The LIGHT^{est} Foundation

DTU Technical Report-2018-6 (updated version)

Sebastian Mödersheim, Anders Schlichtkrull and Andreas Viktor Hess *

February 17, 2020

Abstract

This report presents the Trust Policy Language (TPL), its semantics, and shows how it can be used to specify policies for trust, trust translation, and delegation. TPL and its semantics are highly inspired by Prolog. The semantics come in two flavors, an executable semantics and a logical semantics. Because of the executable semantics, a specification in TPL can also be executed as a program that checks if a decision lives up to a policy. TPL is the language used in LIGHT^{est}, which is a toolbox for digital policies for trust, trust translation, and delegation. TPL serves as its foundation and its semantics defines the meaning of a specified policy is. The semantics can therefore be used to settle any discussion of what the meaning of a specified policy is. In order to make TPL easy to use, we present GTPL which is a graphical language that allows end users to specify policies in TPL.

1 Introduction

This technical report represents the formal basis of the LIGHT^{est} project[1]. It does so by describing the LIGHT^{est} Trust Policy Language (TPL) language and how it can be used to define and describe policies for trust, trust translation, and delegation. While the syntax of a language defines which expressions it contains, the semantics of a language defines what they mean.

We strive for giving TPL a well-defined semantics where it is always clear exactly what the meaning of an expression is. This is important for two reasons. Firstly, we want to avoid that in some corner cases it is unclear what a particular policy means. With a clear semantics any disagreement on the meaning of a policy can be resolved by referring to the semantics. This is particularly important in the LIGHT^{est} project which brings together partners with different backgrounds from both academia and industry. Discovering and resolving such mismatches early in the project is invaluable. Secondly, we want to mechanize policies, i.e., have computer programs called automated trust verifiers (ATVs) determine whether a given decision satisfies a given policy or not. We therefore need a way to turn trust policies into algorithms that the ATVs can follow. With a precise semantics of the policy language it is possible to prove that an algorithm correctly implements a policy, or even better, automatically derive the algorithm from the policy. In fact, inspired from this latter point we have decided to let TPL be a logic programming language based on the Prolog programming language, since this allows for declarative policy specifications that are directly executable. In other words, TPL can be seen as both a specification language and as a programming language.

*We thank Bihang Ni, Rasmus Birkedal, Georg Wagner, Stefan More and Lukas Alber for contributions.

[1] See `https://www.lightest.eu` and `https://www.lightest-community.org`.

We would also like for TPL to be usable by people without a programming background and we therefore introduce a simple graphical language for end-users that can handle most of the use cases of LIGHT^{est} and that can be easily mapped into TPL.

This technical report uses text adapted from deliverables D2.5 and D2.6 of the LIGHT^{est} project and our papers on TPL [12] [14].

2 TPL Design Decisions

The LIGHT^{est} Trust Policy Language, or TPL for short, is a language for defining policies for trust, trust translation, and delegation. TPL allows us to integrate these different parts of the LIGHT^{est} project in a precise and uniform way and its semantics can serve as an unambiguous reference for testing and formally proving the correctness of any implementations based on TPL such as an ATV.

TPL can be seen as a subset of Prolog that abandons some of the more advanced features in favor of achieving a simpler semantics. Like in Prolog, a program is a list of definite Horn clauses and TPL shares most of the syntax of Prolog. This is inspired by similar languages for access control based on Horn clauses such as SecPAL and DKAL [1, 5].

The particular benefits from using Horn clauses are the following:

- Lists of Horn clauses have a simple *logical* semantics, namely the semantics of first-order logic.
- Horn clauses famously have an *executable* semantics. With this semantics, a list of Horn clauses is a program. This means that a TPL policy immediately gives rise to an algorithm for evaluating whether a decision lives up to the policy.
- More generally, the semantics of TPL allows us to make queries of policies that go beyond evaluating whether a specific decision lives up to a given policy. We could, e.g. for a translation policy between levels of assurance from respectively an ISO standard and eIDAS, query which levels of assurance in the ISO standard correspond to eIDAS level "low". This technique can e.g. be used to discover ambiguities in policies.
- Prolog interpreters could be used as reference implementations of ATVs, for testing, or as the basis of implementations of ATVs.
- We can draw on previous experiences with and research on logic programming.
- Many policies are in practice just simple enumerations of cases. These are trivial to express as Horn clauses.
- On the other extreme, with the executable semantics, Horn clauses are Turing complete. Therefore every algorithm can be written using Horn clauses and executed following the executable semantics. This means that the language will never limit us by lack of expressive power.
- Lists of Horn clauses are suitable for expressing many concepts that regularly arise in policies for trust. We give two examples: Firstly, one can easily describe the logical relationships between several criteria, for instance, that when an order is above a certain value then stricter criteria have to be fulfilled. Secondly, Horn clauses are suitable for describing delegation, for instance where documents are signed by the proxy of a company.

- It is possible to extend the *executable* semantics with a number of predefined predicates that trigger the necessary queries to servers, e.g. using DNSSEC, and process their answer. When we make such extensions, we must consider what this means for the logical semantics. Here we can again rely on previous experiences with and research on logic programming.
- For the average users we can provide design patterns for their policy.
- For the average users we can provide an interface to a graphical language which is more intuitive to use but has limited expressive power.
- A TPL interpreter can, for an execution of a policy that results in a transaction being accepted, produce a transcript containing the policy, the query and the record of the interactions with the external environment. Using this transcript, an independent and verified tool, called RP_x, can check that indeed the decision of accepting the transaction was a logical consequence of the policy, query and interactions with the external environment. The reason this is possible is that, like RP_x, TPL is based on first-order logic – specifically a clausal logic.

TPL allows for complex forms of reasoning about trust, for instance trust translation and delegation. It is, however in the hands of each business to set the policy of what they are willing to accept and we expect that in the vast majority of cases simple policies will be sufficient. For both complex and simple policies, it is important that we provide a specification language that is both clear and unambiguous, and where the policies can be evaluated by an ATV. In this technical report, we argue that TPL has these properties.

3 A Gentle Introduction to TPL

We illustrate the flavor of TPL and what specifications could look like with a few examples. The language is based on definite Horn clauses, i.e. formulae of the form

conclusion :– requirement1, ..., requirementN.

One may simply read `:-` as the word "if" and the commas as "and". The logical meaning of a definite Horn clause is that 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 tell us whether the conclusion holds or not.

Consider as a specific example the Horn clause

trust(X) :– delegate_of(X, Y), trust(Y).

This can be interpreted as "I trust X if X is a delegate of an entity Y that I trust." Here, we have introduced as new vocabulary the *predicates* trust and delegate_of that have no built-in meaning in TPL; in fact, they obtain their meaning from the clauses we specify. X and Y are *variables* which are a subclass of the more general notion of *terms*. A term is either a variable (such as X), a constant (such as a, b or c) or a function applied to other terms (such as $f(X, Y)$ or $g(g(X), f(Y)))$). A variable such as X and Y can be replaced by any term and the clause applies. For instance, suppose that trust(a) and delegate_of(b, a) already hold, then we can derive trust(b) using the clause above, because we can *instantiate* X to b and Y to 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_of(b, a).

Then we can derive trust(b).

Suppose we also add delegate_of(c, b) and delegate_of(d, c). Then we can also derive trust(c) and trust(d), 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 us redefine trust(X) to denote direct trust only (say, trust(a) is the only clause specified for the trust predicate) and introduce a new predicate trustD(X, N) for trust through delegation; here N denotes the length of the chain of delegation, e.g., trust(X) is equivalent to trustD(X, 0). The length of a chain of delegations is called a delegation level. Then we can specify trust by delegation as the following Horn clauses:

trustD(X, 0) :− trust(X).
trustD(X, N) :− N > 0, delegate_of(X, Y), trustD(Y, N − 1).

Thus we can now derive in this example:

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

Be aware that in contrast to many Prolog interpreters, the current version of TPL considers N − 1 to be the result of subtracting 1 from N. We elaborate on this in subsection 4.2.

We can now easily express a policy for accepting an electronic purchase based on both the delegation level and the amount of a purchase, specifically the policy that below 100 Euro any delegation level is fine, below 1000 Euro the delegation level should be at most 1, and up to 1 Mio. Euro we do not accept delegation. This is specified as follows:

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).

We use the less-than-or-equals predicate (<=). 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[2] We shall use this now as a glimpse into the typical working of an interpreter.

The interpreter is fed with a rule base, a list of definite Horn clauses 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 whose 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.

[2]The order of the rules determines the order in which solutions are found and, in the worst case, a particular rule order can cause in interpreter to run into an infinite loop (e.g. when rules are recursive and the check for termination is not the first rule).

```
TPLPolicy   ::=  Clause*

Query       ::=  (Predication,)* Predication.

Clause      ::=  Predication.
             |   Predication :- (Predication,)* Predication.

Predication ::=  PredicateSymbol
             |   PredicateSymbol((Term,)* Term)

Term        ::=  VariableSymbol
             |   ConstantSymbol
             |   FunctionSymbol((Term,)* Term).
```

Figure 1: Syntax of TPL specified by a grammar.

We are also able to formulate queries like trustD(X, 2) which corresponds to the question "Who do we trust on delegation level 2?" We obtain two answers, namely X = c and X = d. This way we could use an interpreter also to reason about policies and to test policies, e.g., we can test that the policy indeed reflects what we wanted it to express.

4 A Detailed Definition of TPL

TPL consists of three parts:

- The core language.
- The language extensions.
- The standard library.

In this section we will describe these three parts.

4.1 The Core Language of TPL

The core language of TPL mainly consists of *definite Horn clauses*. Its syntax is based on that of first-order logic and Prolog.

We define four disjoint sets of symbols: (1) Variable symbols – starting with upper-case letters. (2) Function symbols – starting with lower-case letters and having fixed arity. (3) Constant symbols – starting with lower-case letters. (4) Predicate symbols – starting with lower-case letters and having fixed arity.

With this in place, we use a grammar to define the syntax of TPL specifications, as shown in Figure 8.

4.1.1 Semantics

We here briefly sketch two ways to formally define the semantics of TPL.

