D2.4

Formal Description and Analysis of Concepts (1)

Document Identification	
Date	31.08.2017
Status	Final
Version	Version 1.0

Related WP	WP 2	**Related Deliverable(s)**	D2.1-2.3, D3.1-2, D4.1, D5.1, D6.1
Lead Authors	Sebastian Mödersheim, Rasmus Birkedal	**Dissemination Level**	PU
Lead Participants	DTU	**Contributors**	FHG, TUG
Reviewers	GS (now Ubisecure), CORREOS		

This document is issued within the frame and for the purpose of the LIGHT^{est} project. LIGHT^{est} has received funding from the European Union's Horizon 2020 research and innovation programme under G.A. No 700321.

1. Executive Summary

This deliverable D2.4 -- along with the successors D2.5 and D2.6 after project years two and three -- represents the formal basis of the LIGHTest project. At the core, this means to develop a formal language for describing the different aspects of the project, in particular policies for trust, trust translation, and delegation. These must have not only a clear syntax but also a well-defined semantics. This is important for two reasons. First we want to avoid that in some corner cases it is unclear what a particular policy actually means. In fact, the formalization can often reveal when different partners from academia and industry have a slightly different understanding of an intuitive concept. Discovering and resolving such mismatches early in the project is invaluable. Second, we want to mechanize policies, i.e., have automated verifiers to determine whether a policy is satisfied or not; obviously such a verifier cannot reason intuitively, but needs a precise algorithm to follow. With a precise semantics of the policy language it is possible to prove (or find counter-examples) that this algorithm correctly implements the policy language, or even better, automatically derive the algorithm from the policy. In fact, inspired from this latter point we have decided to go for a logic programming approach where we have only one language, the LIGHTest Trust Policy Language TPL that is based on Prolog and that allows for declarative policy specification that are directly executable as soon as all concepts are concrete. Additionally, we envision that for greatest ease of use, we can have simple, possibly visual, languages for end users that can handle most of the common specifications and can be easily mapped into TPL. In fact such a graphical tool is a mandatory part of several other work packages. In this way TPL is a basis and frame for the entire project.

2. Document Information

2.1 Contributors

Name	Partner
Sebastian Mödersheim	DTU
Rasmus Birkedal	DTU

2.2 History

Version	Date	Author	Changes
0.1	09.02.2017	Sebastian Mödersheim	Added sections: Trust Policy Language TPL Basic Scenarios in TPL Separating Policies from Procedures GUI Mockup
0.2	22.06.2017	Rasmus Birkedal	Added sections: TPL Vocabulary Further Use-Cases
0.3	14.07.2017	Rasmus Birkedal	Added section: Designing Translation Policies
0.4	14.07.2017	Sebastian Mödersheim	Updates on Translation vs. Translation Design
0.5	18.07.2017	Sebastian Mödersheim	Minor polishing
0.6	10.08.2017	Sebastian Mödersheim	First Final Draft
0.7	14.08.2017	Rasmus Birkedal	Final Draft
0.8	30.08.2017	Rasmus Birkedal	Post-Review Draft

3. Table of Contents

3.1 Table of Figures

4. Trust Policy Language TPL

4.1 Introduction

We describe here the LIGHTest Trust Policy Langauge, shortly called TPL, for trust schemes, trust translation schemes, and delegation schemes that can in a uniform way integrate the different parts of the project in a precise and formal way and also serve as a basis for reasoning (i.e., the ATV) and testing. The language is based on the existing programming language Prolog in order to draw both from the rich research on *logic programming* (including existing interpreters) and existing policy languages like Secpal and DKAL that are similarly based on *Horn clauses*. Key points in favour of this choice are:

- It is trivial to formalize all simple policies that are based on a kind of enumeration.
- It offers us an easy mechanism to describe relations between concepts, e.g. what criteria need to be satisfied to fulfill a certain standard, logical combinations of policies (and/or/not).
- It is ideal for concepts like delegation and black-listing (for this reason for instance the access control policy languages like SECPAL (Becker, Fournet, & Gordon, 2010) and DKAL (Blass, Caso, & Gurevich, 2012) by Microsoft are based similarly on Horn clauses).
- The language comes directly with a clear formal meaning including an evaluation procedure, i.e., our specifications are directly "executable".
- We can be sure that the language is powerful enough because it is Turing complete (every computable policy can be expressed) (Hopcroft, Motwani, & Ullman, 2003).
- The evaluation can be made to directly trigger necessary queries to servers, e.g., using DNSSec, and process their answer; thus the bulk of the ATV can directly be encoded into the language, either as a prototype/testing reference or even as the final product.
- For the average users we can either provide design patterns for their policy or even interface to a simpler (possibly graphical) language that they can use more intuitively but that is limited in expressive power. In this way one may be able to use LIGHTest without any learning curve in 99% of all cases, but when one wants to express something really non-standard (the remaining 1% of cases), the language still allows that.

We illustrate the flavour of the language and what specifications could look like with a few examples. The language is based on Horn clauses that have the form

$$conclusion :- requirements$$

This loosely corresponds to a sentence of the form: “if the requirements on the right are all satisfied, then the conclusion also holds as a result”. Note that the requirements here are sufficient but not necessary: if the requirements are not met, there could still be another Horn clause that allows the same conclusion.

Consider as a specific example the sentence “I trust X if I trust someone that delegates to X”. This could be expressed as a Horn clause in the following way.

$$trust(X) :- delegate(Y, X), trust(Y).$$

In the above, $:-$ should be read as “if” in the sense of a sufficient (but not necessary) condition, i.e., if the requirements on the right-hand side are not met, there may still be another clause to derive that I trust X. The comma between the requirements should be read as "and". The clause should thus be read as “I trust X if I trust Y, and Y delegates to X”. Here the terms $trust$ and $delegate$ are not built-in parts of the (base) language, and the clause says nothing of their meaning in isolation, i.e. nothing is said of what it means to trust or delegate to something. However, we consider having a library of the most important terms and concepts (so users do not have to start from scratch when specifying their own policy) and they may have also a distinguished meaning for our ATV.

The language would then consist of a set of such clauses, each having exactly one term (the *head* of the clause) before the $:-$ (the “if”), and zero or more terms separated by commas (the *body* of the clause) after the $:-$. Expressing trust by a bounded number of delegations in this language could be done using the following two clauses.

$$trust(X, N) :- trust(X).$$

$$trust(X, N) :- N > 0, delegate(Y, X), trust(Y, N - 1).$$

The first clause should express that I trust X through at most N steps of delegation if I trust X directly (without considering delegation). The second clause says that I trust X through at most N steps of delegation if N is greater than zero, Y delegates to X, and I trust Y through N-1 steps of delegation.

One big advantage of using this language is that it happens to be valid Prolog code. It is then possible to use a Prolog environment to evaluate the policy against a prototype, which, conveniently, can also be specified as Prolog code. The following is a prototype meant to express that I trust *a*, *b*, and *c* (this could represent that I trust these because they are listed in some trust list), and that *c* delegates to *d*, which delegates to *e*.

$$trust(a).$$

$$trust(b).$$

$$trust(c).$$

$$delegate(c, d).$$

$$delegate(d, e).$$

With the two clauses with $trust(X, N)$ in the head loaded into a Prolog environment together with the above prototype, the query $trust(e, 2)$. will return true, and $trust(e, 1)$. will return false.

4.2 A Detailed Definition of TPL

At the core of such Horn-clause based languages are so-called *facts*, which are terms that can be either true or false, e.g. $trust(a)$ above. In contrast to Prolog, which is rather liberal on the syntax of facts, we recommend for TPL typed first-order terms. Here we would distinguish different kind of symbols:

- Fact symbols (like $trust, >$): they represent something that can either be true or false. Note we may have facts that have no argument, like "*lifeisgood".*
- Function symbols (like '-'): they are different from facts as they do not yield simply true or false, but rather a value. We may control with a type system that they are not applied to incompatible types.
- Constant symbols (like 0,1,*a,b,c* above) are a special case of function symbols with no arguments.
- Variable symbols (like A,B,C,X,Y). Note that all identifiers that start with an upper-case letter are variable symbols in Prolog.

Note that except a few built-in symbols, these symbols have no predefined meaning, but rather the meaning comes only from the Horn clauses you specify.

