D2.5
Formal Description and Analysis of Concepts II

Document Identification	
Date	August 30, 2018
Status	Draft
Version	Version 1.0

Related WP	WP 2,3,4,5	**Related Deliverable(s)**	D2.14, D3.1, D3.2, D4.1, D4.2, D5.1, D5.2
Lead Author	Sebastian Mödersheim	**Dissemination Level**	CO
Lead Participant	Sebastian Mödersheim DTU	**Contributors**	Bihang Ni DTU Rasmus Birkedal DTU
Reviewers	David Hixon (CET) Jesse Kurtto (Ubisecure)		

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

The deliverables D2.4, D2.5, and D2.6 represent the formal basis of the LIGHT^{est} project, where D2.4 was issued at the end of the first project year, this here D2.5 is the update after the second project year, and D2.6 will be the final version at the end of the project [6]. 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 LIGHT^{est} 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 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 (on D2.4)	DTU

2.2 History

Version	Date	Author	Changes
0.1	2018-07-046	The Author	1st Draft

3 Table of Contents

3.1 List of Figures

4 Trust Policy Language TPL

This deliverable represents the formal basis of the LIGHT^{est} project. At the core, this means to develop a formal language for describing the different aspects of the LIGHT^{est} system, 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 LIGHT^{est} 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 visual languages for end users that can handle most of the common specifications and can be easily mapped into TPL. In this way TPL is a basis and frame for the entire project.

4.1 TPL Design Decisions

The LIGHT^{est} Trust Policy Langauge, or TPL for short is the language for defining trust schemes, trust translation schemes, and delegation schemes. It allows us to integrate the different parts of the project in a precise and uniform way. As it has a semantics based on mathematical logic, it serves as an unambiguous reference, namely for testing and formaly proving the correctness of any implementations based on TPL, such as the ATV.

The core of TPL is rather simple: it is Horn clauses, in the syntax of the language Prolog. This is inspired by similar languages for access control based on Horn clauses such as SecPAL and DKAL [5, 1]. The particular benefits from using Horn clauses are:

The language is based on the existing programming language Prolog: this allows us to draw both from the rich research on logic programming (including existing interpreters) and thus even have an executable language, i.e., it can be used as a reference implementation of the ATV, for testing, or even be the basis of the ATV.

There are several policy languages for access control that are similarly based on Horn clauses, for instance Secpal and DKAL.

Key points in favour of this choice are:

- Many policies in practice are just simple enumerations of cases. They are trivial to formalize as Horn clauses (as enumerations of facts).

- On the other extreme, Horn clauses are Turing complete, i.e., every algorithm can be written using Horn clauses, so the language will never limit us by lack of expressive power.
- It is most suitable for many concepts that regularly arise in policies, both classical access control and trust. First, one can easily describe the logical relationships between several criteria, for instance, when an order is above a certain value, stricter criteria have to be fulfilled. Similarly, Horn clauses are ideal for describing delegation.
- There use of negation is very limited; like Prolog we allow for a non-monotonic not, but allow its use only for concepts like black listing: one can make non-membership in such a list a criterion. However, one can not, for instance make a policy that an entity is *unable* to present a certain certificate. While this limited support for negation can make some specifications more difficult, the advantage is that the combination of policies can never be undefined or require multi-valued logics.
- Horn clauses directly have a clear *logical* semantics, namely as a conjunction of first-order formulae.
- Drawing from the research in logic programming and the Prolog language in particular, it is also clear how to implement automated algorithms to evaluate a TPL policy, i.e., the language immediately comes with a truely *executable* semantics. This allows us to use existing Prolog interpreters for testing and designing prototype implementations.
- There is a variety of possibilities in the evaluation of Horn clauses, thanks to their logical nature. For instance, we are not limited to checking whether a policy is satisfied, we can also reason about policies, e.g., is a certain translation policy in some sense ambiguous.
- The evaluation can be made to directly trigger necessary queries to servers, e.g., using DNSSec, and process their answer; thus the ATV can directly be encoded, either as a prototype/testing reference or even as the final product. This means linking logical with procedural aspects of policies; also here the rich experience with Prolog can be utilized.
- 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.

While we allow with TPL quite complex forms of reasoning about trust, we want to emphasize that the policies will in the vast majority of cases rather simple: it is in the hand of each business to set the policy what they are willing to accept, e.g., whether they even want to accept trust translations or not. Of course, LIGHT^{est}, TPL and the ATV do not require or just advocate complex reasoning like trust translation or (chains of) delegation. However, LIGHT^{est} does allow for complicated policies, and both for simple and complex policies, we want specification languages that are clear and unambigious, and that can be rigorously checked by the ATV.

4.2 A Gentle Introduction to TPL

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 :- requirement1, ..., requirementN.
```

One may simply read :- as the word "if" and the commas as "and", thus "the conclusion holds, if all the requirements hold". We will also call the conclusion the *left-hand side* or *head* of the clause, and the requirements the *right-hand side* or *body* of the clause. Note that this is a logical implication, not an equivalence: if the requirements do not hold, the clause does not tells us whether the conclusion holds or not.

Consider as a specific example the sentence "I trust X, if X is a delegate of Y and I trust Y.". This could be expressed as a Horn clause in the following way:

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

Here, we have introduced as new vocabulary the *predicates* trust and delegate that have no built-in meaning in TPL; in fact, they obtain their meaning from the clauses we specify. The meaning of the *variables* X and Y is that they can be replaced by any term and the clause applies. For instance, suppose trust(a) and delegate(b,a) already hold, then we can derive trust(b), because we can *instantiate* X=b and Y=a and apply the rule.

In fact, we will typically have a basis of clauses like trust(a) that already hold initially; they are written as *facts*, i.e., they have no right-hand side:

```
trust(a).
delegate(b,a).
```

Suppose we also add delegate(c,b), then we can also derive trust(c), i.e., we can form arbitrary long chains of delegation. If this is not desired in a policy, one can introduce different predicates for trust, distinguishing whether someone is trusted directly or through delegation. For the sake of this example, let use trust(X) for the direct trust and introduce a new predicate trustD(X,N) for trust through delegation; here N denotes the number of levels of delegation, i.e., trust(X) is equivalent to trustD(X,0). Then we can specify delegation as the following Horn clauses:

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

Thus we can now derive in this example:

```
trustD(a,0).
trustD(b,1).
trustD(c,2).
```

We can now easily express the policy for accepting an electronic purchase based on delegation level and the amount of the purchase, e.g. below 100 Euro any delegation level is fine, below 1000 Euro at most one delegation level, and up to 1 Mio. Euro we do not accept delegation. This is formalized as:

```
order(X,M) :- M<=100, trustD(X,N).
order(X,M) :- M<=1000, trustD(X,N), N<=1.
order(X,M) :- M<=1000000, trustD(X,0).
```

Note that logically the order of the clauses does not matter, and neither does the order of the requirements in the body of a clause. However, in a typical Prolog interpreter this order does matter and we shall use this now as a glimpse into the typical working of an interpreter.

The interpreter is fed with a rule base, a set of Horn clauses (including facts) and a *query* which can be any predicate (even containing variables). For instance consider the query order(a,300). The interpreter goes through the clauses and takes the first one whoes left-hand side matches the query; in our case that is the clause for purchases up to 100 Euro. It now checks the requirements in the order they are specified. In the example this fails at the first requirement since the query is for more than 100 Euro. In such a case the interpreter continues with the next matching clause, here the one for 1000 Euro; the first requirement is obviously satisfied. For the second one, the interpreter must start a new query, namely whether trustD(a,N). Note that this query now contains a variable, and thus we are asking if a is trusted in *any* number of delegations. Indeed that will return trustD(a,0) and thus set N=0. With this, also the third requirement of the clause is met.

We are also able to formulate queries like order(X,300), i.e., who is allowed to make orders of 300 Euro. We will obtain in fact two answers, namely X=a and X=b, since both fulfill trustD(X,N),N<=1. In this way we can use the interpreter also to reason about policies and make policy testing, e.g., for a user to check that the policy indeed reflects what they wanted to express.

4.3 A Detailed Definition of TPL

Prolog is quite liberal on the use of function and predicate symbols: there is no type system, there is no distinction between functions and predicate, and symbols can be used with different arity (number of arguments) throughout the code. We deliberately restrict this here as it can lead to bugs that are hard to find, but is not usually make specifications easier, thus we define that we have the following disjoint sets of symbols:

- Variable symbols: they start with upper-case letters.
- Function symbols: they start with lower-case letters and have a fixed arity. Constants are a special case that have zero arguments.
- Predicate symbols: they also start with lower-case letters and have a fixed arity.

Terms are now inductively defined as the least set that contains variables, and for every function symbol f of arity n, if $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term. Further if $t_1, \ldots, t_n$ are terms and p is a predicate symbol of arity n, then $p(t_1, \ldots, t_n)$ is a predicate. A clause is of the form $p{:-}p_1, \ldots, p_n$. where p and the p_i are predicates. Here we will usually require that the variables occuring in p are a subset of the variables occuring in the p_i.

We do not require the user to declare the function and predicate symbols explicitly, but this is infered by the interpreter and checked for consistency in the specification. For example, if the specification contains the fact signature(privkey(alice),X). then the interpreter infers that signature is a binary predicate symbol, privkey is a unary function symbol, alice is a constant and X is a variable.

Formal Semantics as an Interpreter

We here briefly sketch two ways to formally define the meaning of TPL. This is only a sketch since the semantics of Prolog has been described in detail. 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. The interpreter basically works as follows:

- We pick the first predicate q from the query and check the first Horn clause of the form $p{:-}p_1,\ldots,p_n$ such that p and q have a *unifier*, i.e., a substitution of the variables that makes p and q 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 q in the query by $p_1,\ldots,p_n$ and apply σ to the entire query.
- If the resulting query is empty (no more predicates to satisfy), then we have found a solution. Otherwise, we recursively start the same procedure for the new query.
- 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( f(X) ).
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(f(Y))),... 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 constructs that allow to construct potentially infinite loops. 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.

Logical Semantics

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 ← (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 $q_1, \ldots, q_n$, the solutions are those substitutions σ of the variables in the q_i such that $H \vdash \sigma(q1), \ldots, \sigma(q_n)$ 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.

Arithmetic

There are two extensions of Horn clauses that both Prolog and TPL use. First, we may also use built-in datatypes and functions, like the binary operator + and the binary predicate ≥ 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 [2].

Blacklisting

The second extension is the use of not(p) (where p is a predicate) in queries and the body of Horn clauses. In fact, with negation in the body, we *are* 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 first (recursively) tries 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 as the following example shows:

```
trust(X) :- not(blacklisted(X)), positiveCriteria(X).
blacklisted(malice).
blacklisted(horst).
```

On the query trust(alice), the interpreter first checks whether blacklisted(alice) can be derived. Suppose there are no more predicates on blacklisted than shown here, then that query fails, so not(blacklisted(alice)) is satisfied and the compiler continues to check the remaining positiveCriteria(alice).

The introduction of negation has an important effect on the semantics of policies as we discuss next.

Monotonicity

Without negation, we have the montonicity property of classical logic: we add new clauses to our knowledge base this can only increase the number of statements we can derive. In fact, all the Horn clauses specify positively what can be derived. Using Horn clauses for trust policies (and similarly for access control policies) means that we positively define under which conditions somebody is trusted, never negatively under which conditions somebody is distrusted. Thus also if somebody is trusted and we add new policy rules (without changing existing ones) then this entity is still trusted afterwards. Note this also means a *default-deny* principle: if we cannot derive that somebody is trusted, then this entity is not trusted. This can be for lack of a positive proof that some criterion is met, and thus the entity may "come back" with additional certificates for instance and then fulfill the policy.

This enforcement to specify policies only positively with a default-deny otherwise sidesteps many problems of other policy languages that allow both rules for allowing and denying access and that then also need to specify how to resolve if one policy rule is positive and another is negative. Indeed we believe that in practice of trust policies we can specify everything positively without too much trouble—except for blacklisting. Thus we have made the exception for blacklisting. We recommend the following restriction though: negation shall only be used on predicates that are specified only as a set of ground facts (as the list of blacklisted entities in the example above).

Note that blacklisting makes everything non-monotonic: adding the fact `blacklisted(alice)` destroys the proof that `alice` is trusted. Model-theoretic or proof-theoretic formalizations of this phenomenon are pretty ugly, so we omit this here.

4.4 Example: Assurance Level

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

```
trust(X) :- loa(X,L),L>1.
loa(X,3) :- quality(X,good).
loa(a,2).
loa(b,1).
quality(c,good).
```

This formalizes first two policy rules: that we trust every X that has a level of assurance loa greater 1. Second, we specify that every X for which quality holds with level good has loa (level of assurance) 3. Note that none of the predicates trust, loa, or quality has any "built-in" meaning, but its meaning arises only from the rules. The last three facts assert an example scenario where a has level 2, b has level 1 and c has good quality, so we can conclude, together with our policy that loa(c,3), and thus that we trust both a, b and c.

We would like to formulate now, 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 the evaluation may run into an infinite loop at this point.

4.5 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 a 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 worried 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 any rules for the predicate uniq, but it just represents the unformalized 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, l3req, and 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. To 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).
```

5 Formats

The policies will be working on data with a variety of concrete data formats, from XML certificates, DNS resource records to custom data formats for electronic formats. In order to work with these data but avoiding low-level details like parsing into the policies, we use an abstract syntax interface to the concrete formats. This also allows one to use general solutions to prevent injection and overflow attacks entirely.

This interface between concrete syntax (the actual bytestrings) and the abstract syntax shall be a seperate part of a TPL specification. This section outlines a module for TPL for specifying abstract and concrete syntax and their relationship. Basically, what one ultimately needs to implement is two things:

- a parser that reads a bytestring in concrete syntax and extracts the abstract syntax, and
- a pretty printer that produces from the abstract syntax the concrete syntax again.

First, for many formats (like XML) there are already implementations like XML libraries; given that they are implemented without any security problems, they can often serve as a good basis. What is often lacking, though, is a proper abstract data type for the parser to return. Also, one may implement parser manually, but it is much more convenient to have a general method of specifying a description of concrete and abstract syntax that allows for an automated generation of the parser and pretty-printer.

To this end, we use an idea from the field of specifying security protocols. It allows us to 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:

$$\mathit{TLSclientHello}(T, R, S, \mathit{Cipher}, \mathit{Comp})$$

The actual message on the string level would be however:

```
'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 the example, 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 [4]. 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. Thus, 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).

The reasonable properties that this soundness result requires are:

- Each format f 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 *pairwise disjoint*: if $f_1(t_1, ..., t_n) = f_2(s_1, \ldots, s_m)$ then $f_1 = f_2$ (and thus $n = m$, as well as $t_1 = s_1, \ldots, t_n = s_n$ by unambiguity), i.e., no byte string can be parsed as more than one format.

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 <element>...</element>, 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 another important point: 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 support formats with a fixed number of elements [3].

5.1 Interfacing to TPL

Generally, we like to assume that a format consists of a number of attribute-value pairs, i.e., and we can imagine this like a paper form that has several *fields*, where each field is clearly identified by an attribute name and one can fill in an attribute value. For instance an XML-based certificate may have the following format:

```
<cert>
  <firstname>Jane</firstname>
  <lastname>Doe</lastname>
  <dateofbirth>...</dateofbirth>
```