Logical Semantics A 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), r(Y, X) becomes ∀X, Y. p(X, Y) ← q(X) ∧ r(Y, X).

Special care must be taken for built-in predicates, i.e. the interface to the environment. For example we have an extract predicate that is the interface to the concrete formats and their parsers, as well as lookup that is the interface for looking up information on a server. For the semantics, we fix the meaning of these built-in predicates to an (arbitrary) snapshot of the world; in particular, we assume that during the checking of the policy, the state of the world does not change.[3] One may also evaluate logically a historical policy decision by specifying the environment as it was at some point in the past in order to answer the question of whether a given document was within the policy at a previous point in time.

More formally, 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 it holds that $H \models \sigma(q_1) \wedge \ldots \wedge \sigma(q_n)$ where $\models$ is the semantics of first-order logic as defined in any standard textbook. A policy might use built-in predicates which go beyond logical reasoning such as lookup which performs a call to a server. To define the semantics of the solutions in this case we, for a specific policy, allow the inclusion of a formula f that partially specifies this external environment. Such an f could be a trace of the interactions that happened in an execution of the policy. This f simply consists of a number of clauses. As such the semantics is defined as $H \wedge f \models \sigma(q_1) \wedge \ldots \wedge \sigma(q_n)$.

Executable Semantics TPL is similar to Prolog, but does not include the ! operator or negation as failure. Such "counter-logical" elements would forbid interpretation as logical formulas and the resulting clear and simple semantics. Policies are lists of definite Horn clauses and TPL also shares most of Prolog's syntax.

TPL's executable semantics is the same as that of Prolog except that in TPL our unification always includes the occurs check.[4] The semantics of Prolog can be described as an interpreter – see e.g. Deransart, Ed-Dbali and Cervoni's textbook [3], in particular, in Section 4.2. TPL's built-in predicates (such as extract, lookup, <=) are not part of TPL's core language but are defined outside it. We will get back to these predicates in Subsection 4.3.

4.2 The Language Extensions of TPL

We have considered a number of features—inspired by Prolog—that extend the core language of TPL. The included extensions are list syntax and arithmetic.

List Syntax TPL has, like Prolog, special syntax for lists:

[3] With respect to the assumption that the world does not change during policy evaluation, consider the following example. A policy could ask that a transaction is only accepted if approved by officials in two distinct sections, A and B, of a company, where the policy designer (unspokenly) relied on the fact that by company policy, no employee works in both sections. Then it is conceivable that an employee approved the transaction, who happens to *move* from section A to section B – with the corresponding trust list entries being updated just while some transaction is checked against the policy. It could thus happen that the policy is "accidentally" fulfilled by the single employee's approval, even though the trust list never actually showed any employee as members of two sections at the same time. Indeed if such "race conditions" are relevant, this must be solved by a kind of locking of databases for the duration of the policy checking. We believe this kind of scenario is extremely atypical for trust policies and not practically relevant.

[4] The usual unification algorithm will only unify a variable x with a term t if x is not a proper subterm of t (i.e., x occurs in t and they are not equal). For instance, X is a proper subterm of $f(X)$ and so they cannot be unified. This is called the occurs check. Prolog implementations usually omit it for performance reasons, but this leads to unsound unification.

- [] denotes the empty list.
- [X|Xs] denotes the list which has X as the first element and the elements in Xs as the rest of the list in the same order as in Xs.

To add this to the current TPL interpreter implementation would be a very conservative extension since without it, [] can simply be represented by a ground term empty and [X|Xs] using a composite term like cons(X, Xs).

Arithmetic TPL contains built-in arithmetic. The operators allowed are the following:

Symbol	Meaning
+	Addition
−	Subtraction
∗	Multiplication
/	Division
<	Less than
>	Greater than
<=	Less than or equals
>=	Greater than or equals
==	Integer equality

The operators are defined for integers and could be extended for floating point numbers. Prolog interpreters typically impose restrictions requiring that these operators are only ever applied to ground (variable free) terms or terms that are fully instantiated to ground (variable free) terms in the current execution. TPL imposes the same restrictions.

Prolog interpreters typically consider +, −, ∗, and / to be function symbols on line with any other function symbols; the idea is that e.g. (N+1)−(M∗2) is a term in the same way as s(a(N,1),m(M,2))—the former just happens to represent an arithmetic expression. A term containing these symbols can then be evaluated using a predicate called is. In TPL this works differently. Firstly, the subterms of an arithmetic expression cannot itself be an arithmetic expression. Secondly, TPL has no is predicate – instead before the TPL interpreter looks for a clause whose head matches the first predication of the current goal it will evaluate all arithmetic expressions in the predication. If the expressions in the predication contains variables that have not been substituted with integers then an exception is thrown.

4.3 Standard Library

Besides the interpreter core, the TPL contains built-in predicates like extract whose truth value depends on extra-logical facts and actions. These predicates are implemented as external functions that are invoked by the native Java code. For this, it is necessary to partition the parameters of built-in predicates into *inputs* and *outputs*; e.g. for extract, the first two arguments are inputs, and the resulting value is the output. It is required that all the input arguments must be ground terms (containing no variables) when the interpreter reaches them. After finishing an external call like a server lookup, the control is given back to the interpreter.

This section describes built-in predicates of TPL in more detail, as they are currently found in our reference implementation ATV.

Built-in Predicate 1 (extract)**.** *The extract predicate is used to extract information from a document (e.g. a transaction, certificate, or trust list entry). This predicate gives a uniform interface to all kinds of data formats; the interpreter is designed modular so that new data formats can easily be integrated by providing a parser for the respective data structure. For a call*

extract(From, What, Out)

we have that Form is an input document, What is a field of the document, and Out is the output, i.e., the value of that field.

The set of fields that are available depends on the format. We elaborate on our use of formats in a later section of this technical report. Thus, when trying to extract a field that does not exist in the present format, the predicate fails. For every format at least one field is defined, namely format which returns the unique identifier for the document's format.

Central to LIGHT^{est} is the notion of trust lists. A trust list is a list of trusted authorities who may issue certificates. An example is the lists published by the European Union in the eIDAS framework. We introduce predicates to do lookups in general and in particular to trust lists.

Built-in Predicate 2 (lookup and trustlist)**.** *The lookup predicate allows to perform lookups at DNS name servers and HTTP queries authenticated using DANE. The input parameter Domain defines the DNS domain to query, while the output parameter Entry contains the desired document. In a similar manner, the* trustlist *predicate is a more specific case, which is used to retrieve a single entry, identified by the parameter Certificate, from a trust list.*

lookup(Domain, Entry)
trustlist(Domain, Certificate, TrustListEntry)

Built-in Predicate 3 (trustscheme)**.** *The trustscheme predicate checks if a trust scheme claim (a domain name) indeed represents a given trusted scheme. Both parameters are input parameters. For instance, a call*

trustscheme(TrustSchemeClaim, eIDAS_qualified)

is true if and only if the trust scheme claim is a claim for an eIDAS membership.

Built-in Predicate 4 (verify_signature)**.** *The verify_signature predicate has two input parameters. For a call*

verify_signature(Form, PubK)

the TPL interpreter will use the appropriate signature verification function for the format of Form and succeeds if and only if the signature of the form can be verified using the public key PubK.

Built-in Predicate 5 (verify_hash)**.** *The verify_hash predicate checks if an object evaluates to the correct hash value. So for a call*

verify_hash(Form, Hash)

the TPL interpreter will use the appropriate hash function for the format of Form and succeed if and only if the parameter Form has the same hash as passed by the parameter Hash.

4.3.1 Other Predicates

In this technical report we will also show a number of general purpose predicates that can be considered part of the standard library but which are defined in TPL rather than come as built-in predicates implemented in Java.

5 Formats

Policies work on forms represented by a variety of concrete data formats, from X.509 certificates, DNS resource records, and ASIC contains to custom data formats for electronic forms. TPL supports all of these in a flexible way without cluttering the policies with low-level details like parsing. We consider an abstract notion of *formats*, similar to abstract syntax, namely like a *paper form* with fields to fill in and each field having a unique identifier. This abstracts from concrete measures (like XML) to structure this information, and any concrete format can be connected to TPL by providing a parser and pretty-printer for it, i.e. the transformation between actual byte strings and abstract syntax. Let us consider a form for the auction house example. Abstractly, it is a set of attribute-value pairs:

```
{(format, the_auction_house_2019), (bidder_name, "John Doe"),
 (street, "Dartmouth St"), (city, "Midfarthington"),
 (country, "England"), (lot_number, 54678), (bid, 60),
 (signature, ...), (certificate, ...)}
```

The actual transaction on the string level could be an XML representation:

```
<?xml version="1.0" encoding="UTF-8"?>
<format name="the_auction_house_2019">
  <person>
    <name>John Doe</name>
    <street>Dartmouth St.</street>
    <city>Midfarthington</city>
    <country>England</country>
  </person>
  <lot_number>54678</lot_number>
  <bid>60</bid>
  <signature> ... </signature>
  <certificate> ... </certificate>
</format>
```

The idea is that abstract symbols like `bidder_name` should be a sound abstraction of their concrete byte-level format [11]. Notice that the XML representation's tree structure and the attribute value pair set representation are not the same: it is often nice to have a layer on top of an XML format, so one does not have to browse the XML parse tree but has an immediate representation of the data suitable for one's purposes. TPL provides a built-in predicate extract connecting the interpreter with the appropriate parser so that attributes can be extracted from the format as specified by the attribute value pair representation.

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

- XML-based formats.
- ASN-style format descriptions.

Both of these are prototype implementations that support formats with a fixed number of elements [10].

5.1 Interfacing to TPL

As said, we see a format as consisting of a number of attribute-value pairs 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.

We now show how to work with such certificates in TPL without bothering about the concrete syntax of a format by using the extract predicate (that is linked to a parser implementation), 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.