We can build terms using function, constant, and variable symbols as expected, e.g. *signature(privkey(alice),X).* A fact is then a fact symbol applied to a number of terms, e.g. *knows(s,signature(privkey(alice),X)).* It is common to follow the convention *predicate(subject,object)* when using facts, so that one can read it intuitively as "*subject predicate objects*", i.e., in this example "[the server] s knows a signature of alice of message X".

A *Horn clause* has the form

fact :- $fact_1$, ..., $fact_n$.

(One may simply write "*fact.*" in case n=0.)

It is to be read as "*fact* holds if all the facts $fact_1$, ..., $fact_n$ hold." where "if" is to be read as an implication from right to left, i.e., *fact* <= $fact_1$, ..., $fact_n$. For example

$trust(X) :- loa(X, L), L > 1.$

$loa(X, 3) :- quality(X, good).$

$loa(a, 2).$

$loa(b, 1).$

$quality(c, good).$

This is a simple example policy where we trust anybody with level of assurance higher than 1, and a simple trust translation where we say that everybody who satisfies "quality" at level "good"

has level of assurance 3. The next three facts encode a concrete example, of three people *a,b,c* with levels of assurance 2, 1, and quality good. We can now query for instance *trust(a).* The standard evaluation of Prolog is to find the first Horn clause with a conclusion that matches the query. To illustrate that and the hierarchy of calls we make this as a list with sublists where each bullet represents a query:

1. *trust(a)*. The first (and only) matching clause is "*trust(X) :- ...*", meaning that we try to fulfill this clause using *X=a.* Thus, we try to prove the new queries *loa(a,L)* and *L>1*.
    - *loa(a,L)* Now the first matching clause is *loa(X,3) :- quality(X,good).* Thus we now try:
        - *quality(a,good)* This fails, we have no matching clause.

      Since this attempt of proving *loa(a,L)* failed, we backtrack – meaning that we go back to the last state in this procedure where multiple viable choices were available to us – and see if any other clause would work. The only one that is left is *loa(a,2).* As this one has no further conditions, we are done with this one under the match L=2.

    - We have to still satisfy *L>1*, which given L=2 is also satisfied.

    Thus, *trust(a)* has succeeded.

With a similar derivation we see that the query *trust(b)* will fail. Note that a query may have variables, i.e., we can query *trust(Z)* to mean that we want to know any entity *Z* that we trust. Let us see the evaluation of that:

2. *trust(Z)*. We match again the clause "trust(X) :- ..." and thus unify *X=Z*, giving us the new query:
    - *loa(Z,L)* To satisfy this, we can use the clause *loa(X,3) :- quality(X,good)*, thus unifying *X=Z.* Thus we try to prove
        - *quality(Z,good)* This time, we can actually satisfy this, because there is the clause *quality(c,good).* Thus unifying Z=c, we have succeed, and have *loa(c,3)*.
    - *L>1* is now also true since *L=3>1.*

    Thus, trust(c) is true.

Note that this is not the only solution to this query. Prolog will always return the first solution it finds and then ask if you want Prolog to search for further solutions. In this case, it will next find the solution *trust(a)* and, if you then again choose to search for further solutions, it will fail. The order in which solutions are found depends on the order in which you give the Horn clauses (Prolog will try them in the order you specified them) and the same holds for the order of the requirements on the right-hand side of a Horn clause.

4.3 Some Further Examples

Let us illustrate by some further examples how specifications in this language look like:

The above examples have defined e.g. that "*quality" "good"* translates into level of assurance 3. We would like to formulate however, that everybody who satisfies assurance level 3, also satisfies levels 1 and 2. We can do that as follows:

$loa(X, N) :- N < 4, loa(X, N + 1).$

Thus, to prove that somebody has assurance level N, it is sufficient to prove that they have assurance level N+1. Note that we have to bound N+1 here to become at most 4 (or whatever is the highest level of assurance) because otherwise Prolog may run into an infinite loop at this point.

Another example is blacklisting. Here we need a form of negation, and that is often tricky. The philosophy behind Horn clauses is to express things positively, i.e., ways to derive true facts from other true facts. However, Prolog allows for a certain form of negation, for instance we can modify the trust rule from our previous example as follows:

$trust(X) :- loa(X, L), L > 1, not(blacklisted(X)).$

We can now simply represent blacklisted entities by enumerating them, e.g.

$blacklisted(c).$

The evaluation of the example query *trust(Z)* would now run just as before, and first find that *loa(Z,L).Z>1* could be derived with *Z=c, L=3* and it remains to evaluate the query *not(blacklisted(c)).* To that end, Prolog will try to prove the positive query *blacklisted(c).* This succeeds, since *c* is indeed blacklisted, so the negative query *not(blacklisted(c))* fails. At this point Prolog backtracks and finds other solutions to satisfy *loa(Z,L)* and comes up with *Z=a and L=2*. Now the query *blacklisted(a)* fails, and thus *not(blacklisted(a))* succeeds, so *Z=a* is still a solution.

Note that this is called *negation by failure*, i.e., *not(fact)* is true if we fail to derive *fact.* This is contrary to classical logic, because in classical logic, if from A follows B then also from A and A' follows B. In other words, learning additional information (like A') never limits what we can derive (like B). Negation by failure violates this property: in the example, if we add to our Horn clauses the fact "*blacklisted(a).*" then the derivation *trust(a)* is no longer possible.

For blacklists that is exactly the behavior we want, but in general, one has to be careful in the use of negation, because it may easily violate one's intuition and produce hard to understand and hard to debug specifications. Actually the only use should be for blacklist-type queries: when the answer is by default positive, and only negative when explicitly stated so by some rule. Note that this also prevents any "conflicting" definitions, i.e., where one policy says "yes" and the other says "no" -- such a combination simply cannot be made.

4.4 Example: ISO/IEC FDIS 29115

Let us now give an example from the enrollment part of the ISO standard (that was earlier used as an example for ER diagrams). Note that many aspects are outside of an automated evaluation, e.g., checking the (physical) passport of a person who showed in person for enrollment. We assume however that all such facts can be stated about a person in a list of Horn clauses and then handed into the reasoning machinery for evaluation. We are describing now this evaluation as clauses about these facts.

There are in fact many ways to do this, but we first would like to assume that all the parameters and inputs of an enrollment session are represented in some way by a term (in the end, a structured document, not unlike an XML tree). We do not have to really worry about this structure now, we just formulate rules that process one such given term X, for instance:

```
loa1(X) :- uniq(X).
loa2(X) :- loa1(X), l2req(X).
loa3(X) :- loa2(X), l3req(X).
loa4(X) :- loa3(X), l4req(X).
```

This expresses, that an enrollment application satisfies level of assurance 1 if some predicate uniq is satisfied. We did not specify the predicate "uniq" but it just represents the requirement that the identity of the person to be enrolled is unique. In this way we should simply note all those requirements we cannot formalize or do not want to formalize in this process -- they simply remain abstract requirements. For all other levels of assurance, they are defined as achieving the next lower level and some additional requirements. For these we have facts l2req,...,l4req that we do specify in more detail by Horn clauses:

```
l2req(X) :-
                person(X),
                inPerson(X),
                idDocument(X,D).
```

Here, person, inPerson and idDocument are again facts we do not specify in more detail, but they just abstractly refer to the condition that the entity to be enrolled is a person, showed up in person, and has a document D to prove its identity. We may have as well just written here "idDocument(X)" without the variable D, because it could be part of the entire application term that we refer to with "X". However, in other clauses below we want to refer to the document that is used to prove the identity. For that end, we make it a parameter of this fact, so we can easily refer to it when we need to. There are other clauses to satisfy level 2 requirements, namely, in the cases of not showing up in person or not being a person in the first place:

```
l2req(X) :-
                person(X),
                notInPerson(X),
                idDocumentPossession(X).
l2req(X) :-
                nonPerson(X),
                autoritativeInformationRecorded(X).
```

At the level 3 requirements, we actually now refer to the ID-document that the applicant has shown because the requirement is to check this document with the records of the original issuer who once produced this document. Again this is outside the automated systems, but we can still describe it to some extent, namely that D is the document used to prove the identity and that exactly that document should be checked with the source:

```
l3req(X) :-
        person(X),
        inPersonProofed(X),
        contactInformationVerified(X),
        idDocument(X,D),
        checkedWithSource(D),
        personalInformationCorroborated(X),
        verifiedCredentialClaim(X).
```