```
  ...
</cert>
```

Another format may structure the same information in a different way, e.g., like X.509 certificates where no explicit text identifies the fields like firstname, but the format itself defines which bytes of the text mean which field.

To work with such certificate in TPL without bothering about the concrete syntax of a format we assume to have a predicate extract (that is linked to a parser implementation) and that has three arguments: an instance of the format, an attribute, and the value for it, e.g., if C is a certificate of the sketched XML type, we could work with it in TPL as follows:

```
over18policy(C) :-
  extract(C, dateofbirth ,D),
  today(T), addyear(D,18,D2), D2<=T.
```

where we assume today(T) is true for the current day at the execution of the policy, and addyear and <= work on arithmetic for the date datatype as expected.

Additionally, we shall assume that extract has a special attribute format that checks which kind of format a particular text is. For instance in the above we may additionally require on the right-hand side the fact extract(C,format,myXMLcertificateType) where myXMLcertificateType is the identifier for our example XML certificate format. This special *format* field of course assumes that all formats are disjoint, i.e., that there is no bytestring that matches more than one format. In practice this might not always be the case, when in specific contexts some formats are used that are not disjoint to all formats of other contexts. However, we can still "extract" the kind of the certificate within a particular context, as long as we know all kinds of formats that occur in this context by a predicate inContext(K) and if these are pairwise disjoint; namely by writing extract(C,format,K),inContext(K).

We have here assumed so far all formats to be essentially a list of attribute-value pairs. There are some extensions relevant in general: there may be optional attributes, and attributes where the value is itself a structured datatype, e.g., a list of arbitrary length. Note that all this can be implemented by corresponding parsers and pretty printers and the access through the extract predicate can be uniform.

An interesting extension though for TPL that arises from this is a type-system where we check that data is always handled with appropriate predicates. Such a type system is subject of future work.

6 Interfacing to DNSSec

The trust policies will often involve the requirement that the issuer of a given certificate is member of a particular trust list. This actually means that as part of checking the policy, one has to interact with a foreign server to obtain trust list entries. TPL allows us to formulate such a check as part of a policy, however such a mix of policy rules and interaction with third parties may raise some concern, and we will discuss this below in section 7.5.

The general setup, as far as we are concerned here, is that trust lists are stored relative to a domain, such as qualified.trust.admin.eu for the trustlist of qualified eIDAS signers. We do not want to have to download the entire trust list to check if somebody is a member, thus we assume every entry to have a particular search-key in form of a subdomain, for instance

PX2NO4LVPA4WHCBLYXHIKRWVRE.qualified.trust.admin.eu

We assume that we can lookup such a URL to get a trust list entry using a predicate

```
lookup(URL, TrustListEntry)
```

Note that this predicate triggers a server lookup, i.e., at the moment the policy checker (the TPL interpreter) reaches this lookup predicate, it interacts with the server. The predicate will succeed and return the respective TrustListEntry if this entry indeed exists, and otherwise the predicate will fail. It can thus act as a requirement in a policy.

Since a lookup query to a server is a time consuming task (as compared to the other checks that are made locally), there is the risk that an inexperienced user specifies policies that get stuck in checking many queries repeatedly. For this reason, we may integrate a caching mechanism into the lookup predicate: all queries done over a certain period (say 5 minutes) are stored with their results, so that repeated queries within the time frame are answered from the cached result.

A certificate can now claim a trust list membership, for instance qualified signatures. For that, it shall include a field trustList that contains the URL for looking up the membership e.g. PX2NO4LVPA4WHCBLYXHIKRWVRE.qualified.trust.admin.eu.

Pitfall

Note that there is a potential pitfall here: when the designer of a policy is not careful, it may happen that they just extract the trustlist membership claim from a certificate and use the lookup function to just just check the claim, for instance:

```
myfirstpolicy(Certificate) :-
  extract(Certificate,trustList,TrustList),
  lookup(TrustList,TrustListEntry),
  % followed by some checks on the TrustListEntry.
```

Even though the expectation of the modeller is that the trust list is, say, qualified.trust.admin.eu, there is nothing in this policy that checks that: this policy takes just whatever URL is contained

in the document and queries that server, even if the URL is, say, trustme.attackerspace.tk. The checking of the entries on that trust list is probably irrelevant. While it may still be obvious in such a small policy, such an omission can easily slip in a more complex policy. We shall make special checks in TPL to warn the user about such a specification.

For easily checking trust schemes, we thus assume another predicate trustscheme that relates a URL with the trust scheme it belongs to. For this, we assume a fixed association of trust schemes with particular URLs, so that users do not need to spell out URLs in their policies with all the potential vulnerabilities that come with that. For instance, we may have that

```
trustscheme(URL, eIDAS_qualified)
```

is true if and only if URL has "qualified.trust.admin.eu" as a suffix.

7 Examples

We first illustrate TPL with the formalization of three basic scenarios inspired by the LIGHT^{est} architecture deliverable.

7.1 Boolean Trust Scheme without Translation

The first scenario/policy is to check a document Document that was supposedly signed by Signer who has supplied a certificate Certificate which should prove that the Signer can make eIDAS qualified signatures:

```
checkQualifiedSignature(Document, Certificate, Signer) :-
  % Checking the certificate:
  extract(Certificate, format, eIDAS_qualified_certificate),
  extract(Certificate, issuer, Issuer),
  extract(Certificate, bearer, Signer),
  extract(Certificate, pubKey, PkSig),
  extract(Certificate, issuerKey, PkIss),
  extract(Certificate, trustList, TrustMemClaim),
  % check the document was indeed signed with PkSig:
  verify_signature(Document, PkSig),
  % check the certificate is indeed signed with PkIss:
  verify_signature(Certificate, PkIss),
  % check the claimed trustlist membership is eIDAS qualified:
  trustscheme(TrustMemClaim, eIDAS_qualified)
  % check the Signer is really on the that trustlist:
  lookup(TrustMemClaim, TrustListEntry),
  % check that the issuers key is indeed PkIss
  extract(TrustListEntry, pubKey, PkIss).
```

Note that this is assuming that both Document and Certificate are signed messages, which we assume to be basically clear-text formats (so we can apply extract) and the format has a special field that contains a hash of the entire message under a signature algorithm. The verification of this signature with respect to a given public key we formalize in the predicate verify_signature.

The policy thus first requires that the format of the certificate is indeed a certificate for an eIDAS qualified signature and we extract the following information from it: the issuer (the entity who has signed the certificate), the bearer (the entity who owns the certificate), the pubKey (the public key of the bearer), the issuerKey (the public key of the issuer; this may be implicit), and finally the trustList which is the claimed membership in the trust list.

Next, the policy verifies that the document is indeed signed by the with respect the bearer's public key and the certificate with respect to the issuer's public key. Also we check that the trust membership claim is really eIDAS qualified. Only after this (so after all checks that can be done locally), we check the trust membership claim. This makes sence since the latter requires

interaction with the corresponding server which we can spare us if any of the other checks fails. For the simple case of a boolean trust list, we do not get (a lot of) attributes back, but we assume that we can at least verify the public key of the issuer with respect to the membership claim. This in fact abstracts the entire interaction with the DNS server and the checks that need to be performed on the response: we are simply requiring (however the mechanism in detail works) that the trust membership claim can be confirmed in the sense that certificate issuer has indeed the public key claimed in the certificate and is a member of the eIDAS qualified trust list.

7.2 Tuple-Based Trust Scheme without Translation

The second example in addition checks an attribute of the issuer, in the example that the method of identity proofing of the said certificate was *in person*; in fact this is merely an extension of the previous policy by the last line:

```
checkQualifiedSignatureInPerson(Document, Certificate, Signer) :-
  % Checking the certificate:
  extract(Certificate, format, eIDAS_qualified_certificate),
  extract(Certificate, issuer, Issuer),
  extract(Certificate, bearer, Signer),
  extract(Certificate, pubKey, PkSig),
  extract(Certificate, issuerKey, PkIss),
  extract(Certificate, trustList, TrustMemClaim),
  % check the document was indeed signed with PkSig:
  verify_signature(Document, PkSig),
  % check the certificate is indeed signed with PkIss:
  verify_signature(Certificate, PkIss),
  % check the claimed trustlist membership is eIDAS qualified:
  trustscheme(TrustMemClaim, eIDAS_qualified)
  % check the Signer is really on the that trustlist:
  lookup(TrustMemClaim, TrustListEntry),
  % check that the issuers key is indeed PkIss
  extract(TrustListEntry, pubKey, PkIss),
  extract(TrustListEntry, identityProofing, inPerson).
```

7.3 Trust Translation Scheme Scenario

As a third example we look at a Boolean Trust Translation Scheme Scenario. 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; the difference is that we are using a variant of the trustscheme predicate, trustschemeX which we define afterwards.