Any format comes with a special attribute format that stores which kind of format a particular text is. For instance in the above we may additionally require on the right-hand side 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.[5]

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

6 Lookups

As said, the standard library includes predicates lookup and trustlist which do lookups at DNS name servers. 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 trust list membership claim from a certificate and use the lookup function to just check the claim, for instance:

[5] 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.

```
myfirstpolicy(Transaction) :-
  extract(Transaction, certificate, Certificate),
  extract(Certificate, issuer, IssuerCertificate),
  extract(IssuerCertificate, trustScheme, TrustMemClaim),
  trustlist(TrustMemClaim,IssuerCertificate,TrustListEntry),
  % followed by some checks on the TrustListEntry.
```

Even though the expectation of the modeler 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 issuer certificate and queries that server, even if the URL is, say, trustme.attackerspace.tk. While it may still be obvious in such a small policy, such an omission can easily go annoticed in a more complex policy. An integrated development environment for TPL could to warn the user about such a specification.

For easily checking trust schemes, we introduced in the standard library the 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 Further Examples

We now illustrate more advanced uses of TPL with the specification of a number of scenarios inspired by the LIGHT^{est} architecture deliverable.

7.1 Boolean Trust Scheme without Translation

In the first scenario we consider an organization that receives documents and wants to check whether they are signed with eIDAS qualified signatures. Each document is therefore required to be signed and to come with a certificate proving that the signer is allowed to make eIDAS qualified signatures. A signer is allowed to make eIDAS qualified signatures if an issuer who is on the eIDAS trust list has given him this permission. Therefore, the certificate must be signed by such an issuer.

In this example the check made with the trust list is Boolean—either the issuer is on the trust list or she is not. Each entry of the trust list therefore needs to at least contain a record of the public key of the issuer. In eIDAS, the trust list entry is actually the entire certificate.

The organization can use the following TPL policy to make the checks:

```
accept(Form) :-
  extract(Form, format, document_format),
  extract(Form, certificate, Certificate),
  % Checking the certificate:
  extract(Certificate, format, eIDAS_qualified_certificate),
  extract(Certificate, pubKey, PkSig),
  extract(Certificate, issuer, IssuerCertificate),
  extract(IssuerCertificate, trustScheme, TrustMemClaim),
  % check the document was indeed signed with PkSig:
  verify_signature(Document, PkSig),
  % check the claimed trustlist membership is eIDAS qualified:
```

```
  trustscheme(TrustMemClaim, eIDAS_qualified),
  % lookup the entry on the trustlist:
  trustlist(TrustMemClaim, IssuerCertificate, TrustListEntry),
  % extract the issuer's key
  extract(TrustListEntry, pubKey, PkIss),
  % check the certificate is indeed signed with PkIss:
  verify_signature(Certificate, PkIss).
```

The verification of the signature with respect to a given public key we represent by the predicate verify_signature.The policy first requires that the format of the document is document_format and that the format of the certificate is indeed a certificate for an eIDAS qualified signature and we extract the following information from it: the pubKey (the public key of the bearer), and finally the issuer which is the issuer certificate. From the issuer certificate we can extract a URL at which there should be an entry representing the claimed membership in the trust list. From the entry the issuer's public key is extracted.

Next, the policy performs the verification of the document's signature with respect to the bearer's public key. We also check that the trust membership claim is really eIDAS qualified. After this we get the translation entry for the trust membership claim and extract from that the issuer's public key. And then the policy performs the verification of the certificate's signature with respect to the issuer's public key. For the simple case of a Boolean trust list, TrustListEntry does not need to contain a lot of attributes, but it should contain at least the public key of the issuer, such that the signature on the certificate can be verified.

The example shows that policies can be specified on an abstract level. In fact, on this level of concerns we do not specify the entire interaction with the DNS server and the checks that need to be performed on the response. Instead, we only require, however the mechanism works in details, that the trust membership claim can be confirmed, i.e. that we can check that the issuer has signed the certificate and is a member of the eIDAS qualified trust list.

7.2 Tuple-Based Trust Scheme without Translation

This example is a simple extension of the previous one, but this time the check with the trust list will be tuple-based, i.e. the trust list will assign an attribute to the trust lists entry, namely the attribute identityProofing. When this attribute is set to inPerson, it represents that the person has proved his identity by showing up in person.

It is easy to see that the policy is an extension of the previous policy since it extends it with a last line.

```
accept(Form) :-
  extract(Form, format, document_format),
  extract(Form, certificate, Certificate),
  % Checking the certificate:
  extract(Certificate, format, eIDAS_qualified_certificate),
  extract(Certificate, pubKey, PkSig),
  extract(Certificate, issuer, IssuerCertificate),
  extract(IssuerCertificate, trustScheme, TrustMemClaim),
  % check the document was indeed signed with PkSig:
  verify_signature(Document, PkSig),
  % check the claimed trustlist membership is eIDAS qualified:
  trustscheme(TrustMemClaim, eIDAS_qualified),
  % lookup the entry on the trustlist:
```

```
  trustlist(TrustMemClaim, IssuerCertificate, TrustListEntry),
  % extract the issuer's key
  extract(TrustListEntry, pubKey, PkIss)
  % check the certificate is indeed signed with PkIss:
  verify_signature(Certificate, PkIss),
  % check that the signer showed
  extract(TrustListEntry, identityProofing, inPerson).
```

7.3 Trust Translation Scheme Scenario

As a next example, we look at a Boolean trust translation scheme scenario. This is like the Boolean trust scheme scenario, but where the trust scheme is actually foreign (e.g. a Swiss trust scheme) and needs to be translated (e.g. into a 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. The Certificate may now also be of a different format than eIDAS_qualified_certificate, and so we use a format certificate_format that includes both eIDAS and formats with similar fields.

```
accept(Form) :-
  extract(Form, format, document_format),
  extract(Form, certificate, Certificate),
  % Checking the certificate:
  extract(Certificate, format, certificate_format),
  extract(Certificate, pubKey, PkSig),
  extract(Certificate, issuer, IssuerCertificate),
  extract(IssuerCertificate, trustScheme, TrustMemClaim),
  % check the document was indeed signed with PkSig:
  verify_signature(Document, PkSig),
  % check the claimed trustlist membership is eIDAS qualified:
  trustschemeX(IssuerCertificate, eIDAS_qualified, TrustListEntry),
  % extract the issuer's key
  extract(TrustListEntry, pubKey, PkIss)
  % check the certificate is indeed signed with PkIss:
  verify_signature(Certificate, PkIss).
```

The trustschemeX predicate checks that a trustlist membership claim is either directly to the trustlist we are looking for (here eIDAS_qualified), or belongs to a trustscheme that can be translated to eIDAS_qualified. In case the claim is to eIDAS_qualified it binds TrustListEntry to the entry on that list if it is indeed there. In case the claim can be translated, the predicate sets TrustListEntry to be an entry that is a translation to the eIDAS trust list format. In this way the user can continue defining the policy as if TrustListEntry was on the eIDAS trustlist.

```
trustschemeX(IssuerCert, TrustedScheme, TrustListEntry) :-
  extract(IssuerCert, trustscheme, Claim),
  trustscheme(Claim, TrustedScheme),
  trustlist(Claim, IssuerCert, TrustListEntry).

trustschemeX(IssuerCert, TrustedScheme, TrustedTrustListEntry) :-
  extract(IssuerCert, trustscheme, Claim),
  trustlist(Claim, IssuerCert, TrustListEntry),
```

```
  encode_translation_domain(Claim, TrustedScheme, TTAdomain),
  lookup(TTAdomain, TranslationEntry),
  translate(TranslationEntry, TrustListEntry, TrustedTrustListEntry).
```

Here the encode_translation_domain, 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 IssuerCert is in a (hypothetical) Swiss scheme at URL "admin.ch" and the TrustedScheme is eIDAS_qualified, then the TTAdomain 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 translation entry for the Swiss scheme, if it exists, at eIDAS. We use lookup to get our hands on this TranslationEntry and we can then use the translate predicate together with TranslationEntry to translate the Swiss TrustListEntry to the eIDAS trust list format.

We also define trustschemeO:

```
trustschemeO(IssuerCert, TrustedScheme, TrustListEntry, immediate) :−
  extract(IssuerCert, trustscheme, Claim),
  trustscheme(Claim, TrustedScheme),
  trustlist(Claim, IssuerCert, TrustListEntry).

trustschemeO(IssuerCert, TrustedScheme, TrustedTrustListEntry, requires_translation) :−
  extract(IssuerCert, trustscheme, Claim),
  trustlist(Claim, IssuerCert, TrustListEntry),
  encode_translation_domain(Claim, TrustedScheme, TTAdomain),
  lookup(TTAdomain, TranslationEntry),
  translate(TranslationEntry, TrustListEntry, TrustedTrustListEntry).
```

One could extend this approach to work also for chains of translations. This could for instance be done by extending encode_translation_domain to also be able to produce an encoded URL when given a (possibly translated) trust list entry and a desired trust scheme. Imagine, for instance, that there is no direct translation from a Singaporean scheme to eIDAS, but the Singaporean scheme can be translated to the Swiss scheme which then again can be translated to eIDAS. We could 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(IssuerCert, SourceScheme, Hops, TrustedTrustListEntry) :−
  extract(IssuerCert, trustscheme, Claim),
  trustscheme(Claim, SourceScheme),
  trustlist(Claim, IssuerCert, TrustListEntry),
  trustschemeChainAux(TrustListEntry, Hops,TrustedTrustListEntry).

trustschemeChainAux(TrustListEntry, [], TrustListEntry).
trustschemeChainAux(TrustListEntry, [Hop|Hops], TrustedTrustListEntry) :−
  extract(TrustListEntry, trustscheme, Claim),
  encode_translation_domain(Claim, Hop, TTAdomain),
  lookup(TTAdomain, TranslationEntry),
  translate(TranslationEntry, TrustListEntry, TranslatedEntry),
  trustschemeChainAux(TranslatedEntry, Hops, TrustedTrustListEntry).
```

Here we make use of the list notation as explained in Subsection 4.2.

7.4 Trust Scheme with Delegation Scenario

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

- A reference for the mandator: In this example we assume this is an eIDAS issuer certificate (MandatorCert).
- The public key (pkSig) of the proxy, such that the signature of the proxy can be verified.
- The purpose: In this example purchase.
- The authoritative delegation provider (DP).

The ATV is required to check that the delegation provider indeed has a valid entry containing a hash (HMandate) of the delegation mandate. This check ensures that the delegation has not been revoked by the mandator. The delegation provider is a server that has the authority of determining whether the mandate is valid or not. The delegation provider provides an entry containing the entire mandate in encrypted form plus a hash of the mandate. Therefore only the proxy of the mandate can read it, but the ATV that has received the mandate can check it with the hash.

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