A similar modeling method we use here is to extract aspects of the application, e.g., when the applicant does not show up in person, then they must have shown the possession of a level-3 certificate:

```
l3req(X) :-
        person(X),
        notInPersonProofed(X),
        hasCredential(X,C), loa(C,L), L >= 3,
        verifiedCredentialClaim(X).
```

Here, to extract that certificate from the application we have introduced another fact "hasCredential(X,C)" that we also do not specify concretely, and from which we extract the level of assurance with another fact "loa(C,L)". This allows us to formulate the rules without specifying the precise structure of the terms X (the entire application) and C (the concrete credential). The advantage of this modeling is that it does not depend on a particular format of applications or credentials. In fact, this specification is compatible with any credential technology if only you can specify a predicate loa(C,L) on credentials that extracts the level of assurance embedded in such a credential and produces failure when the credential in question does not have this concept of assurance levels. Quite similarly we can now formulate that non-person entities need to apply with a level 3 credential that was issued by a human:

```
l3req(X) :-
        nonPersonEntity(X),
        trustedHardwareUsage(X),
        hasCredential(X,C), loa(C,L), L >= 3, issuer(X,I). person(I).
```

4.5 Formal Semantics

While the previous sections have relied on the readers intuition for a definition of the language, we here want to briefly sketch two ways to formally define the meaning of the language. This is only a sketch since the semantics of Prolog has been described in detail (Programming languages, 1995).

The first way is that of an interpreter. Let us call a query any conjunction of facts (like the right-hand side of a clause). Given a set of Horn clauses and a query, we first rename the variables of the Horn clauses so that they are disjoint from those of the query.

- We pick the first predicate `p` from the query and check the first Horn clause of the form `q :- p1,...,pn` such that `p` and `q` have a *unifier,* i.e., a substitution of the variables that makes the terms equal. For the free algebra (i.e., not taking into account any algebraic properties of function symbols) there is always a *most general unifier*, i.e., so that every other unifier is an instance of the most general unifier.
- Let σ be the most general unifier of p and q, then we replace `p` in the query by `p1,...,pn` and apply σ to the entire query.
- If n=0 and there are no further predicates in the query, we already have a solution.
- Otherwise, we try to recursively solve this query with the Horn clauses.
- If we reach a point where no conclusion of a Horn clause unifies with the first query, then there is no solution for that query.

This describes the algorithm for finding the *first* solution for a query, but it can be easily extended to one that lists *all* solutions, namely each evaluation gives as a result not one, but a list of solutions. Note also that this procedure does not necessarily terminate. For instance consider

$$p(X) :- p\big(f(X)\big).$$

$$p(0).$$

The query $p(Y)$ has now a simple solution, namely $Y = 0$, but since the interpreter will first try the first rule, it will rather try to derive $p(f(Y)), p(f\big(f(Y)\big)), \ldots$ and never actually reach this simple solution. However note that any language that is powerful enough to express every possible algorithm (i.e., that is Turing complete) necessarily has means for formulating such infinite evaluations. We can for most cases restrict ourselves to the *Datalog* fragment of Prolog that does not have any function symbols; that is indeed always terminating, but has therefore also less expressive power.

A more logical view of the semantics can be obtained if we consider the Horn clauses as logical formulas of first-order logic, where :- is $\Leftarrow$ (logical implication from right to left), the comma is logical conjunction and all variables of every Horn clause are universally quantified, e.g., p(X,Y) :- q(X) becomes

$$\forall X, Y.\, q(X) \implies p(X, Y).$$

Then, given a set of Horn clauses *H* and a query $p1, \ldots pn$, the solutions are those substitutions σ of the variables in $p1, \ldots, pn$ such that

$$H \vdash \sigma(p1, \ldots pn)$$

where $\vdash$ means provability (which is equivalent to logical implication in first-order logic). The nice advantage of this logical formulation is that it is independent of the procedural aspects like the order of the clauses or termination issues.

There are two extensions of this basic Prolog/TPL language that need to be considered.

- First, we may also use built-in datatypes and functions, like the binary operator + and the binary predicate >= on integers or floats. Prolog interpreters typically require here restrictions so that at any stage the operators can be applied to concrete terms; from the logical point of view it is quite difficult to integrate this, since already natural number arithmetic is beyond first-order logic. Hinrichs and Gennesereth suggest here slightly different basis called *Herbrand logic* (Hinrichs & Genesereth, 2006).
- The second complication is the use of *not* in queries and Horn clauses -- that are than actually no longer Horn clauses. This is somewhat at odds with classical logic. In a nutshell, when the interpreter encounters a query that starts with $not(p)$ then it should first (recursively) try to prove the query p. If that fails (no solution under which *p* is true), then the query *not(p)* counts as satisfied and the evaluation continues. This is ideal for blacklisting (see example above), but should be used only for this special purpose, because it destroys the monotonicity property of classical logic, namely that adding knowledge (Horn clauses) can only increase (but never decrease) the set of derivable consequences -- this is no longer true when not is involved. Model-theoretic or proof-theoretic formalizations of this phenomenon are pretty ugly.

5. Basic Scenarios in TPL

The Trust Policy Language that we propose is capable of expressing both the policy itself and the operational aspects (like queries to a DNS server). To illustrate that we give here a formalization of three basic scenarios.

5.1 Boolean Trust Scheme without Translation

This is the information flow and scenario first described in Section 12.2.1 of the architecture deliverable D2.14. We formalize it as a predicate of a text `Text` being supposedly signed by a `Signer`, these two parameters would normally be input to a query; as comments we have the step numbers from the information flow diagram above:

```
trustPublicationScenario(Text,Signer) :-
  % Steps 1.-5. checking document and signatures (local):
  document(signature(Text,PkSig)),
  certificate(SigCert),
    issuer(SigCert,Issuer),
    bearer(SigCert,Signer),
    pubkey(SigCert,PkSig),
    issuerkey(SigCert,PkIss),
    altIssuerName(SigCert,TrustMemClaim),
  % Step 6.-8. Query DNS
  queryBTS(TrustMemClaim,Issuer,PkIss),
  % Step 9. Check Trust policy
  scheme(TrustMemClaim,TrustScheme),
  trust(TrustScheme).
```

Here, the predicates `document` and `certificate` refer to facts that should be part of the current "knowledge base", i.e., that are fed into the automated trust verifier initially. The predicates `issuer` and the like are for extracting aspects of the certificate (e.g., who is the issuer of the certificate). This is done with such predicates in order to formulate it in a general way that is independent of a concrete credential format. To support a new credential format, one thus has to just define these predicates for it. Thus these part of the clause represent that we check the existing credentials and extract the Trust Membership Claim from it, which is the URL we then use in the DNS query -- that is then expressed with the predicate `queryBTS(TrustMemClaim,Issuer,PkIss).` Note that conceptually, TPL (or Prolog) do not have a notion of input or ouput in these predicates. This is because either can be left a variable in a query and we then evaluate to find the possible solutions for these variables to make the query true. The query predicate would support in fact three different options for this:

1. Only the URL (`TrustMemClaim`) is the input, and the result of the query is the entire trust list with their certificates.
2. The URL and the Issuer name, and output is the public key if the issuer is in the trust list.
3. All three -- URL, Issuer name and public key -- are input, and the query only can succeed (if the issuer is part of the trust list and has the given public key) or fail.