```
checkQualSigX(Document, Certificate, Signer) :-
  % Checking the certificate:
```

```
  extract(Certificate,format,eIDAS_qualified_certificate),
  extract(Certificate,issuer,Issuer),
  extract(Certificate,bearer,Signer),
  extract(Certificate,pubKey,PkSig),
  extract(Certificate,issuerKey,PkIss),
  extract(Certificate,trustList,TrustMemClaim),
  % extract (potentially foreign) trust scheme:
  extract(Certificate,trust_scheme,TrustScheme),
  % check the document was indeed signed with PkSig:
  verify_signature(Document, PkSig),
  % check the certificate is indeed signed with PkIss:
  verify_signature(Certificate, PkIss),
  % check the claimed trustlist membership is eIDAS qualified
  % or an equivalent one (hence the ...X):
  trustschemeX(TrustMemClaim,eIDAS_qualified)
  % check the Signer is really on the that trustlist:
  lookup(TrustMemClaim,TrustListEntry),
  % Check that the issuers key is indeed PkIss
  extract(TrustListEntry, pubKey, PkIss).
```

The function of the trustschemeX predicate is to check that a trustlist membership claim is either directly to the trustlist we are looking for (here eIDAS_qualified), or it belongs to a trustscheme that can be translated to eIDAS_qualified:

```
trustschemeX(Claim,DesiredScheme)
  % case 1: it is direclty the desired scheme:
  :- trustscheme(Claim,DesiredScheme).
trustschemeX(Claim,DesiredScheme)
  % case 2: we can translate it to the desired scheme:
  :- encodeX(Claim,DesiredScheme,URL),
     lookup(URL,Entry),
     extract(Entry,translation,"equivalent").
```

Here the encodeX, given a claim for a foreign scheme and the name of the desired scheme, generates a URL for the trust translation scheme. For instance, suppose the Claim is a (hypothetical) Swiss scheme at URL "PX2NO4LV.admin.ch" and the DesiredScheme is eIDAS_qualified, then the URL shall be "admin__ch.Translation.signature.trust.eu" (i.e. escaping the base URL of the original scheme, and selecting the corresponding Translation scheme of eIDAS qualified). This URL then should refer to the entry for the Swiss scheme, if it exists, at eIDAS and we can check that the translation yields the result "equivalent". (This is a provision for more complex translation schemes that requires further attributes to fulfilled.)

Note that we could also make chains of translations, for instance, if there is no direct translation from a Singapore scheme to eIDAS, but translatable via a Swiss trust scheme, we can formulate in TPL that such chains of translations should also be allowed. However, we cannot expect the verifier to find an appropriate set of translation hops automatically, so this would have to be

provided as follows:

```
trustschemeChain(Claim,[],DesiredScheme) :-
  trustscheme(Claim,DesiredScheme).
trustschemeChain(Claim,[Hop|Hops],DesiredScheme) :-
  encodeX(Claim,DesiredScheme,URL),
  lookup(URL,Entry),
  extract(Entry,translation,"equivalent"),
  trustschemeChain(URL,Hops,DesiredScheme).
```

Here we make use of the Prolog list notation where [] is the empty list and [X|Xs] represents a list that has at least one element X and a (possibly empty) rest list Xs.

7.4 Trust Scheme with Delegation Scenario

Let us now consider a simple delegation scenario: we have a Document that contains a purchase, signed by the proxy of a company. To this end, the company has issued a Mandate containing at least the following information:

- A reference for the Mandator, in this case we assume this is a pointed to a trust membership claim of eIDAS; this may however be more indirect, e.g. a certificate of the Mandator that was issued by an authority that is on the eIDAS trust list; only for simplicity we assume here the mandator is itself on the trust list.
- The public key of the proxy, so the signature of the proxy on a transaction can be verified. Strictly speaking, one does not need the identity of the proxy here. Alternatively, the proxy could also prove its identity using an eIDAS trust scheme. Using a public key, however, gives pseudonymity for the proxy (while several transactions made with respect to the same mandate are of course linkable).
- The purpose, here purchase. This may be more fine grained, e.g., allowing purchases only up to a certain limit.
- The authoritative delegationProvider DP: the ATV is required to check that the DP indeed has a valid entry containing (amongst others) a hash of the delegation. This check is to ensure that the delegation has not been revoked by the mandator. Also this must be part of the signed delegation, so there is a specified server DP that has the authority over the validity of the mandate. The entry contains the entire mandate, but in encrypted form plus a hash of the mandate (so only the proxy of the mandate can read it, but ATV who has received the mandate can check it with this hash).

The necessary checks to be performed can then be described by the following TPL specification:

```
checkQualSigDeleg(Document,Mandate) :-
  % Check the Mandate fits the document:
  extract(Mandate,format,delegation),
```

```
  extract(Mandate,proxyKey,PkSig),
  verify_signature(Document,PkSig),
  extract(Mandate,purpose,purchase),
  % Check the mandator key (wrt Trust list):
  extract(Mandate,issuer,Mandator),
  trustscheme(Mandator,eIDAS_qualified)
  lookup(Mandator,TrustListEntry),
  extract(TrustListEntry, pubKey, PkIss),
  verify_signature(Mandate,PkIss),
  % Check the mandate is still valid:
  extract(Mandate,delegationProvider,DP),
  lookup(DP,DPEntry),
  extract(DP,fingerprint,HMandate),
  verify_hash(Mandate,HMandate).
```

7.5 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(Document,Certificate,TrustScheme,TrustListEntry) :-
  extract(Certificate,issuer,Issuer),
  extract(Certificate,bearer,Signer),
  extract(Certificate,pubKey,PkSig),
  extract(Certificate,issuerKey,PkIss),
  extract(Certificate,trustList,TrustMemClaim),
  trustscheme(TrustMemClaim,TrustScheme),
  lookup(TrustMemClaim,TrustListEntry),
  extract(TrustListEntry, pubKey, PkIss).
```

In contrast to the formalization of the previous section, this predicate does not itself check the Trustscheme or the attributes of the TrustListEntry, but we leave that to the policy. For instance the simplemost case was the Boolean trust scheme where the policy would simply be:

```
trustpolicy1(eIDAS_qualified,TrustListEntry).

scenario1(Document,Certificate) :-
  checkDocument(Document,Certificate,TrustScheme,Entry),
  trustpolicy1(TrustScheme,Entry).
```

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

```
trustpolicy2(eIDAS_qualified,Entry) :-
  extract(Entry,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(Scheme,Entry) :-
  trustschemeX(Scheme,eIDAS_qualified).
```

7.6 Health Insurance Scenario

This scenario models that a patient needs to prove to a doctor to have health insurance using an electronic certificate of format healthcare_cert_format that contains, amongst others, the attributes issuer (the health insurance provider), country, and trustList (name of the entry in the trust list at "healthcare.trust.eu"). In this scenario the respective trust list entry has fields issuer, country, and pub_key. We need to verify that the fields issuer and country agree with the health care certificate and that the certificate is signed with this pub_key.

```
valid_health_insurance(Certificate) :-
  % check it is the right kind of format
  extract(Certificate, format, healthcare_cert_format),
  % look it up on the trustlist
  extract(Certificate, trustList, Claim),
  trustscheme(Claim, eu_healthcare_scheme),
  lookup(Claim, TrustListEntry),
  % compare information in certificate and trust list entry
  extract(TrustListEntry, format, healthcare_entry),
  extract(Certificate, issuer, InsuranceOrg),
  extract(TrustListEntry, issuer, InsuranceOrg),
  extract(Certificate, country, Country),
  extract(TrustListEntry, country, Country),
  % verify signature
  extract(TrustListEntry, pub_key, PK),
  verify_signature(Certificate, PK).
```

Recall that by the semantics of TPL the use of the same variable in several predicates (e.g. Country) ensures that the value must be the same (but can be arbitrary). Thus, in practice, it means that the first occurrence is binding (since the extraction can only yield one particular value) and the second occurrence implies a comparison.