```
accept(Transaction) :−
  extract(Transaction, format, document_format),
  extract(Transaction, delegation, Delegation),
  extract(Delegation, format, delegationXml),
  extract(Delegation, purpose, purchase),
  checkMandate(Delegation, Transaction),
  checkMandatorKey(Delegation, eIDAS_qualified),
  checkValidDelegation(Delegation).

checkMandate(Delegation, Transaction) :−
  extract(Delegation, proxyCert, ProxyCert),
  extract(ProxyCert, proxyKey, PkSig),
  verify_signature(Transaction, PkSig),
  extract(Delegation, mandatorCert, MandatorCert),
  extract(MandatorCert, mandatorKey, PkMandator),
  verify_signature(Delegation, PkMandator).

checkMandatorKey(Delegation,TrustScheme) :−
  extract(Delegation, mandatorCert, MandatorCert),
  extract(MandatorCert, trustScheme, TrustSchemeClaim),
  trustscheme(TrustSchemeClaim, TrustScheme),
  trustlist(TrustSchemeClaim, MandatorCert, TrustListEntry),
  extract(TrustListEntry, format, generic_trustlist_format),
  extract(TrustListEntry, pubKey, PkIss),
  verify_signature(MandatorCert, PkIss).

checkValidDelegation(Delegation) :−
```

```
extract(Delegation, delegationProvider, DP),
lookup(DP, DPEntry),
extract(DPEntry, format, dp_format),
extract(DPEntry, fingerprint, HMandate),
verify_hash(Delegation, HMandate).
```

This example could be generalized to more involved situations:

- In the example we assume the mandator is represented by an eIDAS issuer certificate, but in general this could be more indirect, e.g. a certificate of the Mandator that was issued by an authority that is then on the eIDAS trust list.
- In the example the mandate contains the public key of the proxy. 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 purchases made with respect to the same mandate are of course linkable. In other words, the proxy can hide its identity behind a pseudonym, namely the public key, but there is no mechanism to hide that two purchases are made with respect to the same mandate.
- The purpose could be more fine-grained, e.g., allowing purchases only up to a certain limit.

7.5 University Case Study

Consider electronic admission to the PhD school of a university, say, the Technical University of Denmark. One of the requirements is that the candidates hold MSc Degrees (in suitable subjects, 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 of checking 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 organization who is the owner of the university certificate. The problem is that any organization could self-apply the title "university". The idea is that one can build trust lists, for instance on a national government level, of which institutions are indeed recognized as universities, probably based on adhering to certain academic standards. The European union can then choose to recognize all such trust lists from its member states and from some non-EU countries based on bilateral agreements. This recognition can again be specified as a trust translation scheme:

```
realUniversity(Diploma, Signer) :−
  extract(Diploma, UniCertificate),
  % Checking the certificate:
  extract(UniCertificate, format, eIDAS_qualified_certificate),
  extract(UniCertificate, issuerName, IssuerName),
  extract(UniCertificate, bearer, Signer),
  extract(UniCertificate, pubKey, PkSig),
  % extract (potentially foreign) trust scheme:
  extract(UniCertificate, issuer, IssuerCertificate),
```

```
  extract(IssuerCertificate, trustScheme, TrustMemClaim),
  % check the document was indeed signed with PkSig:
  verify_signature(Diploma, PkSig),
  % check the claimed trustlist membership is on an EU recognized list
  % or an equivalent one (hence the ...X):
  trustschemeX(IssuerCertificate, eu_recognized_university, TrustListEntry),
  % extract the issuer's key
  extract(TrustListEntry, pubKey, PkIss),
  % check the certificate is indeed signed with PkIss:
  verify_signature(UniCertificate, PkIss).
```

Note that this policy 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 Entity Authentication Assurance Framework

We now go a step further with TPL: actually also the enrollment process for a credential can be (at least partially) formalized in TPL, even though it is typically not part of a trust policy. This allows however to reason about trust schemes, in particular to compare trust schemes which give vital insights when designing translations. This is in particular true if all the details and regulations associated to a trust scheme are complex and hard to have an overview of. This is in fact directly related to WP 3. We use an example also considered in WP 3, from the enrollment part of the ISO standard "ISO/IEC FDIS 29115: Entity Authentication Assurance Framework". The enrollment is specified by giving Horn clauses specifying the predicates loa1, loa2, loa3 and loa4, which represents levels of assurance 1 to 4 as well as Horn clauses specifying the predicates l2req, l3req and l4req, which represents the requirements introduced on levels 2 to 4.

Parts of this example require actions that, at least for now, are rarely expected to be performed by a computer, such as checking the physical passport of the person who shows up in person for enrollment. We express these actions as predicates without specifying them further.

We see two reasons for specifying an enrollment process in this way. The first one is formalization: by specifying the enrollment in a language with a formal semantics we can avoid the ambiguities that can occur in a natural language specification. The second one is computer automation: the enrollment process could, to some extent, be performed by a computer, since the unspecified predicates could later be defined to interface with databases containing the relevant information—e.g. a database containing information of who have shown up in person for enrollment.

We first specify three levels of assurance:

```
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 which represents that the identity of the person to be enrolled is unique. Any level above 1 is defined as living up to the requirements of the all lower levels and some additional requirements. For these we have the predicates l2req, l3req, and l4req which we specify in more detail:

```
l2req(X) :-
  person(X),
```

```
  inPerson(X),
  idDocument(X, D).
```

Here, person, inPerson and idDocument are predicates that refer to the condition that the entity to be enrolled is a person, showed up in person, and has a document D to prove their identity. There are two other ways to satisfy the level 2 requirements, namely, in the case of not showing up in person and in the case of not being a person at all:

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

l2req(X) :-
  nonPerson(X),
  autoritativeInformationRecorded(X).
```

In the level 3 requirements, we 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.

```
l3req(X) :-
  person(X),
  inPerson(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., that when the applicant does not show up in person, then they must have that they possess a level-3 certificate:

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

Here, to extract that certificate from the application we have introduced the predicate hasCredential(X, C) and from the certificate we extract the level of assurance using another predicate 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).