Note that this abstracts from the precise mechanisms with RR-records and verifying the certificate chain to the top-level domain. Finally, we have to check that we even trust the given trust scheme. Usually the trust scheme name may not be identical with the URL of the `TrustMemClaim`, therefore we use an additional predicate `scheme` to extract the trust scheme name. Then we finally verify whether the TrustScheme actually satisfies our trust policy which can be in the example scenario simply an enumeration of trusted schemes e.g.:

```
trust(["TrustScheme","signature","trust","ec","europa","eu"]).
```

5.2 Ordinal Trust Scheme without Translation

This is the second information flow and scenario described in Section 12.2.1. Here in addition, we have to check an attribute, in the example that the method of identity proofing of the said certificate was "in person". To that end, we only modify the query predicate to return a set of Attributes as a result:

```
ordinalTrustPolicyScenario(Text,Signer) :-
  % Steps 1.-5. checking document and signatures (local):
  document(signature(Text,PkSig)),
  certificate(SigCert),
    issuer(SigCert,Issuer),
    bearer(SigCert,Signer),
    pubkey(SigCert,PkSig),
    issuerkey(SigCert,PkIss),
    altIssuerName(SigCert,TrustMemClaim),

  % Step 6.-8. Query DNS
  queryOTS(TrustMemClaim,Issuer,PkIss,Attrib),
  attribute(Attrib,identityProofing,inPerson),

  % Step 9.-10. Check Trust policy
  scheme(TrustMemClaim,TrustScheme),
  trust(TrustScheme).
```

Here the predicate `attribute` is used to extract attributes from the (potentially complex) set of attributes that the query returns. This is again done to make this specification independent of the concrete representation of the attributes (like a list of pairs of attribute name and attribute value), basically only requiring that there is an attribute called "identityProofing" and that its value is "inPerson". Similar things we can do with all kinds of attributes, including level of assurance. One may argue that this is actually mixing of a procedure (checking a document and obtaining certificates) with the actual trust policy (who is trusted, which attributes are required). Our trust policy language of course allows us in a nice way to separate these concerns, as we describe in more detail in the next section.

5.3 Trust Translation Scheme Scenario

As a third example we look at the Boolean Trust Translation Scheme Scenario from Section 12.3 of D2.14. This is like the first scenario, but where the trust scheme is actually foreign (e.g. a Swiss trust scheme) and needs to be translation (e.g. into European qualified signature). The

beginning is similar again to the Boolean trust scheme, and all the difference lies in the predicate `trustWithTranslation` that allows to accept the trust in this scheme if it is either directly trusted or can be translated into a trusted one:

```
trustTranslationScenario(Text,Signer) :-
  % Steps 1.-5. checking document and signatures (local):
  document(signature(Text,PkSig)),
  certificate(SigCert),
    issuer(SigCert,Issuer),
    bearer(SigCert,Signer),
    pubkey(SigCert,PkSig),
    issuerkey(SigCert,PkIss),
    altIssuerName(SigCert,TrustMemClaim),

  % Step 6.-8. Query DNS
  queryBTS(TrustMemClaim,Issuer,PkIss),

  % Step 9.-15. Check Trust policy
  scheme(TrustMemClaim,TrustScheme),
  trustWithTranslation(TrustScheme).
```

Here, the `trustWithTranslation` is defined by two clauses, namely if it is directly a trusted scheme, or otherwise we check if it can be translated into a scheme that we already trust in:

```
trustWithTranslation(TrustScheme) :- trust(TrustScheme).
trustWithTranslation(ForeignTrustScheme) :-
  trust(TrustScheme),
  translation(ForeignTrustScheme,TrustScheme).
```

Here `translation` is a predicate that invokes again a query similar to a query for native schemes as we had them before:

```
translation(ForeignTrustScheme,TrustScheme) :-
  encoding(ForeignTrustScheme,TrustScheme,URL),
  queryBTSTranslationScheme(URL).
```

Here, we first need to generate a URL out of the foreign trust scheme and the one we want to translate to; this is done by the predicate `encoding` for our example, the foreign trust scheme is `["admin","ch"]`, the (native) trust scheme is `["signature","trust","eu"]` and encoding would yield the URL `["dingsbums__admin__ch","Translation","signature","trust","eu"]`.

6. Separating Policies from Procedures

A drawback in the examples of the previous section consists in the mixture of procedural aspects (like querying servers, checking documents) with the actual policy. It is beneficial to separate them, so that the procedure of checking a transaction will always be the same and simply invoke a check of the policy. In case of trust translation, this policy may itself trigger a procedure. Thus the policy (and the translation schemes) can easily be changed without changing the procedure that works on them, and vice-versa.

Here is now the more abstract procedure for checking a document:

```
checkDocument(Text,Signer,TrustScheme,Attributes) :-
  document(signature(Text,PkSig)),
  certificate(SigCert),
    issuer(SigCert,Issuer),
    bearer(SigCert,Signer),
    pubkey(SigCert,PkSig),
    issuerkey(SigCert,PkIss),
    altIssuerName(SigCert,TrustMemClaim),
  query(TrustMemClaim,Issuer,PkIss,Attributes),
  scheme(TrustMemClaim,TrustScheme).
```

In contrast to the formalization of the previous section, we use a unified `query` predicate. Here `query(TrustMemClaim,Issuer,PkIss,Attributes)` represents the DNS query at URL `TrustMemClaim` checking that `Issuer` belongs to the trust list, indeed has public key `PkIss` and attributes `Attributes`. The check procedure simply returns the obtained `TrustScheme` and `Attributes` and we can thus formulate the policy with respect to these two. For instance the simplemost case was the Boolean trust scheme where the policy would simply be:

```
qualifiedSig(["TrustScheme","signature","trust"," europa","eu"]).
trustpolicy1(TP,Attrib) :- qualifiedSignature(TP).

scenario1(Text,Signer) :-
  checkDocument(Text,Signer,TP,A),
  trustpolicy1(TP,A).
```

The predicate `scenario1` then binds procedure and policy together (we omit the similar predicate for the next examples). The second trust policy is additionally requiring in-person proofing:

```
trustpolicy2(TP,Attrib)  :-
  qualifiedSig(TP),
  attribute(Attrib,identityProofing,inPerson).
```

Finally the trust translation example allows for a policy that has translation in there that (when necessary) triggers a procedure for obtaining the translation:

```
trustpolicy3(TP,Attrib)
```

```
  :- qualifiedSig(TP).
trustpolicy3(ForeignTP,Attrib)
  :- qualifiedSignature(TP),translation(ForeignTP,TP).
```

Here, `translation` triggers the a query to the translation URL that is built from encoding the foreign trust policy into a subdomain of the local one:

```
translation(FTP,LTP) :-
  encode(FTP,LTP,URL),
  query(URL,_,_,Attributes),
  attribute(Attributes,recipe,"equivalent").
```

7. Formats

The previous section has not talked explicitly about the concrete data formats that the certificates or resource records from the DNS servers have. This is of course a key point in getting from a policy to a running implementation. The point is that the policy and the TPL language work on the level of *abstract syntax,* where we do not think about the concrete formatting or representation of the information, but only describe on a high-level what information must be present and what its logical meaning is.

To obtain an actual implementation, however, we must make the link to the *concrete syntax,* i.e., the concrete bitstrings that are being transmitted. This section outlines an idea for extending TPL with a new module that establishes this link between the abstract and concrete syntax. We emphasize that we *only* introduce the idea for the module – we do not give specific descriptions of how it should look. That is left for a future iteration of this deliverable.

We thus have a *parser* that reads the bitstring in concrete syntax and extracts the abstract syntax, and a *pretty printer* that produces from the abstract syntax the concrete syntax again.

To make this connection, we use an idea from the field of specifying security protocols. In this way we combine the advantages of concrete and abstract syntax: from concrete syntax that we are completely precise on the used message formats and from the abstract syntax we avoiding of cluttering up the entire protocol description with complicated but largely irrelevant details.

As a real-world example, let us consider the first message of the TLS Handshake protocol. In a high-level description we would like to simply write a term like this:

```
TLSclientHello(T, R, S, Cipher, Comp)
```

The actual message on the string level would be:

'20' '3' '3' [['1' '3' '3' T R $[S]_1$ $[Cipher]_2$ $[Comp]_1$ $]_3$ $]_2$

where 'n' means a byte of value n and $[m]_k$ means that m is a message of a variable length, and this length is given as a k-byte field before m. (In contrast, T and R have a fixed size.)

The idea is now that abstract function symbols like `TLSclientHello` are a sound abstraction of their concrete byte-level format if all formats we use are only fulfilling some reasonable properties (Mödersheim & Katsoris, 2014). With *sound abstraction* we mean:

- If the intruder can attack a system on the low byte level, then there is a similar attack on the abstract function level, i.e. if the abstract system is secure, then also the concrete system is. This means it is safe for us to just think in terms of the abstract level, in particular in modeling and verification/security proofs.