7.7 TrumpU Case Study

Electronic admission to the PhD school of a university, say, DTU. One of the requirements is that the candidates holds an MSc Degree (in a suitable subject, but we may leave this subject-aspect out for simplicity). The point of this scenario is that the applicant could prove this electronically, holding an electronically signed document from his or her alma mater, let us say university U.

First let us only consider the problem to check that the electronic diploma of the student is indeed from the claimed university. In Europe we can easily require that it is an eIDAS qualified signature, and like in the above scenarios we may allow for trust translation. Then we have an instance of the above document checking scenarios where the document is the student's diploma, and the certificate is the university's certificate.

So far however this only assures us that the issuer of the student's diploma is indeed the organisation who is the owner of the university certificate. The problem is that any organization could self-apply the title "university"—hence the title of the case study. The idea is that one can build trust lists, for instance on a national government level, which institutions are indeed recognized as universities, probably based on adhering certain academic standards. The European union can then choose to recognize all such trust lists from its memberstates and from some non-EU countries based on bilateral agreements. This recognition can again be formalized as a trust translation scheme:

```
realUniversity(Diploma, UniCertificate) :-
  checkDocument(Diploma, UniCertificate, TrustScheme, Entry),
  trustschemeX(TrustScheme, eu_recognized_university),
```

Note that this does not check the contents of the Diploma, which may be in a local format and one has to still extract what kind of diploma it is.

8 Designing Translation Policies

The task of LIGHT^{est} is concerned with executing trust policies, trust translation policies, and delegation policies. The task of LIGHT^{est} is *not* the design of such policies; this is often a results out of political decisions (e.g. bilateral agreements between countries), related to issues that cannot be entirely formalized (e.g. a registration process where a person needs to be physically present), or the individual decision of a company (e.g. does the company decide to trust entities of a particular trust scheme for orders up to a certain value). However, the language TPL gives us a possibility to reason about policies, and to formalize–also in parts–the relationships and concepts behind a credential (e.g. under which circumstances a certain level of assurance is satisfied in a given trust scheme). This allows for reasoning and supporting the reasoning by tools, often without a lot of additional work, or simply by formalizing the relevant aspects in TPL. In this section, we expand on this "added value" of TPL, even though this is about a task that is not really an objective of LIGHT^{est}.

We focus on the design of translation policies. Recall that the Automatic Trust Verifier (ATV) when it encounters a scheme that is not directly trusted, it 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.

8.1 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, given 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.

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

```
iso2eidas(iso_loa1,eidas_low).
iso2eidas(iso_loa2,edias_low).
iso2eidas(iso_loa3,edias_substantial).
iso2eidas(iso_loa4,edias_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, edias_law).
```

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.

8.2 Designing Translations: the Rationale Behind a Policy

An important point is now that the translation only depends on information that 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 LIGHTest for any trust scheme that is sufficiently formalized in TPL.

8.3 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.

Example 1. *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, $S \to A$, 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 two Boolean properties, namely in_person and passport to indicate whether those requirements are met.

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

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

This scheme issues a certificate with a High LoA if both properties are true, it issues a Low LoA if only in_person is true. If none of these requirements are met, s_A issues no certificate, indicated with $\perp$ (undefined).

8.4 Translation

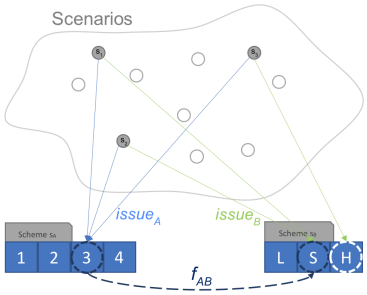

Figure 1: Illustration of a translation scenario.

This section discusses how to specify translation from a scheme s_A to another s_B. The translation is a function, $f_{AB} : A \to B$, 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 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 gives an example: a credential with the LoA 3 can be issued in the scheme s_A from three scenarios: s_1, s_2, and s_3. In the 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.

More generally, if the credential a translates to the credential b, then it does not necessarily hold that the preimages of a and b under $issue_A$ and $issue_B$ are equal, i.e., that $\{x \mid issue_A(x) = a\} = \{x \mid issue_B(x) = b\}$ does not hold. In other words, the set of entities x that qualify for credential a is not necessarily the same as the set of entities qualifying for b, simply because the issuing process may be based on slightly different criteria. Thus we cannot always ensure that the translation gives a truely equivalent credential; it is however recommended that the translation does *not* yield a *better* credential that the input to the translation. To that end, let us define a partial order on credentials that formulates what "better" actually means:

$$\begin{aligned} a >_{AB} b &\iff \{x \mid issue_A(x) = a\} \subsetneq \{x \mid issue_B(x) = b\} \\ a \geq_{AB} b &\iff \{x \mid issue_A(x) = a\} \subseteq \{x \mid issue_B(x) = b\} \end{aligned}$$

So $a > b$ says that a is strictly better the b, since the entities that satisfy a are a proper subset of those that satisfy b. The convention we recommend is thus:

Convention 1. *In general we recommend the following property of a translation f_{AB} from scheme A to scheme B:*

$$f_{AB}(a) = b \implies a \geq_{AB} b$$

According to this convention, a translation may downgrade a credential, but in general one wants to stay as close as possible to the original one:

Convention 2.

$$f_{AB}(a) = b \implies \forall b' >_{BB} b . a \not\geq_{AB} b'$$

This says, if a translates to b than there should not be any better level b' that would satisfiy the convention: if that were the case then one should translate: $f_{AB}(a) = b'$.

In general, for a given a, there can be several values b and b' that satisfy the two conventions above, but then b and b' are incomparable. In purely ordinal schemes, however, there is always a uniquely defined maximum element.

Example 2. *As an example, consider the example of classes of drivers licenses: there may be a class for normal cars, one for motor cycles and one for trucks. The one for motor cycles entails the one for normal cars, and so does the one for trucks; however neither trucks subsume motor cycles nor vice-versa. Suppose now that we translate from another scheme for drivers licenses, where e.g. a military driver's license includes the ability to drive all kinds of vehicles, so both the translation to a motor cycle license and the translation to a truck license would be satisfying our conventions. Unless the target scheme has an equivalent for both motor cycle and truck, there is no best translation in general.*

Let us finally also consider an example that shows why in a negotiation, one may actually choose to not satisfy the conventions above:

Example 3. *Consider the space* $S = \mathbb{B} \times \mathbb{B} \times \mathbb{N}$*, representing the Boolean criteria* in_person*,* passport *and (for simplicity a natural number) the* age*.*

The schemes s_A *and* s_B *are different in the condition for issuing the highest level credential:*

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

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

Following our conventions, there is no possible translation so that anyone gets an 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.*

It may thus be a decision of a policy designer whether they want to break the convention here. Given that the age difference is only one year, one may decide to "turn a blind eye" to the small discrepancy and make a translation e.g. $f_{AB}(\text{LoA } 2) = \text{LA}$. However, one should be clear that this could have legal implications: suppose in the scheme B the legal drinking age is 21 and the LA credential is considered a reliable proof of that, then inserting this translation may break this trust relationship, e.g. so that LA credentials cannot be used after all for legally proving to be over 21.

