until we reach a fixed point, i.e., until no more analysis is possible.

Updating K:

$$(\bar{K}, \varphi) = ANA(K[X_i\ \mapsto\ m])$$

We define ANA(K) according to the different cases of received messages:

1. Asymmetric encrypted message: An asymmetric encrypted message has the form of `crypt(k,m)`. ANA(K):

$$\begin{array}{l} (1)a-\ \ Find\ a\ key\ [\mathcal{X}_j, \texttt{inv(k)}] \overset{?}{\in} K \\ if\ yes\ \mathcal{X}_j \\ (1)b-\ \ add\ to\ the\ checks\ \varphi\ a\ new\ check:\ vcrypt(\mathcal{X}_j, \mathcal{X}_i) == \top \\ (1)c-\ \ \mathcal{X}_k \mapsto\ decrypt(\mathcal{X}_j, \mathcal{X}_i), s.t.\ \mathcal{X}_k \notin K. \\ (2) Check\ if\ any\ \mathcal{X} \in K\ can\ be\ generated\ in\ a\ different\ way. \\ if\ yes\ add\ it\ to\ the\ checks\ \varphi \end{array}$$

2. Symmetric encrypted message: Symmetric encrypted message has the form of `scrypt(k,m)`. ANA(K):

$$\begin{array}{l} (1)a-\ \ Find\ a\ key\ [\mathcal{X}_j, \texttt{k}] \overset{?}{\in} K \\ if\ yes\ \mathcal{X}_j \\ (1)b-\ \ add\ to\ the\ checks\ \varphi\ a\ new\ check:\ vscrypt(\mathcal{X}_j, \mathcal{X}_i) == \top \\ (1)c-\ \ \mathcal{X}_k \mapsto\ descrypt(\mathcal{X}_j, \mathcal{X}_i), s.t.\ \mathcal{X}_k \notin K. \\ (2) Check\ if\ any\ \mathcal{X} \in K\ can\ be\ generated\ in\ a\ different\ way. \\ if\ yes\ add\ it\ to\ the\ checks\ \varphi \end{array}$$

3. Signed message: A signed message has the form of `sign(inv(k),m)`. ANA(K):

$$\begin{array}{l} (1)a - \; \textit{Find a key } [\mathcal{X}_j, \mathtt{k}] \overset{?}{\in} K \\ \textit{if yes } \mathcal{X}_j \\ (1)b - \; \textit{add to the checks } \varphi \textit{ a new check}: \; vsign(\mathcal{X}_j, \mathcal{X}_i) == \top \\ (1)c - \; \mathcal{X}_k \mapsto \; open(\mathcal{X}_j, \mathcal{X}_i), s.t. \; \mathcal{X}_k \notin K. \\ (2) \textit{Check if any } \mathcal{X} \in K \textit{ can be generated in a different way.} \\ \textit{if yes add it to the checks } \varphi \end{array}$$

4. Formatted message: A message that is enclosed with a *format f*) with a form of *f*($t_1, ..t_n$). We also consider concatenating represented by a comma separated list of messages (the way it is composed) with $proj_i$ for decomposing it. We consider it as special case of formats although we do not encourage its use as mentioned earlier. ANA(K):

$$\begin{array}{l} (1)a - \; \textit{add to the checks } \varphi \textit{ a new check}: \; verify_f(\mathcal{X}_i) == \top \\ (1)b - \; \mathcal{X}_k \mapsto \; get_1(\mathcal{X}_i), \ldots \; \mathcal{X}_{k+n} \mapsto \; get_n(\mathcal{X}_i), s.t. \; \mathcal{X}_k, \ldots\mathcal{X}_{k+n} \notin K. \\ (2) \textit{Check if any } \mathcal{X} \in K \textit{ can be generated in a different way.} \\ \textit{if yes add it to the checks } \varphi \end{array}$$

5. A non-parsable message: A message that can not be decomposed any further. This case includes:
   - Atomic messages such as constants and variables
   - Hashed messages
   - Message authentication codes
   - Exponentiation
   - Multiplication

   All those messages are not supposed to be decomposed, but they may give the ability for other messages to be further decomposed. The analysis ANA(K) will be:

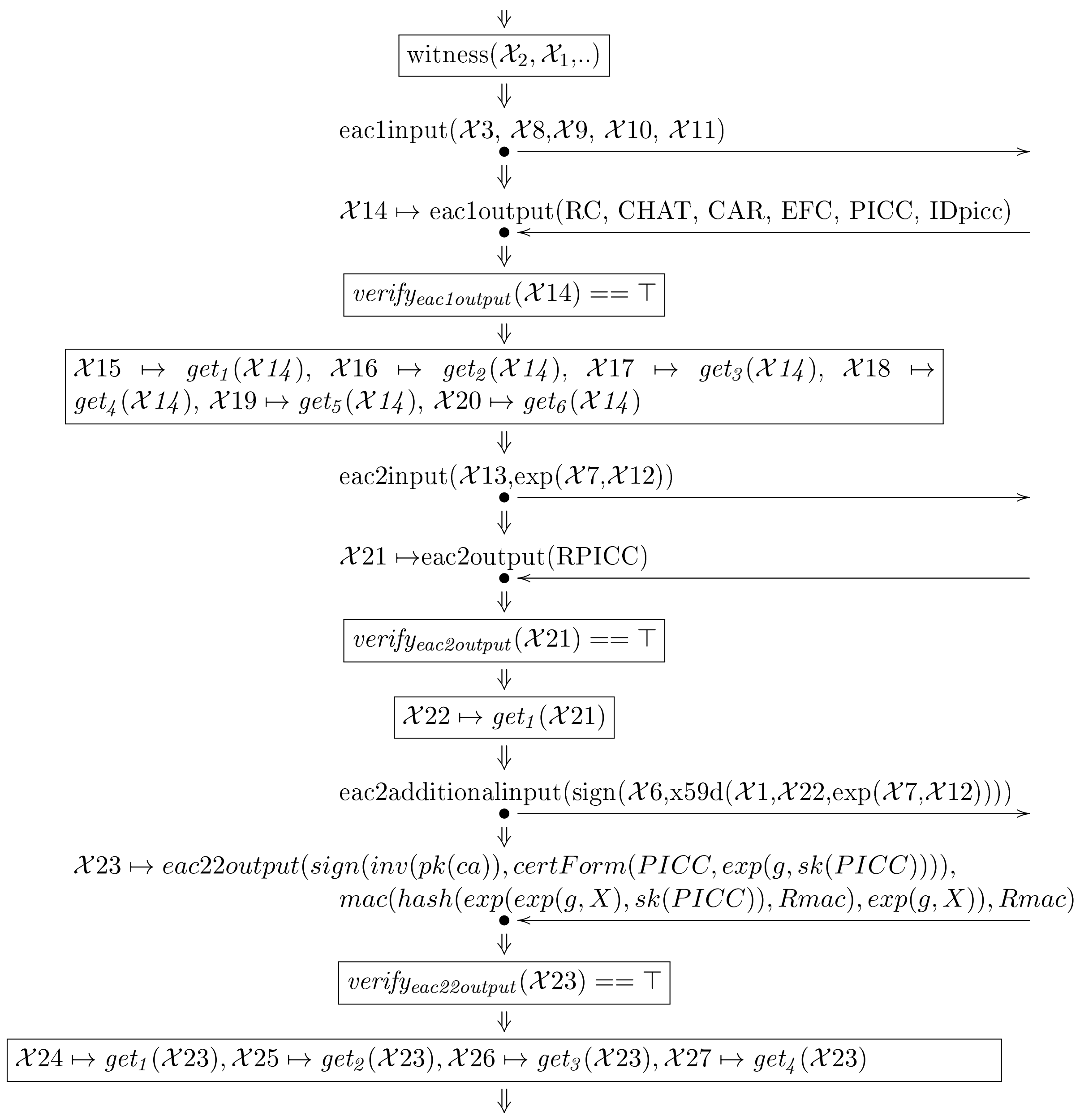
$$\begin{array}{l} \textit{Check if any } \mathcal{X} \in K \textit{ can be generated in a different way.} \\ \textit{if yes add it to the checks } \varphi \end{array}$$

We defined tr_{ex} for message exchanging for this time and leave other case for future extensions

7.3.6 EAC in *detailed strands*

Here we show the *detailed* strand for the participant PCD.