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) :-
  nonPerson(X),
  trustedHardwareUsage(X),
  hasCredential(X, C),
```

loa(C, L),
L >= 3,
issuer(X, I),
person(I).

An advantage of modeling the evaluation process abstractly with only loa1, loa2, loa3, loa4, l2req, l3req and l4req being specified by Horn clauses is that the specification does not depend on a particular format of applications or credentials. For example, l3req is compatible with any credential technology for which one can specify a predicate loa(C, L) on credentials that extracts the level of assurance and produces failure when the credential in question does supply an assurance level.

9 Designing Translation Policies

LIGHT^{est} concerns 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 result of political decisions (e.g. bilateral agreements between countries), related to issues that cannot yet be executed by a computer (e.g. a registration process where a person needs to be physically present), or the individual decision of a company (e.g. whether the company decides to trust entities of a particular trust scheme for orders up to a certain value). However, the language TPL gives us a possibility to formalize policies and also—at least 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 formalization allows us to reason about policies and to use software tools that can perform such reasoning. In the example in the introduction, we already saw how we could use a TPL interpreter to do a bit of reasoning by asking the query "Who do we trust on delegation level 2?". By formalizing not only a policy but also the relationships and concepts behind it, we can ask similar questions on that level of abstraction. As we will see in this section, such formalizations can often be made without a lot of additional work— we simply need to formalize the relevant aspects in TPL. In this section, we expand on this "added value" of TPL.

We focus on the design of translation policies.

9.1 An Example

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

As an example, consider a translation policy from an ISO29115-scheme to an eIDAS-scheme where

- 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.

This translation policy is shown as represented by a table in table 1 and is straightforward to specify in TPL:

iso2eidas(iso_loa1, eidas_low).
iso2eidas(iso_loa2, eidas_low).

ISO LoA	eIDAS LoA
1	low
2	low
3	substantial
4	high

Table 1: A translation from levels of insurance in an ISO29115-scheme to levels of insurance in an eIDAS-scheme.

iso2eidas(iso_loa3, eidas_substantial).
iso2eidas(iso_loa4, eidas_high).

More complicated translations may be then expressed between schemes with several attributes. Keep in 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 does not even have a unique answer in every case. We can show that with TPL by making a query asking which ISO-levels are translated to eIDAS level low:

iso2eidas(X, eidas_low).

This query yields the solutions X = iso_loa1 and X = iso_loa2. A translation in the direction eidas2iso could be defined in many ways, for example even in a way such that a Substantial LoA in the eIDAS-scheme is not enough to get a LoA of 3 in the ISO29115-scheme.

9.2 Designing Translations: the Rationale Behind a Policy

A translation only depends on information that is present in the credential—in the example from the previous subsection this was 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 of 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 person and whether this person showed up in person to the issuing process. 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. (The ordering 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 the next sections, we show how to

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

9.3 The Issuing Process

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

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

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 the 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.

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

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

9.4 Translation

This section discusses how to specify a 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.

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 $\{x \mid issue_A(x) = a\} = \{x \mid issue_B(x) = b\}$. 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 truly equivalent credential; it is however recommended that the translation does *not* yield a credential that is *better* than 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 than 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 that if a translates to b then there should not be any better level b' that would satisfy 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 driver's 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 driver's 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.

10 GTPL: Graphical TPL

As a concluding chapter to this technical report we present a new idea to graphically represent a large fragment of the TPL policies: *Graphical TPL*, or *GTPL* for short. WP 6 includes the development of user interfaces with extensive studies on user experience; this aims at making intuitive, easy-to-learn interfaces for the most basic and most common policies. GTPL, in contrast, is a language for the expert user who wants to formulate rather complicated policies and yet ejoy on the amenities of a graphical language. It is indeed born out of our – the project participants – own communications and visual designs.

The central graphical metaphor of GTPL is that of a *paper form*. For instance, when applying for admission to a university, one needs to fill out a form provided by the university that has different *fields* with a predefined meaning, e.g. name, address, date of birth, and one has to attach to the form a number of documents such as a high-school diploma. An office clerk who processes the application will follow a policy for checking the documents, e.g. whether the attached diplomas indeed qualify the applicant for this study line, whether the grades are good enough, and whether the information such as name and address in the different documents indeed matches. One could thus describe this checking process as *constraints* on the fields of the forms, e.g., that certain fields match each other and that certain values are in acceptable range. Hence, one could specify a policy quite formally by putting these constraints directly into an empty form, essentially specifying *what to look for*. Such a policy is not only easy to write, it is also possible to read very quickly, as it literally gives the "overview" over what matters. Moreover, in contrast to a textual representation, it is less likely that the policy author accidentally forgets to specify a constraint on some field, since the entire form is in view.

The contributions of GTPL include the definition of a graphical language for trust policies that consists of a small number of language constructs. The language is parameterized over the concept of a form as a list of fields, and can thus be used with forms from any business domain. Moreover, we have implemented GTPL as a graphical policy editor that includes a translator to the LIGHTest TPL. This makes GTPL a formal language with a precise *semantics* (through the semantics of TPL) and it immediately makes the policies usable in LIGHTest and its automated trust verifier [2]. While GTPL is closely related to LIGHTest, the idea of specifying policies by constraints on fields of forms is general: We see this as a contribution towards language design that abstracts from irrelevant technical details and allows users to focus on the business logic.

We introduce GTPL in the following by a concrete example of trust policies for an auction house that wants to allow online bids. This allows us to introduce all constructs of GTPL step by step as a collection of policy rules, where each rule describes one sufficient condition for the auction house to accept an online bid. We then summarize the general concepts of GTPL in a textual syntax. This syntax both describes the data structures of the GTPL editor and is the basis for the semantics by translation to LIGHTest TPL.

10.1 A Running Example

We introduce the Graphical Trust Policy Language GTPL by using an example of a classical auction house that wants to extend its traditional business to electronic bidding and formulate trust policies for that matter. This should be based on a trust infrastructure like LIGHTest [2], and we will introduce the relevant concepts of LIGHTest along the way.

The auctions 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 does not want to put any entrance barrier for new customers who just "stumbled" upon an item by an Internet search, on the another hand they want to avoid that, for instance, somebody practically anonymously bids on an item just to get the price up and then do not pay if that bid was the highest.

This is a classical *trust* problem. The classical (non-electronic) solutions are that customers have to bring references from other auction houses or a bank statement, or be present at the auction in person, proving their identity before the auction starts. The point of trust infrastructures like LIGHTest is to facilitate these aspects in the digital world so that one can benefit from the large potential of digitalization without losing the security and trust guarantees of the classical non-digital world. This example allows us to illustrate GTPL with realistic policies that an auction house may want to choose.

10.1.1 Bidding Forms

Auction houses typically allow customers to bid via standard (non-digital) mail if they cannot be physically present at the auction house. The bidder would tell the auction house a maximum bid for a particular item, and the auction house could accordingly act as if the bidder was present at the auction and place bids for the customer up to the maximum bid. For this purpose, each auction house would have their own bidding form: a paper sheet bearing the name of the auction house and the particular auction, like "The Auction House 2018". The form contains fields to fill in, such as the personal information of the bidder and a list of items (the lot numbers and the maximum bid) and finally a field where the bidder must sign the form. This signed form is then mailed to the auction house.

The first step of digitization for auction houses was providing online auction catalogs, where customers can click on items and place a bid. This would basically lead to an electronic version of the classical paper bidding form, and it is sent to the auction house using https or simply email. Such an electronic bidding form could look like the XML snipped in Fig. 2—for simplicity we consider bidding only on a single item. We have here already included a field for the digital signature, which is actually still optional in many of today's online bidding solutions. If used, it would be a digital signature on a hash of the document.

We consider the XML-based form in Fig. 2 as *concrete syntax*, since there are many different ways to convey the same informations, but the essence, the *abstract syntax*, is that it is a list of attribute-value pairs, as shown in Fig. 3, where every value is a blank box to be filled, and every attribute is an annotation like "Bidder Name" that declares the meaning of the corresponding value. As a first approximation let us say that the content of the blank boxes will be a string. Moreover, the form carries a title, identifying the meaning of the form as an entirety, so that nobody accidentally considers this form, say, as a passport. This title corresponds in the non-electronic world to having the name of the auction house and the particular auction printed on the paper bidding form. (This is crucial also to prevent a hacker to *replay* a bidding form at the next auction.) Both the signature and the certificate fields are special fields, while the other fields are generic and carry no built-in meaning for GTPL.