9 Graphical TPL

We now finally a new idea to graphically represent a large fragment of TPL policies. Note that this is language is aimed at the "expert-mode users" who would like to work on the TPL level, but would also like a shorthand at writing it. For the end users, we there are the graphical interfaces developed in WP 6 with extensive user testing.

We like to introduce here the *Graphical TPL*, or *GTPL* for short, using an example of an online platform for auction houses. We are talking here not about peer-to-peer auction houses like eBay that only operate only online, but the platforms that have arised to connect classical auction houses to the digital world without the need for every auction house to develop their own webpage and interface to the auctions. The auctions in questions may easily range up to thousands of Euros for a single item, which gives of course the classical problem of ensuring that the successful bidder indeed pays the sum they have bid. On the one hand, the auction house wants no entrance barrier for new customers who just "stumbled" upon an item by an Internet search; but on the other hand they want to avoid that, for instance, somebody practically anonymously bids on an item just to get the price up and then not paying if they have the highest bid.

This is of course a classical *trust* problem. The classical solutions are that one has to bring references from other auction houses or a bank statement, or be present at the auction in person, proving one's identity before the auction starts. This example shows how to transfer these aspect to the digital world using LIGHT^{est} so one can benefit from the large potential of the digital world without losing the security and trust guarantees of the classical auction houses.

9.1 Bidding Forms

Classically, auction houses allow customers to bid via standard mail, if they cannot be physically present as the auction. Bidders would for this purpose tell the auction house (per mail) a maximum bid for a particular item, and the auction house would accordingly act as if the bidder was present at the auction and bid up to the maximum bid. (After the auction, the succesful bidders are notified and with an invoice; only after payment, the auction house sends the items.)

For this purpose, each auction house would have their own bidding form. This is classically a simple paper form containing the personal information of the bidder and a list of items (the lot numbers and the maximum bid) – and of course a field where the bidder must sign the form. This would be sent to the auction house by mail.

Of course, the first step of digitalization was to have online cataloges, where one can click on items one wants to bid on, and enter a maximum amount. This would basically lead to an electronic version of the classical paper form and it would finally be transfered via https or simply email to the auction house. Such an electronic bidding form could look as follows (for simplicity we consider bidding only on a single item):

```
<AUCTIONHOUSEFORMAT
  auctionID="AUCTION18">
```

```
<bidder>...</bidder>
<address>
  <street>...</street>
  <city>...</city>
  <country>...</country>
</address>
<bid lotnr="..."
     amount="...">
<signature>
jmj7l5rSw0yVbvlWAYkKYBwk
</signature>
</AUCTIONHOUSEFORMAT>
```

We have here already included a field for a digital signature, which is actually still optional in many of todays online bidding solutions. If used, it would be a digital signature on a hash of the document. The first step to integrate such a form in LIGHT^{est} is of course to define the abstract syntax, i.e., describe it as a list of attribute-value pairs and describe the relationship to the concrete syntax (parser and pretty-printer). This is the small amount of work to connect a new custom format into LIGHT^{est}. From there on, we can assume to have that format in the library. We also may assume, since formats are essentially a list of attribute-value pairs, that we can easily graphically represent them as a table with two columns as such:

The Auction House Auction 2018	
Bidder Name	
Street	
City	
Country	
Lot Number	
Bid	
Signature	

The only fields that have a built-in meaning are the title fields (they identify the format name, and thus which fields are present) and the signature field: here we assume that this is really a digital signature algorithm. Also to each field we may attach a type, e.g. that the Lot number and the Bid must be integer numbers, and we may have range restrictions on them. However, whether a bid is for instance in Euro or Swiss Francs is not a concern of the abstract syntax itself, but of the semantics of the form (although the syntax may have indications like “Bid (Euro)”).

A First Policy Rule

The basic idea is now that we can use this simple graphical representation of the "empty" form as a basis for describing policies. For instance, let us define as a first example policy rule that the auction house wants to accept any bids up to 100 Euro (note again, that the currency is implicit in the format), even without any signature:

The Auction House Auction 2018	
Bidder Name	
Street	
City	
Country	
Lot Number	
Bid	≤ 100
Signature	

This is easily translated to a textual TPL policy as follows:

```
accept(FORM) :-
  extract(FORM, format, theAuctionHouse2018format),
  extract(FORM, bid, BID), BID<=100.
```

i.e., we just check that the form is of the correct format, thus ensuring there is a bid-field, and then extract that BID and check it is up to 100 Euro. Thus the basic idea is that policies can be formulated as a combination of constraints on the items of a form. Everything that is unconstrained remains an untouched (grey) field.

Checking Signatures

As a next example, let us add that the auction house accepts any bid up to 1500 Euro, if it is signed by an eIDAS qualified signature. To that end we use that the Signature field has a distinguished meaning, namely that the ATV can check the signature with respect to a given public key and the document. Of course, we do not wish to specify here any concrete public key, but rather that it is a key belongs to a particular trust scheme, here eIDAS qualified. We thus introduce here the notation [eIDAs qualified], i.e., the name of a trust scheme in square parenthesis to indicate that.

The Auction House Auction 2018	
Bidder Name	
Street	
City	
Country	
Lot Number	
Bid	≤ 1500
Signature	[eIDAS qualified]

The notation [eIDAS qualified] here is a short notation for a verification process that actually has quite a number of details. In particular, this implies that the signature comes with an eIDAS certificate (or has a pointer to where one can be obtained). Then the signature has to be verified against the public key of the certificate. The certificate itself is signed, and that signature has also to be checked, namely whether the issuer is indeed on the trust list and with the public key that verifies the certificate. Let us describe also this relationship in GTPL:

The Auction House Auction 2018	
Bidder Name	
Street	
City	
Country	
Lot Number	
Bid	≤ 1500
Signature	PK
Certificate	eIDAS certificate: issuer; bearer; pubKey = PK; trustList = [eIDAS_qualified]; Signature

Note that we have here depicted the Certificate field as an additional field of the Auction house form; this may of course organized in a different way, e.g. as a link to a URI where the information can be obtained. Note that we have here put the Certificate field below the signature which is

The Auction House Auction 2018	
Bidder Name	
Street	
City	
Country	∈ EU_EFTA_list
Lot Number	
Bid	≤ 1500
Signature	≈[eIDAS qualified]

a convention of GTPL to indicate that the signature does not span the Certificate (one may of course do that, but just to illustrate the convention).