The *reasonable properties* that this soundness result requires are:

- Each format is *unambiguous*: if $f(t_1,\ldots,t_n) = f(s_1,\ldots,s_n)$ then $t_1 = s_1,\ldots t_n = s_n$, i.e. there is no byte string that can be read in more than one way as format f.
- The formats are *disjoint*: if $f_1(t_1,\ldots,t_n) = f_2(s_1,\ldots,s_m)$ then $f_1 = f_2$ (and thus $n = m$ and $t_1 = s_1,\ldots,t_n = s_n$ by unambiguity), i.e., no byte string can be parsed as more than one format.

Of course, implementations need two modules that deal with concrete syntax:

- Parser: reading a concrete string in concrete syntax and obtain a parse tree -- in abstract syntax.
- Pretty Printer/Code Generator: given a piece of code in abstract syntax, generate the string that corresponds to the concrete syntax.

Thus, as soon as we have these two modules for each of the message formats, the rest of the implementation does not need to "think" in terms of the low-level string representation anymore.

Let us consider one more example: many protocols exchange information using some XML-based formats. XML is itself a construction that allows arbitrary hierarchical structures and ensures unambiguity and disjointness, and will return the parsing result in abstract syntax, i.e., in that tree we have no longer any concrete syntax characters like angle brackets and slashes of an expression like `<dingsbums>...</dingsbums>`, but we rather see only a tree node of entity `dingsbums` and what its children are. Still, it is often nice to put yet another abstraction layer on top of such an XML format, so one does not have to browse such a XML parse tree but has a more immediate representation of the data that is suitable for ones purposes.

This also shows one thing: when using the XML-libraries for parsing/pretty printing of XML-formats, one can avoid many of the usual implementation problems "by construction", such as buffer overflows: assuming your XML-library is well-written, you do not need to write a parser yourself by hand. More generally, one of the ideas behind formats is that one should have a library of parsers/pretty printers, one for each format, and then use them, similar to a library of crypto functions. The point is in both cases that not every programmer needs to repeat all the common mistakes, but just use the "best" solution for a subtle programming problem.

There is some preliminary work on automatically generating said libraries of formats from description of formats supporting:

- XML-based formats.
- The ASN-style format description seen in the TLS example above.

Both of these are prototype implementations that supports formats with a fixed number of elements (Mejborn, 2016). Related available research work and implementations:

- The paper on formats (Mödersheim & Katsoris, 2014).
- The language SPS (formerly APS) developed by Almousa et al. during the FutureID project. A language based on Alice-and-Bob notation that can both generate formal

models to be used in protocol verification tools and generate implementations in JavaScript (a requirement from FutureID) using the formats (Almousa, Mödersheim, & Viganò, Alice and Bob: Reconciling Formal Models and Implementation, 2015).

- A first version of the language was described in the FutureID deliverable D42.3 (FutureID-Project, D42.3 - APS Definition Functions, 2014), and several specifications of protocols as found in D42.8 (FutureID-Project, D42.8 APS Files for Selected Authentication Protocols, 2015).
- The noted BSc thesis (Mejborn, 2016).
- Formats are also used as a basis for compositionality and the automated compositionality checker APCC, see (Almousa, Mödersheim, Modesti, & Viganò, 2015).

One frequently used TPL-predicate that serves exactly to abstract away from dealing with specific formats is the extract predicate. See the following section for a more elaborate example that uses it. The predicate takes three arguments. The first is some collection of data, like a certificate, the second is the name of some data item in that collection, and the third is the value of that item.

For example, `extract(Certificate, name, Name)` asserts that the `name`-property in the `Certificate` is `Name`. Assuming an XML-style format, if `Certificate` is equal to **<cert><name>Jane</name>…</cert>**, then `Name` should be *Jane*, because *Jane* is the value corresponding to the property `name` in the `Certificate`.

8. A Health Care Scenario

Description

A doctor is about to treat a patient, and wants to verify that the patient is properly insured. The verification procedure should be largely automatic. It may rely on a policy that is predefined by the doctor, and data delivered to the doctor by the patient.

Making this an analogy to the standard LIGHT^{est} scenarios would see the decision of the doctor – of whether to accept the claim of insurance of the patient – as a *trust decision*. Further, the transfer of data from the patient would be seen as the *transaction* of a *document*.

Goals

The doctor wants to specify a policy that accepts the patient as properly insured when both of the following requirements are met.

- **The patient has health insurance**. Enforcing this requires some mechanism to detect an attempt by the patient to lie about having insurance. Even a patient that actually has health insurance might try to lie – claiming to have a better provider or a better insurance plan than what is actually the case. To simplify this scenario, we assume that each health insurance provider offers exactly one insurance plan. However, we still need to be sure that a patient cannot convince the doctor that the insurance comes from a different provider.
- **The patient's health insurance provider is approved**. The point is to reject insurance providers that are not be expected to pay out any money. These can be bogus providers – made up just to pass this check – or providers that have become blacklisted, e.g. due to bankruptcy. To decide if a provider should be approved, the doctor consults a list of approved health insurance providers.

Assumptions

1. The patient has a **health insurance card** that has been issued to him/her by a health insurance provider – the *issuer*. The doctor scans the card and gets the data needed to perform the verification.
   The data is cryptographically signed by the issuer – so we call it a certificate – establishing a link between the patient and the issuer. This allows checking the first requirement above; that the patient indeed has health insurance, and that the provider is the one the patient claims.
   In practice, the certificate will contain more data than necessary for specifying this scenario – such as the name of the patient. In the following the certificate is assumed to contain at least the following data:

   - **issuer**: The name of the health insurance provider that issued the card.
   - **country**: The country in which the card is valid.

- **pub_key**: The public key that corresponds to the private key that the health insurance provider used to sign the data.
- **trust_list**: The name of the entry in the *list of approved health insurance providers* (see

Health Insurance - Natural Language Policy

The certificate *C* is proof of health insurance if (all of the below, in the given order):

- *C* has the format `healthcare_cert_format`, which is some certificate-format that the policy-executor can understand.
- The *trust_list* attribute of *C* is present. Let the corresponding value be called *SubDomain*.
- A URL is formed by combining *SubDomain* with `healthcare.trust.eu`. The URL is queried, and the result of the query is called *TrustListEntry* in the following.
- *TrustListEntry* has the format `healthcare_entry`, which is some format for such a data record that the policy-executor can understand.
- The *issuer* attribute of C is present. Let the corresponding value be called *InsuranceOrg*.
- The *entity* attribute of *TrustListEntry* is present. The corresponding value is equal to *InsuranceOrg*.
- The *country* attribute of C is present. Let the corresponding value be called *Country*.
- The *country* attribute of *TrustListEntry* is present. The corresponding value is equal to *Country*.
- The *pub_key* attribute of *TrustListEntry* is present. Let the corresponding value be called *PK*.
- The digital signature on *C* is by the private key corresponding to the public key *PK*.

below) that corresponds to the issuer.

2. A **list of approved health insurance providers** includes the health insurance provider that issued the patient's health insurance card. This fact satisfies the second requirement for the patient; that the insurance provider is approved.
   The list is hosted at a fixed, trusted location, `healthcare.trust.eu`, analogously to how trust lists and schemes are hosted in other scenarios.
   Indeed, such a list – or more likely lists from several countries – is comparable to the EU Trusted Lists. In fact, one could imagine including health insurance providers in the trusted lists among those *not qualified according to Regulation (EU) No 910/2014*.
3. It is possible to query [trust_list].`healthcare.trust.eu`, to get a record of the approved information about the issuer, where [trust_list] must be replaced by the trust_list-attribute of the certificate. The record contains at least the following data:

   - **entity**: The name of the health insurance provider in question.
   - **country**: The country in which cards issued by the insurance provider are valid.
   - **pub_key**: The public key of the insurance provider.

Policy

The policy is a specification of what certificates are acceptable proofs of health insurance. We assume that it is to be executed by some application, the policy-executor. This application is analogous to the ATV.

Here we express with two variants the same basic policy. The first variant uses natural language to express the policy, and the second uses TPL.

Notice that the natural language policy is essentially a horn-clause (except that it is not first-order logic), in the sense that is comprises exactly one conclusion whose premise is a conjunction of statements. This makes it straight forward to translate it into TPL, and the TPL-policy below has a very close correspondence with its natural language counterpart.

Also noteworthy is the use of the word “if” in the first line. The meaning here is exactly the meaning of “:-“ in TPL.