PCD:
$\mathcal{X}1 \mapsto$ PICC, $\mathcal{X}2 \mapsto$ PCD, $\mathcal{X}3 \mapsto$ sign(inv(pk(ca)), certForm(PCD, pk(PCD))),
$\mathcal{X}4 \mapsto$ pk(PCD), $\mathcal{X}5 \mapsto$ pk(ca) $\mathcal{X}6 \mapsto$ inv(pk(PCD)), $\mathcal{X}7 \mapsto$ g;
$\mathcal{X}8 \mapsto$ CertDesc, $\mathcal{X}9 \mapsto$ RCHAT, $\mathcal{X}10 \mapsto$ OCHAT, $\mathcal{X}11 \mapsto$ AuxData,
$\mathcal{X}12 \mapsto$ X, $\mathcal{X}13 \mapsto$ NoCert


witness($\mathcal{X}_2$, $\mathcal{X}_1$,..)
eac1input($\mathcal{X}$3, $\mathcal{X}$8,$\mathcal{X}$9, $\mathcal{X}$10, $\mathcal{X}$11)
$\mathcal{X}$14 $\mapsto$ eac1output(RC, CHAT, CAR, EFC, PICC, IDpicc)
$verify_{eac1output}(\mathcal{X}14) == \top$
$\mathcal{X}15 \mapsto get_1(\mathcal{X}14)$, $\mathcal{X}16 \mapsto get_2(\mathcal{X}14)$, $\mathcal{X}17 \mapsto get_3(\mathcal{X}14)$, $\mathcal{X}18 \mapsto get_4(\mathcal{X}14)$, $\mathcal{X}19 \mapsto get_5(\mathcal{X}14)$, $\mathcal{X}20 \mapsto get_6(\mathcal{X}14)$
eac2input($\mathcal{X}$13,exp($\mathcal{X}$7,$\mathcal{X}$12))
$\mathcal{X}$21 $\mapsto$eac2output(RPICC)
$verify_{eac2output}(\mathcal{X}21) == \top$
$\mathcal{X}22 \mapsto get_1(\mathcal{X}21)$
eac2additionalinput(sign($\mathcal{X}$6,x59d($\mathcal{X}$1,$\mathcal{X}$22,exp($\mathcal{X}$7,$\mathcal{X}$12))))
$\mathcal{X}23 \mapsto eac22output(sign(inv(pk(ca)), certForm(PICC, exp(g, sk(PICC)))), mac(hash(exp(exp(g, X), sk(PICC)), Rmac), exp(g, X)), Rmac)$
$verify_{eac22output}(\mathcal{X}23) == \top$
$\mathcal{X}24 \mapsto get_1(\mathcal{X}23), \mathcal{X}25 \mapsto get_2(\mathcal{X}23), \mathcal{X}26 \mapsto get_3(\mathcal{X}23), \mathcal{X}27 \mapsto get_4(\mathcal{X}23)$

$$\begin{tikzcd}[arrows=Rightarrow]
{} \arrow[d] \\
\boxed{\begin{array}{l} vsign(\mathcal{X}5, \mathcal{X}24) == \top \\ \mathcal{X}28 \mapsto open(\mathcal{X}24) \\ \mathcal{X}29 \mapsto get_1(\mathcal{X}28) \\ \mathcal{X}30 \mapsto get_2(\mathcal{X}28) \end{array}} \arrow[d] \\
\boxed{\begin{array}{l} \mathcal{X}26 == exp(\mathcal{X}7, \mathcal{X}12) \\ \mathcal{X}29 == \mathcal{X}1 \\ \mathcal{X}25 == mac(hash(exp(\mathcal{X}30, \mathcal{X}12), \mathcal{X}27) \end{array}} \arrow[d] \\
\boxed{\text{request}(\mathcal{X}_2, \mathcal{X}_1,..)} \arrow[d] \\
\boxed{\text{secret}(..,\mathcal{X}_2, \mathcal{X}_1)} \arrow[d] \\
\boxed{\text{secret}(.., \mathcal{X}_2, \mathcal{X}_1)} \arrow[d] \\
EOS
\end{tikzcd}$$

8 Generating implementation

The implementation generated from specification is divided into two categories, first one is generating programs per protocol participant (a program for PCD and another for PICC). The second is to generate a class per format in the `Formats` section, e.g., *eac1input*. We demonstrate how the generated implementation is supposed to be.