```
<AUCTIONHOUSEFORMAT
 auctionID="AUCTION18">
<bidder>...</bidder>
<address>
  <street>...</street>
  <city>...</city>
  <country>...</country>
</address>
<bid lotno="..."
      amount="...">
<signature>
jmj7l5rSw0yVbvlWAYkKYBwk
</signature>
<Certificate>
...
</Certificate>
</AUCTIONHOUSEFORMAT>
```

Figure 2: Example form in XML

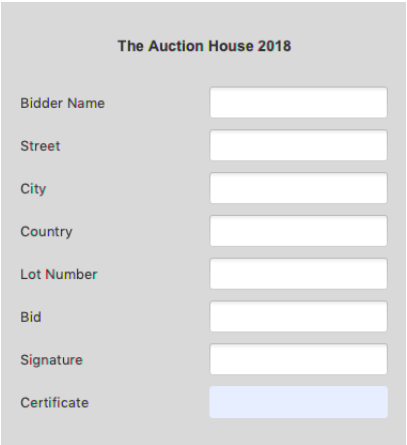

Figure 3: Graphical representation of the form in GTPL

It is thus possible to connect a variety of forms to GTPL, as it is for LIGHT^{est} in general: one simply needs to define a new format with a parser for the new form (and a connection to signature verification for the used signature scheme), and add it to the library of formats, as explained in Section 5.

10.1.2 The Layout of the GTPL Application

Fig. 4 gives the overview of the GTPL application. The left part (labeled "GTPL") is the list of policy rules defined so far, the middle part (labeled "Working space") is the policy rule currently under edit, and the right part (labeled "Format Library") is a palette of formats and certificates that are currently available. The normal workflow to create a new policy rule is to select a format or certificate from the library and drag it to the center workspace, to open a new blank form. The figure shows this for the format "The Auction House 2018". One can now make constraints on this form, give it a name (in the "Rule name" field) and then click the "Add policy" button. Then it will appear in the list of the rules on the left; from there it can be selected later for editing (or deleting). Finally one can store the rules in GTPL format or export them to LIGHTest TPL for use in the LIGHTest architecture.

10.1.3 The Hello World of GTPL

The basic idea is that we can use this simple graphical representation of the "empty" form as a basis for describing trust policies. For instance, let us define as a first example policy rule in Fig. 5 where the auction house wants to accept any bids up to 100 Euro; note that the currency is implicit in the format (it may explicitly write that in the syntax of the field). The policy is specified by entering into the bid field the text `<=100` and leaving blank all other fields. This means that this policy rule has only one requirement, namely that the bid is a number up tp 100. In particular we do not require a signature, i.e., for bids below 100 Euro, this auction house is not worried about trust.

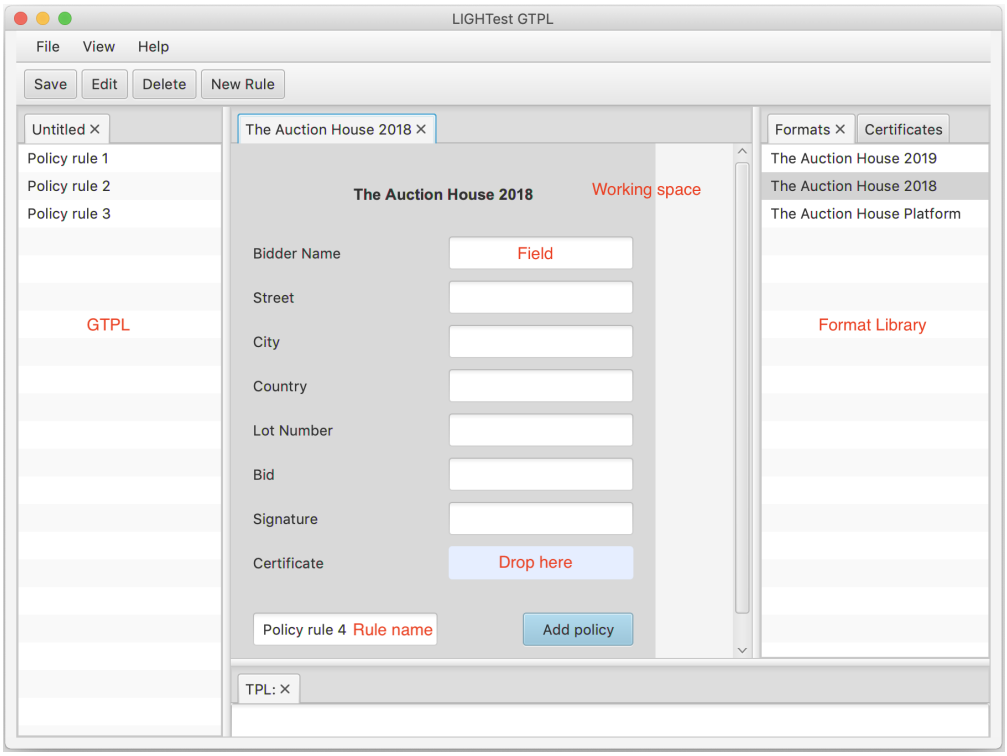

Figure 4: The layout of the GTPL application (the annotations in red are not part of the GUI).

The basic concept of filling a constraint into a field is to either demand a particular value, e.g. entering `Denmark` for the country field, or a comparison with a value like `<=100`. The latter requires that the corresponding field of that format is defined to have an ordinal *type*. Such ordinal types can also be defined as part of the library, e.g., we could have a type *rating* with ordered elements `standard` < `gold` < `premium`. Moreover, we plan to allow that a field could be any value in a list, e.g., the policy author can specify a list of countries `CL` and constrain the country field of the form to be any value of the list by writing `in CL`.

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

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

Here, a form is accepted if it is of the right format – this ensures that it has a bid-attribute—and that the BID is at most 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.

10.1.4 Checking Signatures

As a second policy rule, the auction house accepts any bid up to 1500 Euro if it is signed by an eIDAS qualified signature [4]. To that end, we use that the Signature field has a distinguished meaning: we specify in this field the public key with respect to which the signature must verify.

Figure 5: A first graphical policy rule

It is part of the format definition which part of the document is signed. The graphical convention is that the signature includes *all fields above the signature field*, e.g., in the auction house form the signature comprises all fields except the certificate field.[6]

In a signature field we practically never want to specify a particular fixed public key, but specify that it relates to a given certificate. If we use again the metaphor of a paper form, then a certificate would be an *attachment* to the main form, i.e., the person submitting the form provides additional relevant information that itself has some structure. Thus, such attachments can be regarded as forms themselves, e.g., a certificate may be in the X.509-format. Further, we want to "bind" attachments to the main document in a suitable way; one could have for this a special container format (like Associated Signature Containers [8]) or simply directly have another field in the form for such attachments, like the "Certificate" field in our example. The value of such an attachment field must then also be a format.

To embed this concept into our graphical language, we have adapted the notion of a *sub-form*, i.e., a field in a form can host an entire form itself. Fields of this type are highlighted blue in the GTPL application, indicating that the policy author can drag a form from the library onto this blue field. Fig. 6 shows the result of dragging a certificate format for eIDAS certificates to the certificate field, inserting a blank subform for entering constraints. Observe that in Fig. 6 we have now specified the *variable* `PK` both on the signature of the main form and in the `pubKey` field of the certificate. This means that the signature of the main form must verify with the public key we can extract from the eIDAS certificate. The eIDAS certificate is itself signed with yet another key `PkIss`—this is also a variable.[7] This illustrates the fact that a certificate is itself a signed document and without verifying the signature of the certificate it does not mean much. In standard PKIs one may have an arbitrary long sequence of certificates until one reaches the certificate from an already trusted organization. Here, instead, we want to formalize that the certificate is part of a particular trust scheme, eIDAS in this in this example, i.e., the issuer is member of a particular trust list headed by the EU.

There are several possible ways to organize and implement the check of this trust membership claim. Essential to the policy author are only two aspects. First, one identifies the desired trust scheme, here `[eIDAS_qualified]`. We require that such trust lists are defined as part of the library of formats and certificates, in particular which URL is the relevant authority for a particular trust scheme. Second, if the lookup of the trust list entry is successful (otherwise the

[6]This should also include the kind of form it is, i.e., the format name; this is implied by the standard requirements on the disjointness of formats [11].

[7]Recall that the scope of the signature field of the auction house format spans all fields above it; similarly, the scope of the signature field of the certificate are all fields of the subform above the signature field (here, all fields).

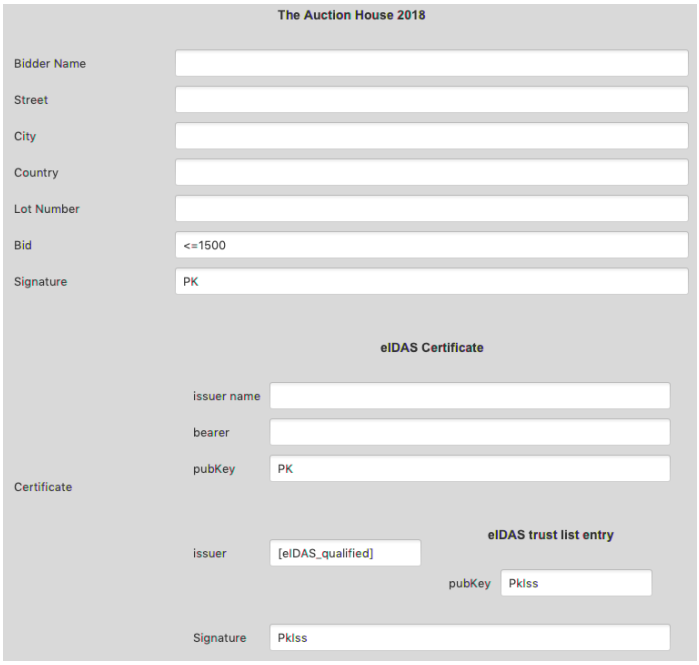

Figure 6: A graphical policy rule with eIDAS qualified signature

policy is not satisfied), one may specify constraints on the trust list entry that may contain a number attributes like a trust level. This entry is depicted graphically in GTPL to the right of the trust list—again as a form. In the example we assume that the entry contains a public key and specify that it has to be the same variable `PkIss` that we also have in the field of the eIDAS certificate's signature. This means, the eIDAS certificate must verify against the public key from the trust list entry, i.e., the certificate was indeed issued by a member of the eIDAS qualified trust scheme.

This completely explicit handling of trust list entries allows to specify quite complex policies when the trust list entry contains more information, while for simple Boolean trust lists (i.e., just checking that the entity is on the trust list like in this example), this is a bit overkill. Therefore we allow here also a simplified notation as syntactic sugar, namely one could just specify `[eIDAs_qualified]` in the `Signature` field, i.e., keeping the subform of the eIDAS certificate implicit.

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, certificate, Certificate),
  extract(Certificate, format, eIDAS_qualified_certificate),
  extract(Certificate, pubKey, PK),
  extract(Certificate, issuer, IssuerCertificate),
  extract(IssuerCertificate, trustScheme, TrustMemClaim),
  trustscheme(TrustMemClaim, eIDAS_qualified),
  trustlist(TrustMemClaim, IssuerCertificate, TrustListEntry),
```

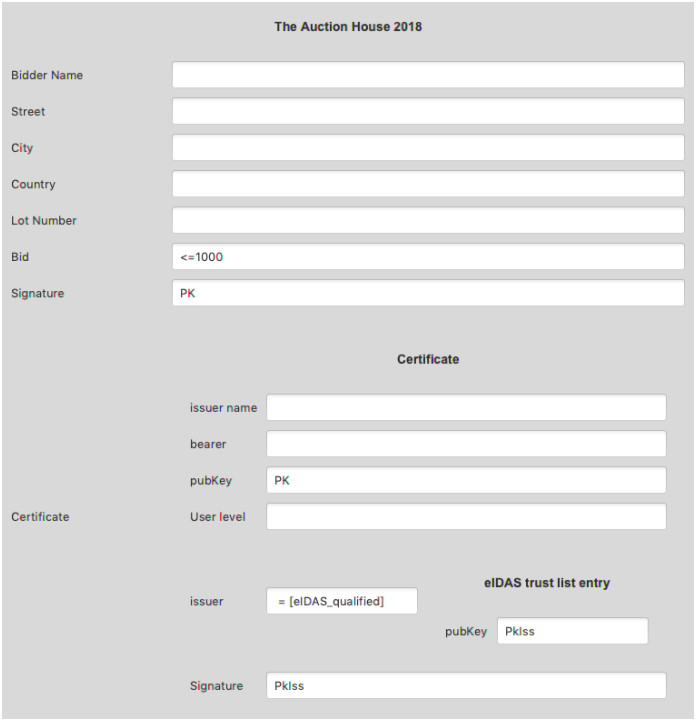

Figure 7: A graphical policy rule with eIDAS equivalent signature

```
% extract the issuer's key
extract(TrustListEntry, pubKey, PkIss),
verify_signature(Certificate,PkIss),
verify_signature(Form,PK).
```

10.1.5 Allowing for Trust Translation

LIGHTest facilitates also the specification of trust translations, e.g. the authority of a trust scheme can specify that they regard another trust scheme as equivalent, for instance the European Union may declare that they regard some foreign trust scheme as equivalent to eIDAS. It is of course the decision of each policy author whether they want to accept trust translation in the first place. For our auction house example, we could imagine the following policy: we do accept certificates with foreign trust schemes that eIDAS considers equivalent, but set a lower limit on the bid in this case. Fig. 7 shows just that: we have replaced `[eIDAS_qualified]` with `=[eIDAS_qualified]`, meaning we do allow eIDAS, or one that eIDAS considers equivalent, and thereby allowed trust translation, but we have capped the bid to `<=1000` in this case. Also in this case the certificate is not an eIDAS certificate, but a generic certificate.

```
accept(FORM) :-
  extract(FORM, format, theAuctionHouse2018format),
  extract(FORM, bid, BID), BID <= 1000,
  extract(FORM, certificate, Certificate),
  % The following is the standard check of the certificate
```

```
% and its trust membership claim specialized for eIDAS
extract(Certificate, format, certificate),
extract(Certificate, pubKey, PK),
extract(Certificate, issuer, IssuerCertificate),
extract(IssuerCertificate, trustScheme, TrustMemClaim),
% check the claimed trustlist membership is eIDAS qualified or equivalent:
trustschemeX(IssuerCertificate, eIDAS_qualified, TrustListEntry),
% extract the issuer's key
extract(TrustListEntry, pubKey, PkIss),
% check the certificate is indeed signed with PkIss:
verify_signature(Certificate, PkIss),
% check the document was indeed signed with PK:
verify_signature(FORM, PK).
```

10.1.6 Putting it all together

We have specified a number of policy rules. They are collected all in the left tab of the GTPL application (cf. Fig. 4). All these graphical trust policy rules are put together by *disjunction*, i.e., in our example, a bid is accepted if *any* of the rules match. The order of the rules in the left-hand tab of the interface only determines in which order they are checked, so it makes sense to put most common cases first, and the rarer cases later.

10.2 GTPL Syntax and Semantics

While GTPL is a graphical language, it is also formal in the sense that is has a precise syntax and semantics. This is crucial for trust decisions and thus for all machinery that works on GTPL specifications. The abstract syntax of GTPL is defined in terms of Java data structures; for reading convenience, we use an EBNF-style notation in Fig. 8. Let us briefly review each item with an intuitive semantics. The formal semantics is defined afterwards by translation to the LIGHT^{est} TPL.