Health Insurance - TPL Policy

```
valid_health_insurance(Certificate) :-
    extract(Certificate, format, healthcare_cert_format),

    extract(Certificate, trust_list, SubDomain),
    lookup(SubDomain, "healthcare.trust.eu", TrustListEntry),
    extract(TrustListEntry, format, healthcare_entry),

    extract(Certificate, issuer, InsuranceOrg),
    extract(TrustListEntry, entity, InsuranceOrg),
    extract(Certificate, country, Country),
    extract(TrustListEntry, country, Country),

    extract(TrustListEntry, pub_key, PK),
    verify_signature(Certificate, PK).

lookup (SD,TrustList,TrustListEntry) :-
    encode(SD, TrustList, URL),
    query(URL, TrustListEntry).
```

9. Designing Translation Policies

In LIGHT[est], translation refers to the process of the Automatic Trust Verifier (ATV) when it encounters a scheme that is not directly trusted. The ATV queries the Trust Translation Authority (TTA) for a Trust Translation List (TTL), to see if there is a translation from a trusted scheme to the one that is not trusted.

The information in the TTL – what schemes and Levels of Assurance (LoAs) translate to what – we see as a translation policy. We have already discussed translation policies before. In this chapter we take a step back and discuss how to obtain such translation policies, i.e., what design considerations are involved. We note two important things here:

- The design of translation policies itself is *not an objective* of LIGHT[est]. It is however possible to assist on the design with the technologies that the project designs, as we now show. Also, this discussion is in our opinion helpful to reflect on the concepts and get a clear separation of what the project does and does not do.
- The design of a translation policy is in general *not* a *purely technical* issue that logically follows from some formal specification. First, many considerations in the design could be referring to concepts that are not entirely formalized or need to be legally interpreted, e.g., what it means to be a senior citizen. Second, there may be political considerations, e.g. bilateral agreements for the translation with certain legal guarantees. Thus, what our project can provide here is only a suggestion for a translation policy.

9.1 An Example

As a running example in this chapter, we use an eIDAS-scheme with three LoAs – *Low*, *Substantial*, and *High* – and an ISO29115-scheme with four – 1, 2, 3, and 4. In this case, a translation policy can be represented as a table, stating what LoA in one scheme translates to what LoA in the other scheme.

As an example, the table below gives a policy that

- a LoA of 1 or 2 translates to a *Low* LoA,
- a LoA of 3 translates to a *Substantial* LoA,
- and a LoA of 4 translates to a *High* LoA.

Translate From	1, 2	3	4
Translate To	Low	Substantial	High

Such a translation is straightfoward to specify in TPL, for instance:

```
iso2eidas(iso_loa1,eidas_low).
```

```
iso2eidas(iso_loa2,eidas_low).

iso2eidas(iso_loa3,eidas_substantial).

iso2eidas(iso_loa4,eidas_high).
```

More complicated translations may be then expressed between schemes with several attributes. Mind that this translation has a direction, namely from ISO to eIDAS, i.e., it is not supposed to be applied for translating from eIDAS to ISO (since that has not even a unique answer in every case). TPL specification allow however to be evaluated in both directions, i.e., we can query what ISO-levels would be translated eIDAS level low. The query:

```
iso2eidas(X,eidas_low).
```

would yield the answers `X=iso_loa1` and `X=iso_loa2`.

In fact, the translation in the other direction could be entirely different. For example, a Substantial LoA in the eIDAS-scheme may not be enough to get a LoA of 3 in the ISO29115-scheme.

9.2 Designing Translations: the Rationale behind a Policy

An important point is now that the translation *only* depends on information is present in the credential, in the example the level of assurance information in the given ISO credential. The reason why ISO level 2 is translated to eIDAS `low` and not eIDAS `substantial` is *not* formalized in this policy -- and it should not be the concern of the ATV for instance. Rather, this is a concern in the design of the translation policy that may be based on all aspects of the two credential schemes, in particular the *issuing* process. This is because issuing depends on properties of the entity that the credential is being issued for, e.g., whether it is a natural person and whether this person showed in person to the issuing. Such properties may have an influence on the assurance level that this person will obtain, but it may not be an *attribute* of the credential, so from the issued credential it is not (directly) visible whether it is a person who showed up in person. Thus, the policy cannot (directly) refer to such a property that is not reflected in the credential. It may however, be a design consideration for the translation policy. In a nutshell, the rationale could be:

Credential C_A in scheme A should be translated to credential C_B in scheme B, if every entity who receives credential C_A in scheme A would get in scheme B the credential C_B *or better*.

This rationale requires several notions:

- There is a total ordering on the credentials of scheme B, expressing what is better. (This should refer to only the ordinal aspects of credentials, not on other information like bearer name etc.)

- It requires that all properties of the issuing process of the credentials are sufficiently formalized and comparable between the two schemes. In fact, this is a question (partially) answered by the work on vocabularies and description of credential schemes.

We now show how to represent the above rationale in LIGHT^{est} for any trust scheme that is sufficiently formalized in TPL.

9.3 The Issuing Process

Comparing credentials can be done by considering the details of issuance. Two schemes may have similar requirements for issuing credentials, and based on that decide to recognize the other scheme in a translation policy.

To talk about what requirements are met, we introduce the concept of a *scenario* as the input to the issuing process. A scenario assigns values to all properties relevant to the issuing process. A scheme can then decide to recognize a credential from a different scheme by evaluating all scenarios that could lead to that credential being issued.

In the following we see a scheme as a tuple $s_A = (A, issue_A)$, where A is the set of all credentials that s_A can issue, and $issue_A$ is a function, $S \rightarrow A$, that maps a scenario to a credential.

As an example, to issue a credential to a person, a scheme may require that person to show up in person and present a passport. In this case, the scenario defines the Boolean *in_person*- and *passport*-properties to indicate whether those requirements are met.

Example – Issuing

The function $issue_A$ of the scheme s_A is defined as follows.

$$issue_A(in\_person, passport) = \begin{cases} \text{LoA High} & \text{if } in\_person = t \wedge passport = t \\ \text{LoA Low} & \text{otherwise if } in\_person = t \\ \bot & \text{otherwise} \end{cases}$$

This s_A issues a certificate with a High LoA if both properties are true and otherwise it issues a Low LoA if *in_person* is true. If none of these requirements are met, s_A issues no certificate, indicated with $\bot$.

9.4 Translation

This section discusses how to specify translation from a scheme s_A to another s_B. The translation is a function, f_{AB}: $A \rightarrow B$, from credentials of the first scheme to credentials of the second scheme. The idea is to express f_{AB} in general, so that the same definition can be reused every time a new translation policy is to be defined.

One application opportunity is a tool that will automatically produce (or recommend) a translation policy. The only requirement would be to define the two input schemes in a compatible way by defining the *issue*-functions.

The tool is out of the scope of LIGHT^{est} to provide. We mention it here as a possible use-case worth considering.

The idea is to define f_{AB} so that the credential it outputs is determined by the set of scenarios that the input-credential can be issued from. The problem is that these scenarios may correspond to more than one output-credential in the output-scheme.