8.1 Participants programs

In Listing 9 we show roughly how a Java program for a participant (PCD) of the EAC protocol may look like. One can notice how this code is a direct mapping from detailed strands of PCD shown in the previous section. Note also that fact related to goals (secret, witness, etc.) are not present in the generated code since they are not relevant to it.

Listing 9: PCD Java code sketch

```
void progPCD(Agent x1, Agent x2, Number x3,
      PublicKey x4, PublicKey x5, PrivateKey x6,
      Number x7, ImpData x8, ImpData x9,
      ImpData x10, ImpData x11, Number x12,
      Number x13)
{
send(eac1input(x3, x8,x9,x10,x11).encode());
String x14=recv();
if(!verifyeac1output(x14)) throw formatException(...);
eac1output x14f=new eac1output(x14);
String x15=x14f.get1();
String x16=x14f.get2();
String x17=x14f.get3();
String x18=x14f.get4();
String x19=x14f.get5();
String x20=x14f.get6();
send(eac2input(x13, exp(x7,x12)).encode());
String x21=recv();
if(!verifyeac2output(x21)) throw formatException(...);
eac2output x21f=new eac2output(x21);

String x22=x12f.get1();
send(eac2additionalinput(sign(x6,
                x59d(x1,x22,exp(x7,x12)))).encode());
String x23=recv();
if(!verifyeac22output(x23)) throw formatException(...);
eac22output x23f=new eac22output(x23);
```

```
String x24 = x23f.get1();
String x25 = x23f.get2();
String x26 = x23f.get3();
String x27 = x23f.get4();
if(!vsign(x5,x24)) throw signatureException(...);;
String x28=open(x5,x24);
String x29 =  x28.get1();
String x30 =  x28.get2();
if(!(x26==exp(x7,x12))) throw checkException(...);
if(!(x29==x1)) throw checkException(...);
if(!(x25==mac(hash(exp(x30,x12),x12)))  throw checkException(...);
}
```

8.2 Formats classes

For each format in `Formats` section we generate a Java class that implements the following methods:

- A constructor for abstract syntax.
- A constructor that gets a plain text (string) in concrete syntax and parses it, e.g., complaining if required elements are missing or the message is not well-formed.
- A method `encode` to "pretty-print" the object into concrete syntax.
- A function `get`i`()` for every element X_i of the object.

We here give Java-like class sketch for *certForm* format of EAC protocol base on its declaration in `Formats` section (certForm(Agent, PublicKey);)

```
class certForm
{
    private Agent A;
    private PublicKey PK;


    public certForm(String XML)
    {
            // parse XML fields in a plain text into
        // certForm data member variables A and PK

         }
        public certForm(Agent A, PublicKey PK){
                this.A = A;
                this.PK = PK;
                // May contain filters to prevent
                // injection attacks
            }

        public String encode(){
          return "<certForm> <agent>" + A.toString()+
          "</agent> <publickey>"  + PK.toString()+
          "</publickey> </certForm>";
        }

        public Nonce get1(){return A;}
        public PublicKey get2(){return PK;}
        public bool verifycertForm(String S)
        { // verifies if S can be parsed into vertForm format
        }

    }
```

The format's class generation can be connected to its corresponding concrete structure driven from instance from XML or XSD file (correspondence by similar naming maybe).

9 The AVISPA Intermediate Format (IF)

In this section we give an overview of IF and the significance of translating protocol specification in *Future AnB* to it. IF [3] was developed within the AVISPA project as the common input language for the mentioned AVISPA tools OFMC [7], SATMC [12], and CL-AtSe [13]. It is designed to be actually independent of particular methods, but it is somewhat biased towards model-checking.

The IF specification of a protocol describes an infinite-state transition system by: an initial state, a transition relation, and a set of attack states. Every state is a finite set of facts. Examples of these facts are $state\_role(msgs)$ that describes the current local state of an honest agent in a given protocol role; $\mathsf{iknows}(m)$ as before to denote that the intruder knows m; and there are other facts for describing the goals.