At the top level, GTPL is a list of forms, where each form means one rule of the policy, like in figures 4–6. The meaning of the full policy is the *disjunction* of the rules, i.e., the policy is fulfilled, if at least one rule is. A form consists of a formatname (like "Auction House 2018") and a list of attribute-value pairs. The meaning of this policy is that firstly the given input must be parsable as the given format indicated by formatname, and secondly the conjunction of the constraints specified by the attribute-value pairs must be satisfied.

$$
\begin{array}{rcl}
\mathit{GTPL} & ::= & \mathit{Form}^{\star} \\
\mathit{Form} & ::= & \mathit{Formatname}(\mathit{AttVal}^{\star}) \\
\mathit{AttVal} & ::= & (\mathit{Attributename}, \mathit{Value}) \\
\mathit{Value} & ::= & \mathit{BLANK} \mid \mathit{Constant} \mid \mathit{Variable} \mid \mathit{op}\ \mathit{Constant} \mid \mathit{op}\ \mathit{Variable} \\
 & \mid & \mathit{in}\ \mathit{Listname} \mid \mathit{Form} \mid =^{?}\ [\mathit{Trustlist}]\ \mathit{Form}^{?} \\
\mathit{op} & ::= & < \mid > \mid <= \mid >= \mid ==
\end{array}
$$

where *Formatname*, *Attributename*, *Variable*, *Trustlist* and *Listname* are alphanumeric identifiers and *Constant* is either a sequence of digits or printable ASCII characters in quotes

Figure 8: Syntax in GTPL in a textual/data structure form; terminal symbols are set in blue.

An attribute-value pair consists of an attribute name and value, of course. Here, the attribute name (like "Country") indicates one of the fields of the form, and the value gives a constraint on the value of this field. The first possibility is *BLANK* meaning that the policy author left the field blank, and thus there is no constraint on this field. Second, it can be a constant (either numeric or an ASCII string in quotes), meaning the value must be just that. Third, it can be a variable (like `PK` in the examples). The meaning of a variable is that the value can be arbitrary, but all fields where the same variable is specified (in the present rule) must have the same value. The fourth and fifth possibilities are a comparison operator followed by a concrete value or a variable. This is can only be used on fields where an ordering is defined (e.g. numerics, levels, dates). The sixth possibility is to specify membership in a user-defined list (e.g. the country must be one in a given list of countries). The seventh possibility is a form itself. This can be only used on fields that are highlighted blue, i.e., that allow for a subform as a value, e.g. the certificate field in the examples. The meaning is simply that in this case the condition specified for the subform are checked as expected.

The last possibility of a value has several options, and can be used for the `issuer` field of certificates. The most basic form is to specify only a trustlist (like `[eIDAS_qualified]`). The meaning is that this issuer field contains an issuer certificate containing a URL. The issuer certificate and URL are used to look up an entry of a trust list. The constraints we specify here are (a) that trust list of the URL indeed belongs to the specified trust list (like eIDAS), (b) that the trust list entry indeed exists and (c) that it contains a public key that verifies the signature of the given () certificate. One option for this trust list specification is to specify also a form (the fact that this is optional is specified by the question mark). If specified, the meaning is that the returned trust list entry must meet the constraints expressed by the given form. This allows for trust schemes where the trust list entry contains further entries, e.g., a trust level that can then be constrained as part of the policy. Finally, one can also put an equal sign in front of the trust list specification and thereby allow trust translation.

The above explained last possibility of a value can also be used in the `signature` field. E.g. in the case the value is `[eIDAS_qualified]` this will mean that the form is signed by a signature that comes from an eIDAS qualified certificate. On the technical level, this is done by an expansion to the above case where `[eIDAS_qualified]` is in the `issuer` field. For details see the translation below. Translation is also supported using the `=[eIDAS_qualified]` notation. Delegation also supports the `[eIDAS_qualified]` notation: If the notation occurs in the `delegation` field of a form, then it signifies that the form is signed by a signature that comes from a delegation with an eIDAS qualified issuer certificate – see also section 7.4.

The semantics starts with calling $[\![F(Attlist)]\!]^{\text{Toplevel}}$ 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 of the translation is given here. Note that if an expression matches multiple cases of the specification then it is the first one that defines the translation.

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

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

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

$[\![(\text{signature}, t) : Attlist]\!]^{\text{Attlevel}}_{Form} =$
$[\![Attlist]\!]^{\text{Attlevel}}_{Form}$
, verify_signature$(Form, t)$
where t is either a concrete value or a variable.

$[\![(\text{signature}, [Value]) : Attlist]\!]^{\text{Attlevel}}_{Form} =$
$[\![$
$(\text{signature}, PK) :$
$(\text{certificate}, \text{generic\_format}((\text{pubKey}, PK) : (\text{issuer}, [Value]))) :$
$Attlist$
$]\!]^{\text{Attlevel}}_{Form}$
where PK is a new variable.

$[\![(\text{signature}, = [Value]) : Attlist]\!]^{\text{Attlevel}}_{Form} =$
$[\![$
$(\text{signature}, PK) :$
$(\text{certificate}, \text{generic\_format}((\text{pubKey}, PK) : (\text{issuer}, = [Value]))) :$
$Attlist$
$]\!]^{\text{Attlevel}}_{Form}$
where PK is a new variable.

$[\![(\text{issuer}, [Value]) : Attlist]\!]^{\text{Attlevel}}_{Form} =$
$[\![(\text{issuer}, [Value]\ \text{generic\_format}((\text{pubKey}, PKIss))) : (\text{signature}, PKIss) : Attlist]\!]^{\text{Attlevel}}_{Form}$
where $PKIss$ is a new variable.

$[\![(\text{issuer}, = [Value]) : Attlist]\!]^{\text{Attlevel}}_{Form} =$
$[\![(\text{issuer}, = [Value]\ \text{generic\_format}((\text{pubKey}, PKIss))) : (\text{signature}, PKIss) : Attlist]\!]^{\text{Attlevel}}_{Form}$
where PK is a new variable.

$[\![(\text{issuer}, [Value]\ R(Attlist_s)) : Attlist]\!]^{\text{Attlevel}}_{Form} =$
, extract$(Form, \text{issuer}, IssuerCertificate)$
, extract$(IssuerCertificate, \text{trustScheme}, TrustMemClaim)$
, trustscheme$(TrustMemClaim, Value)$
, trustlist$(TrustMemClaim, IssuerCertificate, TrustListEntry)$
, $[\![R(Attlist_s)]\!]^{\text{Formlevel}}_{TrustListEntry}$
, $[\![Attlist]\!]^{\text{Attlevel}}_{Form}$
where $IssuerCertificate$, $TrustMemClaim$ and $TrustListEntry$ are new variables.

$[\![(\text{issuer}, = [\mathit{Value}]\ R(\mathit{Attlist}_s)) : \mathit{Attlist}]\!]^{\text{Attlevel}}_{\mathit{Form}} =$
$, \text{extract}(\mathit{Form}, \text{issuer}, \mathit{IssuerCertificate})$
$, \text{extract}(\mathit{IssuerCertificate}, \text{trustScheme}, \mathit{TrustMemClaim})$
$, \text{trustschemeX}(\mathit{IssuerCertificate}, \mathit{Value}, \mathit{TrustListEntry})$
$, [\![R(\mathit{Attlist}_s)]\!]^{\text{Formlevel}}_{\mathit{TrustListEntry}}$
$, [\![\mathit{Attlist}]\!]^{\text{Attlevel}}_{\mathit{Form}}$
where $\mathit{IssuerCertificate}$, $\mathit{TrustMemClaim}$ and $\mathit{TrustListEntry}$ are new variables.

$[\![(\text{delegation}, [\mathit{Value}]) : \mathit{Attlist}]\!]^{\text{Attlevel}}_{\mathit{Form}} =$
$, \text{extract}(\mathit{Form}, \text{delegation}, \mathit{Delegation})$
$, \text{extract}(\mathit{Delegation}, \text{format}, \text{'delegationxml'})$
$, \text{checkMandate}(\mathit{Delegation}, \mathit{Form})$
$, \text{checkMandatorKey}(\mathit{Delegation}, \mathit{Value})$
$, \text{validDelegation}(\mathit{Delegation})$
$, [\![\mathit{Attlist}]\!]^{\text{Attlevel}}_{\mathit{Form}}$
where $\mathit{Delegation}$ is a new variables.

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

$[\![(\mathit{attname}, R(\mathit{Attlist}_s)) : \mathit{Attlist}]\!]^{\text{Attlevel}}_{\mathit{Form}} =$
$,\text{extract}(\mathit{Form}, \mathit{attname}, \mathit{Subform}),$
$[\![R(\mathit{Attlist}_s)]\!]^{\text{Formlevel}}_{\mathit{Subform}}$
$[\![\mathit{Attlist}]\!]^{\text{Attlevel}}_{\mathit{Form}}$
where $\mathit{Subform}$ is a new variable.

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

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 that first checks the format of the form and then the function continues by translating the 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., if the third argument is a value the translated policy will check that the attribute in the format has that value, and if the third argument is the first occurrence of a variable then that variable is bound to the value corresponding to the attribute in the format. 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.

In case that an 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.

The attribute signature is a special case: in this case we first verify the form with its corresponding public key. As an additional step, not specified in the translation, our implementation arranges the clauses such that they have the verification of the signatures in the end; the reason is that putting the verification in the end ensures that the public key has been extracted before we verify something with it. In fact, we assume here that the form itself provides in case of a signature enough information to determine which public key corresponds to the signing key, the part of the text that is signed, and the signature itself. In case the notation [trustscheme] or =[trustscheme] is used for the signature this is a shorthand for a more involved process as seen in the auction house example and specified in the translation. The expanded form contains the special issuer attribute which we now explain.

The issuer attribute is also special: the value of such an attribute shall use the notation [trustscheme] to indicate the requirement that the certificate issuer must be on the given trustscheme, or shall use 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 trustScheme is extracted from the issuer certificate and we use the predicate trustscheme to verify that the URL indeed belongs to the desired trust scheme; then we, using trustlist, look up if the issuer certificate is indeed in the trustlist on the URL and 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, but uses the predicate trustschemeX instead to allow for translations.