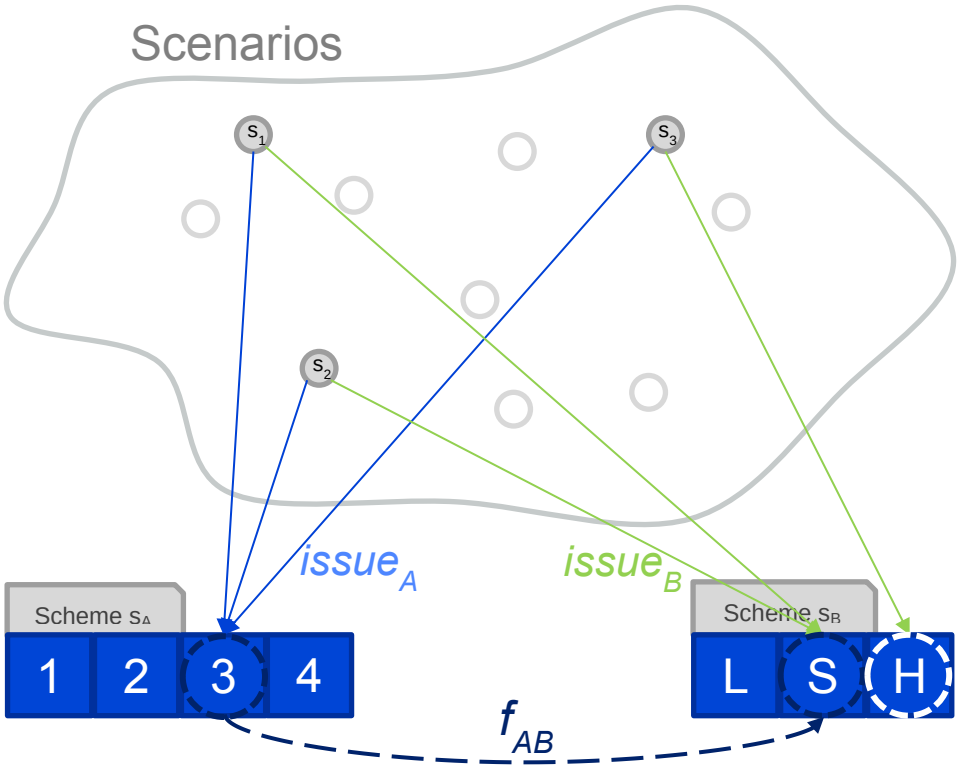

Figure 1 Translation

The figure above gives an example. In it, a credential with the LoA 3 can be issued in the scheme s_A from three scenarios: s_1, s_2, and s_3. However, in a second scheme, s_B, the scenarios s_1 and s_2 would be issued to a credential with the LoA Substantial, but s_3 would be issued to the LoA High.

To state this more generally, assume that the credential *a* translates to the credential *b*. Then it does not in general hold that the preimages of *a* and *b* under $issue_A$ and $issue_B$ are equal. That is, assuming:

$f_{AB}(a) = b$,
$a^{-1} = \{x | issue_A(x) = a\}$, and
$b^{-1} = \{x | issue_B(x) = b\}$,

it does *not* in general follow that $a^{-1} = b^{-1}$.

To make sure that f_{AB} never gives more than one result, we define it as the minimum among the possibilities:

$$f_{AB}(a) = \min\{issue_B(s) \mid issue_A(s) = a\}$$

This definition assumes a complete ordering on the credentials. The LoAs of eIDAS or ISO29115 are examples of concepts that define such an ordering.

However, even if the credentials are not completely ordered, it may still be possible to choose *some* minimal credential, and use it in application of the above definition. For example, if the credentials are partially ordered and a least credential (e.g. ⊥, meaning no credential issued) exists, the definition of f_{AB} can be applied with the *min* replaced by *greatest-lower-bound*.

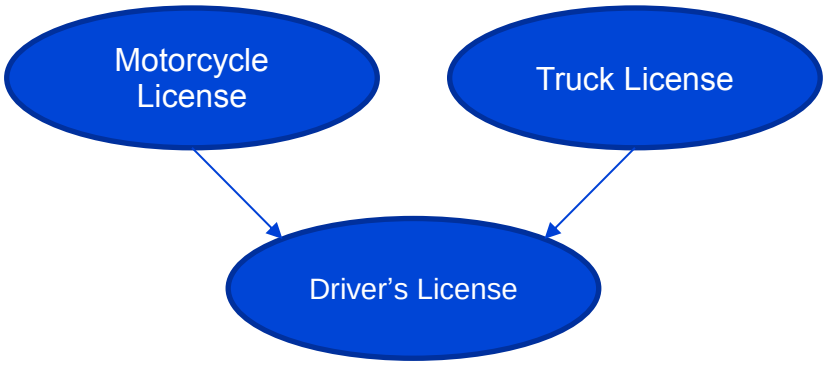

Figure 2 Partially Ordered Credentials

Example – Partially Ordered Credentials

As an example of credentials that are partially ordered, consider credentials that represent driver's licenses. Let us consider a scheme that issues three types of licenses; a regular driver's license, a motorcycle license, and a truck license. Further, let us assume that the motorcycle- and truck-licenses can be used whenever the regular driver's license can be used, but neither of the two can replace the other.

So the motorcycle- and truck-licenses are not ordered with respect to each other, but both are "greater" than the regular driver's license.

Now consider a translation from a credential of another scheme into such a license. If the result should be the motorcycle license or the truck license depending on the scenario the credential was issued from, then f_{AB} should return the minimum of the two; which is impossible because they are not comparable. So the definition of f_{AB} does not provide a translation. However, taking the *min* in the definition of f_{AB} to mean greatest-lower-bound will fix this. Then f_{AB} returns the regular driver's license.

Example – Translation

In this example the space of scenarios has three Boolean properties, $S = \{B \times B \times N\}$, *in_person*, *passport*, and *age*. The schemes s_A and s_B are different in the condition for issuing the highest level credential:

$$issue_A(in\_person, passport, age) = \begin{cases} \text{LoA 2} & \text{if } in\_person = t \wedge age \geq 20 \\ \text{LoA 1} & \text{otherwise if } in\_person = t \wedge age \geq 18 \\ \text{LoA 0} & \text{otherwise} \end{cases}$$

$$issue_B(in\_person, passport, age) = \begin{cases} \text{LA} & \text{if } in\_person = t \wedge age \geq 21 \\ \text{LB} & \text{otherwise if } in\_person = t \wedge age \geq 18 \\ \perp & \text{otherwise} \end{cases}$$

In this case it is not possible to get a LA credential by translating from a credential issued under scheme s_A – even if the holder of the credential is actually 21 years old or older.

The above example on translation makes it impossible to obtain by translation the credential that certifies the holder as at least 21 years old. This is because the other scheme, which we are translating from, only has credentials that certify a lower age.

Given that the age in question is 21, making an exception allowing those who are at least 20 to get that credential is probably a bad idea because of the legal implications. But if the age limit was set more arbitrarily – for example an age limit that decides who is entitled to a senior rebate – then a more lax policy might be preferable.

The view of *Translation Policies we* present here allows for that. Authorities can design a translation policy as they see fit – possibly by use of a tool based on f_{AB} that provides a draft. The LIGHT^{est} Automatic Trust Verifier (ATV) can then query the translation policy.

Another thing that is interesting to consider, is making the ATV responsible for executing f_{AB}. Then the Issuing Authorities are only responsible for defining their own schemes (and not how they translate to other schemes), and the ATV must query the Trust Translation Authority for this information. Note that the information must be defined by the Issuing Authorities in such a way that it is comparable to information from other Issuing Authorities.

10. GUI Mockup

This is a quick mockup of a graphical editor for translation of trust schemes in LIGHT^{est} -- similar ideas may be used for the specification of trust policies itself or other translations.

At the beginning, we shall have the credentials between which the translation should be defined side by side -- we only consider here the most general case of tuple-based credentials, the other cases are of course similar. So let us consider two imaginary credential types "EasyTrustIdentity" and "QuatschID".

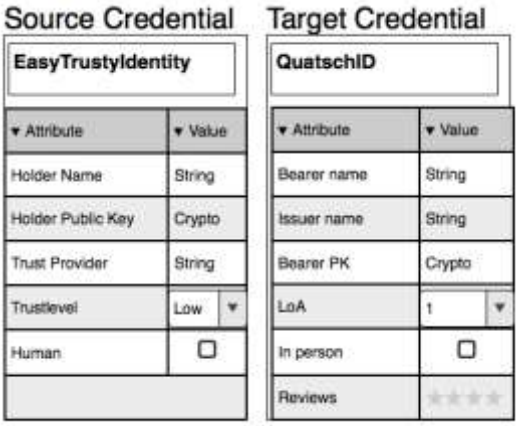

Figure 3 Credential View

They have several attributes of several types, where "String" is a human-readable text, while "Crypto" stands for some bitstring that is not intended for the human eye. For each credential type we may define special types like some ordinal value (with an order defined, e.g. Low<Medium<High or A>B>C). Some values may have also a "safe default", e.g. Trust-level "Low", i.e., when nothing better is known, we can assume that default. Note that such a default can be tricky, e.g. 0 of 4 stars as "Reviews" may be considered a terrible value.