With all the details clarified, we can translate this policy into TPL as follows:

```
accept(FORM) :-
  extract(FORM, format, theAuctionHouse2018format),
  extract(FORM, bid, BID), BID<=1500,
  extract(FORM, signature, Signature),
  extract(FORM, certificate, Certificate),
  extract(Certificate, format, eIDAS_qualified_certificate),
  extract(Certificate, pubKey, PK),
  verify_signature(Signature, PK),
  extract(Certificate, issuerKey, PkIss),
  verify_signature(CertSignature, PkIss),
  extract(Certificate, trustList, TrustMemClaim),
  extract(Certificate, signature, CertSignature)
  trustscheme(TrustMemClaim, eIDAS_qualified)
  lookup(TrustMemClaim, TrustListEntry),
  % check that PkIss is indeed in the TrustListEntry!
  extract(TrustListEntry, pubKey, PkIss).
```

Allowing Trust Translation

As a next step, we want to additionally allow that the we accept also those signatures outside eIDAS, if they are deemed equivalent via a translation scheme of eIDAS. To that end, we introduce the notion of ≈ for a trust scheme, meaning equivalence modulo a translation (where we rely on the trust translation schemes provided by the authority of the target scheme): We have here introduced yet another notation: since country would be from a finite list of values, we users may define subsets (like `EU_EFTA_list`) and check membership. So this policy would

accept bids up to 1500 Euro from EU/EFTA-countries, as long they have an eIDAS qualified certificate, or equivalent.

```
accept(FORM) :-
  extract(FORM, format, theAuctionHouse2018format),
  extract(FORM, bid, BID), BID<=1500,
  extract(FORM, signature, Signature),
  extract(Signature, format, sigform),
  extract(Signature, certificate, Certificate),
  % The following is the standard check of the certificate
  % and its trust membership claim specialized for eIDAS
  extract(Certificate, format, eIDAS_qualified_certificate),
  extract(Certificate, issuer, Issuer),
  extract(Certificate, bearer, Signer),
  extract(Certificate, pubKey, PkSig),
  extract(Certificate, issuerKey, PkIss),
  extract(Certificate, trustList, TrustMemClaim),
  % extract (potentially foreign) trust scheme:
  extract(Certificate, trust_scheme, TrustScheme),
  % check the document was indeed signed with PkSig:
  verify_signature(FORM, PkSig),
  % check the certificate is indeed signed with PkIss:
  verify_signature(Certificate, PkIss),
  % check the claimed trustlist membership is eIDAS qualified or equivalent:
  trustschemeX(TrustMemClaim, eIDAS_qualified)
  % check the Signer is really on the that trustlist:
  lookup(TrustMemClaim, TrustListEntry),
  % check that the issuers key is indeed PkIss
  extract(TrustListEntry, pubKey, PkIss).
```

Similarly, we can have additionally the rule that we accept any bid up to 1000 EUR, from eIDAS and translated (i.e., when the country is arbitrary):

The Auction House Auction 2018	
Bidder Name	
Street	
City	
Country	
Lot Number	
Bid	$\leq$ 1000
Signature	$\approx$[eIDAS qualified]

Note that all the policies we formulate are put together by *disjunction*, i.e., a bid is accepted if any of the rules matches. The order of the rules only determines in which order they are checked, so it makes sense to put the most common cases first, and the more specific cases later. Also, we here first check the easy-to-check constraints first, e.g., constraints on the amount of the bid, while the more "expensive" checks that involve a server-lookup are put last.

Custom-built Trust Schemes

LIGHT^{est} does not only support a fixed number of pre-existing schemes like eIDAS, but also allows for defining new trust schemes. As an example, there may be a platform for auction houses that acts as a unifying service to auction houses, i.e., auction houses can register there and thus share a customer base, where customers do not have to register themselves or have certificates. Suppose this auction house platform, let us call it `platform.de`, has its own bidding form to allow users to place bids. This one has now has the particular auction house as a field, say auctionID of the form:

User Name	
User Level	
Street	
City	
Country	
AuctionID	`Auctionhouse18`
Lot Number	
Bid	
Signature	`platform.de`

Note that here we fill in a concrete value for the Signature field, namely the domain name `platform.de` as a means to say that the signature must be with a key from `platform.de`. It is thus not the users signing the bid, but the auction house platform, therefore signing on behalf of the user. This actually the status how it is done today in most such platforms. Given that users could build up a reputation and given that the auction house trusts the platform, the auction house may formulate their own trust policy with respect to such bids arriving from the platform, e.g. let us specify that we accept bids up to 5000 Euro if the user has level premium:

User Name	
User Level	PREMIUM
Street	
City	
Country	
AuctionID	Auctionhouse18
Lot Number	
Bid	$\leq$ 5000
Signature	platform.de

The TPL translation would simply be:

```
accept(FORM) :-
  extract(FORM, format, thePlatformFormat),
  extract(FORM, userlevel, premium),
  extract(FORM, auctionID, auctionhouse18),
  extract(FORM, bid, BID), BID<=5000,
  extract(FORM, signature, Signature),
  extract(Signature, format, sigform),
  verify(FORM, pk_platform_de).
```

Here, for simplicity, we have assumed that the auction house already has the corresponding public key of the platform (pk_platform_de), but this may again be obtained via certificates based, e.g., on eIDAS.

Now to make this more flexible in the future, and to avoid that users have to reveal all their bids to the platform, the platform can issue certificates for their users and thus establish a trust list:

The Auction House Auction 2018	
Bidder Name	
Street	
City	
Country	
Lot Number	
Bid	$\leq$ 5000
Signature	PK

Platform.de User Certificate	
User Name	
User Level	PREMIUM
Street	
City	
Country	
Publickey	PK
Signature	platform.de

Here we have the bidding sheet of the auction house and a certificate from the platform vouching for the user. The link between them is the variable PK, the public key of the user: the bidding sheet has to verify with respect to this public key, and the certificate has to be established with respect to this.

The translation into TPL is as follows:

```
accept(FORM) :-
  extract(FORM, format, theAuctionHouse2018format),
  extract(FORM, bid, BID), BID<=5000,
  extract(FORM, signature, Signature),
  extract(Signature, format, sigform),
  extract(Signature, certificate, Certificate),
  % The following is the standard check of the certificate
  % and its trust membership claim specialized for eIDAS
  extract(Certificate, format, thePlatformCertificate),
  extract(Certificate, userlevel, premium)
  extract(Certificate, issuer, Issuer),
  extract(Certificate, bearer, Signer),
  extract(Certificate, pubKey, PkSig),
  % check the document was indeed signed with PkSig:
  verify_signature(FORM, PkSig),
  % check the certificate is indeed signed with PkIss:
  verify_signature(Certificate, pk_platform_de),
```

Again, the public key here is fixed pk_platform_de, as we are talking about a fixed issuer.

Trust Scheme for References

As a final point in this example, we want to show how we a full-fledged tuple-based trust scheme can also be used. While so far the auction house could directly rely on the platform as a trusted party, we want to make the scenario where it is more indirect. In the classical auction landscape, a bidder who is known at one auction house may get a reference for use at another auction house where he or she is bidding for the first time. Classically, that makes only sense if the auction house that gives the reference is actually known and can be verified. This is again where trust schemes can help, since the auction house platform may establish a trust list of auction houses that are valid for giving letters of reference, so our auction house may choose to rely on them without even knowing them. The following policy expresses exactly that: the auction house accepts bids up to 5000 Euro when there is a certificate of reference by an auction house on the trust list `refs.platform.de`:

Auction House'18	
Bidder Name	
Street	
City	
Country	
Lot Number	
Bid	≤ 5000
References	User Reference: Issued by; User Name; Street; City; Country; Publickey: PK; Rating: PREMIUM; Signature: [refs.platform.de]
Signature	PK

Here, we use for the first time the fact that the attribute of a form may itself be form, namely the certificate. Note also that we use here a URL to refer to a trust scheme; such a technical aspect may be otherwise "hidden" by a constant e.g. references_platform_trustlist.