LIGHT^{est} also provides another user interface for specifying policies called the Natural Language Layer [21]. We provided a translation from this user interface to TPL by translating the policies in the Natural Language Layer to forms in GTPL and thereafter using the translation from GTPL to TPL to arrive at TPL code.

11 Verification

Several aspects of TPL and LIGHT^{est} are verified. In particular we have looked at verifying trust decisions and at protocol verification.

11.1 Verifying Trust Decisions

Jim [9] introduced the trust management system SD3 with certified evaluation. When SD3's evaluator decides whether a transaction lives up to a policy, it provides a proof of this. A separate proof checker can then check the proof's correctness. The proof checker is a very simple program, and thus it is easy to inspect and understand its code – making it highly trustworthy.

TPL also allows certified evaluation, with the crucial difference that the trustworthiness of the proof checker does not come from a claim that its code is simple. Instead, we base our proof checker on the prover RP_x [18, 17, 19, 20] which is, with exception of its parser, verified in Isabelle/HOL [15]. Isabelle is a proof assistant i.e. a computer program that allows its user to prove theorems in e.g. computer science. The idea is that Isabelle ensures the proofs' correctness because RP_x is proved in Isabelle/HOL to be sound and complete for first-order clausal logic.

For successful queries the TPL interpreter within the ATV (that is not formally verified) should construct a proof certificate as a triple $(p, (q_1, ..., q_n), b)$ where p is the policy, $q_1, ..., q_n$ is the query and b is a record of the results from all calls to the built-in predicates that happened

during execution including server-lookups, extractions from forms, signature verification and comparisons of e.g. numbers. The proof checker works as follows:

1. Let c be $p \wedge (\forall(\neg q_q \vee \cdots \vee \neg q_n)) \wedge b$ encoded in the input format of RP_x. Here $\forall F$ represents the universal quantification of F over all variables occurring in F. Notice also that p is a conjunction of clauses following the logical semantics of clauses specified earlier in this technical report.

2. Run RP_x on c.

3. If RP_x is successful in proving the formula unsatisfiable, then the proof check was successful.

The idea is that we want to prove $p \wedge b \models \exists(q_1 \wedge ... \wedge q_n)$. Here $\exists F$ represents the existential quantification of F over all variables occurring in F. This is equivalent to proving that $p \wedge (\forall(\neg q_q \vee \cdots \vee \neg q_n)) \wedge b$ is unsatisfiable, and RP_x can do that for any correct positive decision thanks to its soundness and completeness.

Our integration of RP_x in TPL is currently in the state of an early prototype. We have written a program that can encode a triple $(p, (q_1, ..., q_n), b)$ in RP_x's input format. Using this program, we have run RP_x on a number of such triples and seen that it gives the correct result. One could argue that this is not necessary since RP_x is formally verified, however, one should, as e.g. Paulson [16] recently pointed out, not see formal verification as a replacement for testing. Indeed we have run RP_x on a number of encoded triples but more systematic testing would be needed to ensure a production quality certifier. Notice also that while the core of RP_x is verified, the encoding of the formula $p \wedge (\forall(\neg q_q \vee \cdots \vee \neg q_n)) \wedge b$ in ${RP_x}'s$ input format is still left unverified and so is the parser of RP_x.

11.1.1 Interpreter

Working on the above verification of trust decisions we noticed opportunities for improving the TPL interpreter – in particular to make sure that the interpreter implements unification correctly.

A very simple policy is the following:

```
id(X,X).
```

with the query `id(c,c)`. If you follow the description of the general resolution algorithm from a standard text book on prolog e.g. "4.2.1" in [3], you will reach a state in which `id(X,X)` and `id(c,c)` have to be unified. If you try to unify them using the unification algorithm from "3.2.2" in the same book, you will see that the unification algorithm will give as result $\{X \leftarrow c\}$. If you continue following the description of the general resolution algorithm you will see that the result from the policy should be "true".

We noticed however that the trust policy interpreter instead gave a warning

```
Warning: 'id(X1, X1).' A Variable in a single Statement makes no sense!
```

and the result `false`.

Based on this discovery we improved the interpreter with new implementations of the general resolution algorithm and the unification algorithm. With these improvements the above policy and query gave the expected results.

11.2 Verifying Stateful Security Protocols

The LIGHT^{est} project is built on top of communication systems such as stateful security protocols that exchange information with servers with persistent data storage. For instance, the U2F protocol from FIDO[8] has been proposed as a means of providing two-factor authentication for mobile identities[9]. These protocols are assumed to be secure and any flaws they may have can affect the guarantees of the LIGHT^{est} project. It is therefore important to ensure that they work as intended, and there exists many automated tools to verify protocols that one can use for this purpose.

As part of our ongoing research we are integrating automated protocol verifiers with Isabelle to produce machine-verified security proofs, giving us the extreme correctness guarantee that Isabelle provides. The idea is to extend the OFMC [13] protocol verifier to yield a "proof" of correctness and then automatically verify this "proof" within Isabelle using an Isabelle-formalized protocol verification framework [6]. This means that we do not rely on the correctness of automated verification tools at all—the Isabelle-formalized checks will simply fail if the output of the verifier is wrong.

Moreover, the protocols used by LIGHT^{est} run in an environment together with other protocols—some of which may be flawed—and their security may depend on the other protocols in the environment. Even if the environment consists only of a handful of protocols, their combination (e.g. running in parallel) results in very complex systems where verification quickly becomes infeasible. To mitigate these problems there exists results that provide conditions under which protocols can safely be run together, even in the presence of flawed ones (given reasonable assumptions on what can be shared between protocols). We have previously proven such a result for stateful protocols in Isabelle [7] and by integrating automated protocol verifiers with this framework it should be possible to apply these results.

12 Conclusions

We have introduced the trust policy language (TPL) for describing 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 powerful enough for our purposes. TPL is powerful in the sense that with its executable semantics it serves as a programming language in which we can write policies as programs. TPL is simple in the sense that its language is simply that of Horn clauses which from a logical perspective has a simple semantics namely that of first-order logic. A particular benefit of using Prolog-style Horn clauses is the direct executability of TPL policies. This also makes it feasible to implement the machinery of LIGHT^{est} on this basis and to prove implementations correct. Another benefit is accountability: one can easily prove that a decision was made correctly adhering to a given policy.

For simple business cases with rather straightforward policies one may argue that not much is gained from the powerfulness and simplicity of TPL. On the other hand, for complicated policies the powerfulness of the language may be necessary to even state them, and the simple semantics may be needed to resolve disagreements over the meaning of a policy. Notice also that powerfulness and simplicity come for free—the powerfulness and simplicity do not make it difficult to express simple policies as shown by the numerous examples in this technical report. Furthermore, there is work on usability in LIGHT^{est}—an example being GTPL. This language is designed with a wide variety of people in mind, who do not necessarily have a background

[8] `https://fidoalliance.org/`
[9] `https://www.lightest.eu//about/related_projects/`

in computer science. That it is based on the more powerful TPL comes at no extra cost, but ensures that it has a simple logical basis.

References

[1] M. Y. Becker, C. Fournet, and A. D. Gordon. SecPAL: Design and Semantics of a Decentralized Authorization Language, 2006. Microsoft Research Technical Report MSR–TR–2006–120.

[2] B. P. Bruegger and P. Lipp. LIGHTest–A Lightweight Infrastructure for Global Heterogeneous Trust Management. In *Open Identity Summit 2016*, 2016.

[3] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard: Reference Manual*. Springer, 1996.

[4] N. Engelbertz, N. Erinola, D. Herring, J. Somorovsky, V. Mladenov, and J. Schwenk. Security analysis of eidas - the cross-country authentication scheme in europe. In *12th USENIX Workshop on Offensive Technologies, WOOT 2018*, 2018.

[5] 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.

[6] A. V. Hess. Typing and Compositionality for Stateful Security Protocols, 2019. PhD Thesis, DTU.

[7] A. V. Hess, S. A. Mödersheim, and A. D. Brucker. Stateful Protocol Composition. In *Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part I*, pages 427–446, 2018.

[8] E. T. S. Institute. Electronic signatures and infrastructures (esi); associated signature containers (asic). Technical Report ETSI EN 319 162-1 V1.1.1, 2016.

[9] T. Jim. SD3: A trust management system with certified evaluation. In *2001 IEEE Symposium on Security and Privacy, Oakland, California, USA May 14-16, 2001*, pages 106–115, 2001.

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

[11] 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.

[12] S. Mödersheim, A. Schlichtkrull, G. Wagner, S. More, and L. Alber. TPL: A Trust Policy Language. In *ITIPTM 2019*. (Accepted).

[13] S. Mödersheim and L. Viganò. The open-source fixed-point model checker for symbolic analysis of security protocols. In A. Aldini, G. Barthe, and R. Gorrieri, editors, *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*, volume 5705 of *Lecture Notes in Computer Science*, pages 166–194. Springer, 2009.

[14] S. A. Mödersheim and B. Ni. GTPL: A Graphical Trust Policy Language. *Open Identity Summit 2019*, 2019.

[15] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[16] L. C. Paulson. Computational logic: its origins and applications. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 474(2210):20170872, 2018.

[17] A. Schlichtkrull, J. C. Blanchette, and D. Traytel. A verified functional implementation of Bachmair and Ganzinger's ordered resolution prover. *Archive of Formal Proofs*, 2018. `http://isa-afp.org/entries/Functional_Ordered_Resolution_Prover.html`, Formal proof development.

[18] A. Schlichtkrull, J. C. Blanchette, and D. Traytel. A verified prover based on ordered resolution. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 152–165, 2019.

[19] A. Schlichtkrull, J. C. Blanchette, D. Traytel, and U. Waldmann. Formalization of Bachmair and Ganzinger's ordered resolution prover. *Archive of Formal Proofs*, 2018, 2018.

[20] A. Schlichtkrull, J. C. Blanchette, D. Traytel, and U. Waldmann. Formalization of Bachmair and Ganzinger's ordered resolution prover. *Archive of Formal Proofs*, 2018. `http://isa-afp.org/entries/Ordered_Resolution_Prover.html`, Formal proof development.

[21] S. Weinhardt and O. Omolola. Usability of policy authoring tools: A layered approach. In *International Conference on Information Systems Security and Privacy*, 2019.