First we may define how the fields correspond to each other:

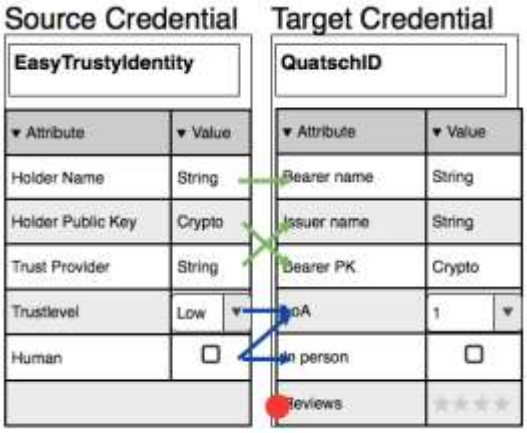

Figure 4 Credential Translation

Here the green arrows -- note this is only a first mockup, and other visualizations may be chosen -- denote that a field on the source credential literally carries over to the target credential.

The blue arrow is the more complicated case "it depends" ... here the translation is specified for several values in detail. The red dot indicates that here we have no translation, so a safe default must be taken.

The more complicated translation indicated by the blue arrows is now in our opinion best described by a set of different cases. Let us have as case one, that "Trustlevel low" is translated into "LoA 1":

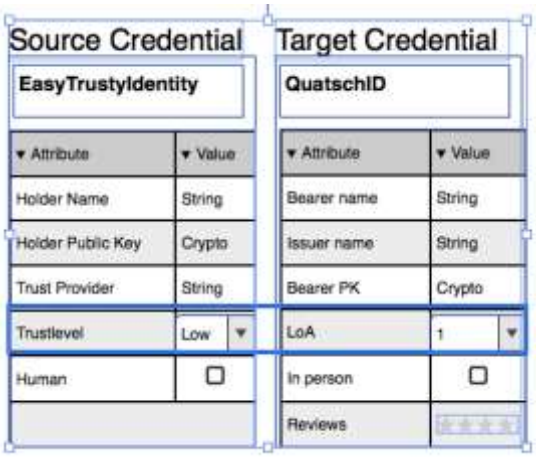

Figure 5 Translation from Low to 1

The next case is that "Trustlevel Medium" translated to "LoA 2":

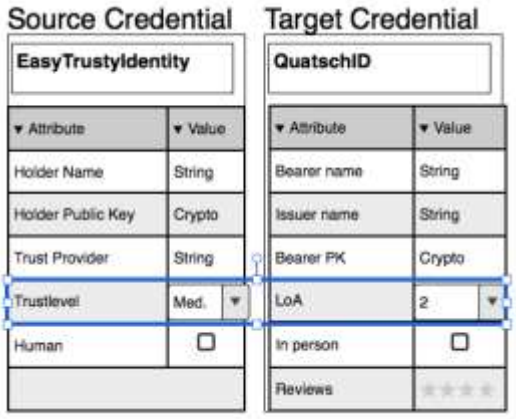

Figure 6 Translation from Medium to 2

Finally, "Trustlevel High" translates to "Loa 3" but *only* if also the attribute "Human" is true in the source. And then also "In person" will be true in the translation:

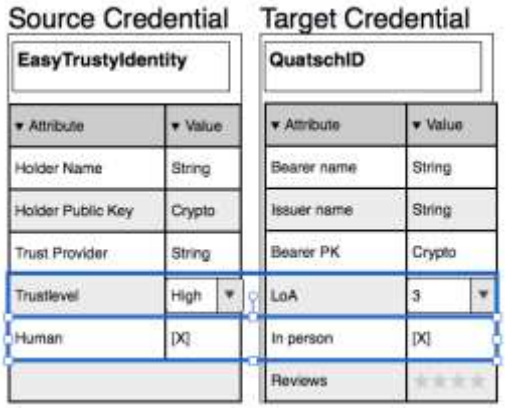

Figure 7 Translation from High to 3

This is of course just a made-up example, but it should illustrate the most common translation:

- Specify on the target side the case you want to translate to
- Then select on the source side what is at least required for it.
- One may also distinguish between attributes on the source side to have exactly that value vs. "or better". The latter is only possible if there is an order on the values, e.g. in the source example we silently assumed that " Human [X]" is better than "Human [ ]".

11. References

Almousa, O., Mödersheim, S., & Viganò, L. (2015). Alice and Bob: Reconciling Formal Models and Implementation. *Festschrift in honor of Pierpaolo Degano, 2015.*

Almousa, O., Mödersheim, S., Modesti, P., & Viganò, L. (2015). Typing and Compositionality for Security Protocols: A Generalization to the Geometric Fragment. In *Computer Security -- ESORICS 2015: 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part II* (pp. 209--229).

Becker, M., Fournet, C., & Gordon, A. (2010). SecPAL: Design and Semantics of a Decentralized Authorization Language. *J. Comput. Secur.*, 619--665.

Blass, A., Caso, G. d., & Gurevich, Y. (2012). *An Introduction to DKAL.* Microsoft Research.

FutureID-Project. (2014). *D42.3 - APS Definition Functions.*

FutureID-Project. (2015). *D42.8 APS Files for Selected Authentication Protocols.*

Hinrichs, T., & Genesereth, M. (2006). *Herbrand Logic.* Stanford.

Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2003). *Introduction to automata theory, languages, and computation - international edition (2. ed).* Addison-Wesley.

Mejborn, B. K. (2016). *ASN.2: A model-driven approach to secure protocol implementation.*

Mödersheim, S., & Katsoris, G. (2014). A Sound Abstraction of the Parsing Problem. *{IEEE} 27th Computer Security Foundations Symposium*, (pp. 259--273). Vienna.

Programming languages, t. e. (1995). *13211-1:1995, ISO/IEC Information technology - Programming languages - Prolog - General core.* Geneva, Switzerland: International Organization for Standardization.

12. Project Description

LIGHTest project to build a global trust infrastructure that enables electronic transactions in a wide variety of applications

An ever increasing number of transactions are conducted virtually over the Internet. How can you be sure that the person making the transaction is who they say they are? The EU-funded project LIGHTest addresses this issue by creating a global trust infrastructure. It will provide a solution that allows one to distinguish legitimate identities from frauds. This is key in being able to bring an efficiency of electronic transactions to a wide application field ranging from simple verification of electronic signatures, over eProcurement, eJustice, eHealth, and law enforcement, up to the verification of trust in sensors and devices in the Internet of Things.

Traditionally, we often knew our business partners personally, which meant that impersonation and fraud were uncommon. Whether regarding the single European market place or on a Global scale, there is an increasing amount of electronic transactions that are becoming a part of peoples everyday lives, where decisions on establishing who is on the other end of the transaction is important. Clearly, it is necessary to have assistance from authorities to certify trustworthy electronic identities. This has already been done. For example, the EC and Member States have legally binding electronic signatures. But how can we query such authorities in a secure manner? With the current lack of a worldwide standard for publishing and querying trust information, this would be a prohibitively complex leading to verifiers having to deal with a high number of formats and protocols.

The EU-funded LIGHTest project attempts to solve this problem by building a global trust infrastructure where arbitrary authorities can publish their trust information. Setting up a global infrastructure is an ambitious objective; however, given the already existing infrastructure, organization, governance and security standards of the Internet Domain Name System, it is with confidence that this is possible. The EC and Member States can use this to publish lists of qualified trust services, as business registrars and authorities can in health, law enforcement and justice. In the private sector, this can be used to establish trust in inter-banking, international trade, shipping, business reputation and credit rating. Companies, administrations, and citizens can then use LIGHTest open source software to easily query this trust information to verify trust in simple signed documents or multi-faceted complex transactions.

The three-year LIGHTest project starts on September 1st and has an estimated cost of almost 9 Million Euros. It is partially funded by the European Union's Horizon 2020 research and innovation programme under G.A. No. 700321. The LIGHTest consortium consists of 14 partners from 9 European countries and is coordinated by Fraunhofer-Gesellschaft. To reach out beyond Europe, LIGHTest attempts to build up a global community based on international standards and open source software.

The partners are ATOS (ES), Time Lex (BE), Technische Universität Graz (AT),EEMA (BE), G&D (DE), Danmarks tekniske Universitet (DK), TUBITAK (TR), Universität Stuttgart (DE), Open Identity Exchange (GB), NLNet Labs (NL), CORREOS (ES), IBM Danmark (DK) and Globalsign (FI). The Fraunhofer IAO provides the vision and architecture for the project and is responsible for both, its management and the technical coordination.

The Fraunhofer IAO provides the vision and architecture for the project and is responsible for both, its management and the technical coordination.