The translation to TPL would be as follows:

```
accept(FORM) :-
  extract(FORM, format, theAuctionHouse2018format),
  extract(FORM, bid, BID), BID<=5000,
  extract(FORM, signature, PkSig),
  % check the document was indeed signed with PkSig:
  verify_signature(FORM, PkSig),
  extract(Reference, format, platform_user_reference),
  extract(Reference, publickey, PkSig),
  extract(Reference, rating, premium),
  extract(Reference, signature, Signature),
  extract(Reference, issuerKey, PkIss),
  extract(Reference, trustList, TrustMemClaim),
  trustscheme(TrustMemClaim, refs_platform_de)
  verify_signature(Reference, PkIss),
```

```
lookup(TrustMemClaim, TrustListEntry),
extract(TrustListEntry, pubKey, PkIss).
```

GTPL Translation Semantics

We now give the semantics of GTPL by a translation function from GTPL to standard TPL (which of course already has a semantics). The GTPL input, although graphical, is actually also represented here in a textual form as follows: every form is represented as an expression F(attributes) where F is the name of the form (e.g. eIDAS_certificate) and attributes is a list of (attribute-value) pairs, where the first component of each pair is an attribute name like signature and the second component is one of the allowed expressions of GTPL, namely either BLANK (the gray fields), a concrete value, a variable, a comparison operator along with a concrete value or variable, or another form in the same syntax. In the special case of the trustlist attribute, we also allow the attribute to be [value] or ≈[value] where value is the name of a trust scheme (like eIDAS). The semantics starts with calling $[\![F(Attlist)]\!]^{Toplevel}$ for a form, and it will generate a TPL clause accept(Form) :− ... . that holds true if Form satisfies all the requirements specified in $F(Attlist)$. The full specification is given in Figure 2.

The GTPL translation procedure starts from the top level form, e.g., the online form of the auction house in the previous example. The first semantic function starts a new rule of the trust policy, it first checks the format of the form and then continues with a list of attributes and their corresponding values contained in the form.

If any attribute is unconstrained, i.e., an untouched BLANK field, then we continue with the rest of the attributes in the list until the attribute list becomes empty. If for an attribute a particular value or a variable is specified, then the translation is to use the extract predicate for this attribute and have the value or variable as a third argument, i.e., it will be checked that the element has that value or, if this is the first occurrence of this variable, then the variable is bound by this. If for an attribute an operator and term was specified, then the translation is to extract the attribute into a new variable and compare the variable to the term.

The attribute signature is a special case: in this case we verify that the form is indeed signed with the private key of the corresponding public key X. In fact, we assume here that the form itself provides in case of a signature enough information to determine what is the public key that corresponds to the signing key, the part of the text that is signed, and the signature itself. If, for instance, the public key is actually not part of the information conveyed in this form, then there is a complication that we omitted in the semantics here: in this case there must be another place (outside given the form) related to the policy where the public key is obtained; then one must delay the checking of the signature until reaching the step where this public key is extraced.

In case that the attribute value is itself a form (i.e. a subform), we extract the subform and continue with the subform's attribute list with the same translation procedure recursively.

Finally, trust_list is also a special attribute: the value of such an attribute shall use the notation to indicate the requirement that the certificate issuer must be on the given trustscheme , or using

$[\![F(Attlist)]\!]^{Toplevel} =$
accept(Form) :–
$[\![F(Attlist)]\!]^{Formlevel}_{Form}$.
where Form is a new variable.

$[\![F(Attlist)]\!]^{Formlevel}_{Form} =$
extract(Form,format,F)
$[\![F(Attlist)]\!]^{Attlevel}_{Form}$

$[\![(attname, BLANK) : Attlist]\!]^{Attlevel}_{Form} = [\![Attlist]\!]^{Attlevel}_{Form}$

$[\![(signature, X) : Attlist]\!]^{Attlevel}_{Form} =$
,verify_signature(Form,X)
$[\![Attlist]\!]^{Attlevel}_{Form}$

$[\![(attname, t) : Attlist]\!]^{Attlevel}_{Form} =$
,extract(Form,attname,t)
$[\![Attlist]\!]^{Attlevel}_{Form}$
where t is either a term or a value.

$[\![(attname, Comp\ t) : Attlist]\!]^{Attlevel}_{Form} =$
,extract(Form,attname,Valuevar),Valuevar Comp t
$[\![Attlist]\!]^{Attlevel}_{Form}$
where Valuevar is a new variable and t is either a term or a value.

$[\![(attname, R(Attlist_s)) : Attlist]\!]^{Attlevel}_{Form} =$
,extract(Form,attname,Subform),
$[\![R(Attlist_s)]\!]^{Formlevel}_{Subform}, [\![Attlist]\!]^{Attlevel}_{Form}$
where Subform is a new variable.
$[\![(trust\_list, [Value]) : Attlist]\!]^{Attlevel}_{Form} =$
,extract(Form,trust_list,TrustMemClaim)
,trustscheme(TrustMemClaim,Value)
,lookup(TrustMemClaim,TrustListEntry)
,extract(TrustListEntry,pubKey,PK)
,verify_signature(Form,PK)
$[\![Attlist]\!]^{Attlevel}_{Form}$
where TrustMemClaim, TrustListEntry, and PK are new variables.

$[\![(trust\_list, \approx [Value]) : Attlist]\!]^{Attlevel}_{Form} =$
,extract(Form,trust_list,TrustMemClaim)
,trustschemeX(TrustMemClaim,Value)
,lookup(TrustMemClaim,TrustListEntry)
,extract(TrustListEntry,pubKey,PK)
,verify_signature(Form,PK)
$[\![Attlist]\!]^{Attlevel}_{Form}$
where TrustMemClaim, TrustListEntry, and PK are new variables.

Figure 2: The translation function from GTPL to TPL.

the notation ≈[trustscheme] to indicate the more relaxed requirement that the signature is either on this trustscheme or on a foreign trust scheme that can be translated to trustscheme. In the first case, the attribute of trust_list is extracted, a URL, and we use the predicate trustcheme to verify that the URL indeed belongs to the desired trust scheme; then we look up such a URL to get a trust list entry, to extract the public key from and verify the certificate with. The second case with trust translation is almost identical, except for using the predicate trustchemeX instead to allow for translations.

10 Conclusions

We have introduced TPL as a way to describe trust policies, trust schemes, trust translation schemes, and trust delegation schemes. The main aim was to define a language that is conceptually simple and clear, while at the same time being powerful enough for our purposes. Indeed one may wonder if it is overkill for many simple business cases with rather straightforward policies. Note that there is much work on usability in LIGHT^{est}, giving the user simple graphical and textual languages that easily cover the most standard applications that are accessible to wide variety of people without computer science background. That these languages for the end users are all based on the more powerful TPL comes at little extra cost, rather, everything has a simple logical basis. A particular benefit of using Prolog-style Horn clauses is the direct "executability", i.e., it is without too much trouble to implement the machinery of LIGHT^{est} on this basis and prove implementations also to be correct. Last but not least, this helps accountability: one can easily prove that a decision was made correctly adhering to a given policy.

List of References

[1] Y. Gurevich and I. Neeman. DKAL: distributed-knowledge authorization language. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, USA, 23-25 June 2008*, pages 149–162, 2008.

[2] T. Hinrichs and M. Genesereth. Herbrand logic. Technical Report LG-2006-02, Stanford University, CA, USA, 2006. http://logic.stanford.edu/reports/LG-2006-02.pdf.

[3] B. K. Mejborn. ASN.2: A model-driven approach to secure protocol implementation, 2016. Bachelor Thesis DTU, 2016, available upon request.

[4] S. Mödersheim and G. Katsoris. A sound abstraction of the parsing problem. In *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*, pages 259–273, 2014.

[5] Moritz Y. Becker and Cédric Fournet and Andrew D. Gordon. SecPAL: Design and Semantics of a Decentralized Authorization Language, 2006. Microsoft Research Technical Report MSR–TR–2006–120.

[6] The LIGHT^{est} consortium. LIGHT^{est} Grant Agreement, Annex I Description of Work, 2016. Classified.

11 Project Description

LIGHT^{est} 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 LIGHT^{est} 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 LIGHT^{est} 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 LIGHT^{est} open source software to easily query this trust information to verify trust in simple signed documents or multi-faceted complex transactions.

The three-year LIGHT^{est} 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 LIGHT^{est} consortium consists of 14 partners from 9 European countries and is coordinated by Fraunhofer-Gesellschaft. To reach out beyond Europe, LIGHT^{est} 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 Denmark (DK) and Ubisecure (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.