The initial state contains the initial local state for each honest agent and session, as well as the initial intruder knowledge. The transition relation is given by conditional rewrite rules on states. These rules are of the form

$$L|Cond = [V] \Rightarrow R$$

This rule works as follows:

- L is a set of facts that has to be true in the current state for the rule to be applicable.
- $Cond$ is a set of conditions like equalities and inequalities of terms in L that also have to be true.
- V is a set of variables; these are used when agents create fresh (random/unpredictable) values.
- R is a set of new facts to replace L when applying the rule.

Goals correspond to attack state, and a protocol achieves a goal if and only if the attack state corresponds to that goal is not reachable. state-transition systems we can formalize. We leave the definition of the translation function from *labelled* strands to IF specification for abbreviation.

10 Conclusion

We introduced an APS language, *Future AnB*, that extends AnB with basic extensions we found essential for eID protocol specification and verification. We used EAC protocol as a running example through this document. We presented the syntax of our languages in EBNF and defined a semantics for our language by defining a translation to strands that we defined a semantics for. From the intermediate stage of *detailed* strands we demonstrated how generated Java implementation may look like and we also introduced an example of a format class in Java. We also defined a translation from our intermediate stage of *detailed* strands to IF specification. We also pointed out some possible future extensions such as sub-protocols and database manipulation. We have already a prototype implementation and we plan to extend it based on what we defined.

A *Future AnB* Grammar in EBNF

```
⟨Protocol⟩ ::= [⟨ProtocolName⟩]⟨Types⟩[⟨Mappings⟩][⟨Formats⟩]
               [⟨Macros⟩]⟨Knowledge⟩ ⟨Actions⟩⟨Goals⟩

⟨ProtocolName⟩ ::= ‘Protocol’‘:’ ⟨Ident⟩‘;’

⟨Types⟩    ::= ‘Types’‘:’
               (⟨Type⟩ ⟨Ident⟩ ( ‘,’ ⟨Ident⟩)* ‘;’)+

⟨Type⟩     ::= ‘Agent’ | ‘Nonce’ | ‘PublicKey’
               | ‘Number’ | ‘SymmetricKey’ | ‘ImpData’
               | ‘PrivateKey’| ‘Msg’
⟨Mappings⟩ ::= ‘Mappings’‘:’
               (⟨Ident⟩ ‘:’⟨Type⟩ (‘,’ ⟨Type⟩)* ‘->’ ⟨Type⟩‘;’)+

⟨Formats⟩ ::= ‘Formats’‘:’
               (⟨Ident⟩ ‘(’ ⟨Ident⟩(‘,’⟨Ident⟩)* ‘)’ = ⟨Msg⟩ ‘;’)+

⟨Macros⟩ ::= ‘Macros’‘:’
               (⟨Ident⟩ ‘(’ ⟨Ident⟩(‘,’⟨Ident⟩)* ‘)’ = ⟨Msg⟩ ‘;’)+

⟨Knowledge⟩ ::= ‘Knowledge’‘:’
               (⟨Agent⟩ ‘:’ ⟨Msg⟩;)+
               [‘where’ ⟨Agent⟩ ‘!=’ ⟨Agent⟩ (‘,’⟨Agent⟩)*
               (,⟨Agent⟩ ‘!=’ ⟨Agent⟩ (‘,’⟨Agent⟩)*)* ]
⟨Actions⟩ ::= ‘Actions’‘(’‘Main’‘)’‘:’ (S)+
               (‘Actions(’⟨Type⟩ ⟨Ident⟩
               ‘(’⟨Type⟩ ⟨Ident⟩ (‘,’⟨Type⟩ ⟨Ident⟩)* ‘):’ (S)+)*

⟨S⟩        ::= (⟨Agent⟩ ⟨Channel⟩ ⟨Agent⟩ : ⟨Msg⟩
               | ⟨Agent⟩ ‘:’ ⟨Task⟩ | ‘let’ ⟨Ident⟩ ‘=’ ⟨Msg⟩)‘;’

⟨Channel⟩ ::= ‘->’|‘*->’|‘->*’|‘*->*’

⟨Task⟩     ::= ‘if’‘(’⟨Condition⟩‘)’‘then’ ⟨S⟩ ‘else’ ⟨S⟩ ‘fi’
               | ⟨S⟩ ‘OR’ ⟨S⟩

⟨Goals⟩    ::= ‘Goals’‘:’ ⟨Goal⟩+

⟨Goal⟩     ::= ⟨Agent⟩ ‘authenticates’ ⟨Agent⟩ ‘on’ ⟨Msg⟩
               | ⟨Agent⟩ ‘weakly authenticates’ ⟨Agent⟩ ‘on’ ⟨Msg⟩
```

```
                | ⟨Msg⟩ 'secret of' ⟨Agent⟩ (','⟨Agent⟩)*

⟨Condition⟩ ::= ⟨Msg⟩'=='⟨Msg⟩ | ⟨Msg⟩'!='⟨Msg⟩

⟨Msg⟩      ::= ⟨Ident⟩ | ((⟨Sgmac⟩|⟨Sgmam⟩) '('⟨Msg⟩')'

⟨Agent⟩    ::= ⟨Ident⟩ | 'i'

⟨Ident⟩    ::= ⟨Vars⟩|⟨Consts⟩

⟨Sgmac⟩    ::= 'scrypt'|'crypt'|'sign'|'mult'|
               'exp'|'hash'|'mac'

⟨Sgmam⟩    ::= 'inv'|'pk'|'shk'

⟨Caps⟩     ::= [AZ]

⟨Smalls⟩   ::= [az]

⟨Vars⟩     ::= ⟨Caps⟩ (⟨Alphanum⟩ |'-'|'_')*

⟨Consts⟩   ::= ⟨Smalls⟩ (⟨Alphanum⟩ |'-'|'_')*

⟨Alpha⟩    ::= ⟨Caps⟩|⟨Smalls⟩

⟨Alphanum⟩ ::= ⟨Alpha⟩ [09]
```

B EAC protocol specification in *Future AnB*

In here we show complete specification of EAC.

Listing 10: EAC

```
Protocol: EAC
Types:
    Agent PCD, PICC,ca;
    Nonce IDpicc, Rpicc, Rmac;
    Number X, g;
    ImpData CertDesc, RCHAT, OCHAT, AuxData, RC, CHAT,
         CAR, EFC, NoCert;
Mappings:
    pk : Agent -> PublicKey;
    inv:    PublicKey-> PrivateKey;
    sk : Agent -> Exponent  ;

Formats:
    eac1input(Msg, ImpData, ImpData, ImpData, ImpData);
    eac1output(ImpData, ImpData, ImpData, ImpData, Agent, Nonce);
    eac2input(ImpData, Number);
    eac2output(Nonce);
    eac22output(Msg, Msg, Nonce);
    eac2additionalinput(Msg);
    certForm(Agent, PublicKey);
    x59d(Agent, Nonce, PublicKey);

Macros:
    cert(A,K,CA)=sign(inv(pk(CA)),certForm(A,K))
    kdf(SEC, PUB, NONCE)=hash(exp(PUB, SEC), NONCE)

Knowledge:
    PCD:     PICC, PCD,cert(PCD,pk(PCD),ca), pk(PCD),
        pk(ca), inv(pk(PCD)), g;
    PICC:    PCD, PICC,cert(PICC,exp(g,sk(PICC)),ca), pk(ca),
        sk(PICC), g;

        where PCD!=ca, PICC !=ca;

Actions(Main):
    [PCD]*->*[PICC]: eac1input(cert(PCD,pk(PCD),ca), CertDesc,
                RCHAT, OCHAT, AuxData)
        [PICC]*->*[PCD]: eac1output(RC, CHAT, CAR, EFC, PICC, IDpicc)

        let PK_PCD=exp(g,X)

        [PCD]*->*[PICC]: eac2input(NoCert, PK_PCD)

        [PICC]*->*[PCD]: eac2output(Rpicc)

        [PCD *->*[PICC]: eac2additionalinput(sign(inv(pk(PCD)),
                      x59d(PICC, Rpicc, PK_PCD)))

        let PK_PICC=exp(g,sk(PICC))
        let K=kdf(sk(PICC),PK_PCD, Rmac)
        let Tpicc=mac(K, PK_PCD)

        [PICC]*->*[PCD]: eac22output(cert(PICC,PK_PICC,ca), Tpicc,
          Rmac)

    Goals:
        PICC authenticates PCD on Tpicc
        PCD authenticates PICC on Tpicc
        Rpicc secret of PICC, PCD
        K secret of PICC, PCD
```