TPL: A Trust Policy Language*

Sebastian Mödersheim[1], Anders Schlichtkrull[1], Georg Wagner[2], Stefan More[2], and Lukas Alber[2]

[1] DTU Compute, Formal Methods, Technical University of Denmark, 2800 Kongens Lyngby, Denmark {samo,andschl}@dtu.dk
[2] Institute for Applied Information Processing and Communications (IAIK), Graz University of Technology, Inffeldgasse 16a, 8010 Graz, Austria {gwagner,smore,lalber}@iaik.tugraz.at

Abstract. We present TPL, a Trust Policy Language and Trust Management System. It is built around the qualities of modularity, declarativity, expressive power, formal precision, and accountability. The modularity means that TPL is built in a way that makes it easily adaptable to different types of transactions and signatures. From the aspect of declarativity and expressive power, the language is built such that policies are always formulated in a positive form and the language is Turing complete. The formal precision and accountability of the language eliminates ambiguity and allows us to achieve verified evaluations. The idea is that for any decision, the system can generate a proof that can then be checked by a prover that is formally verified, in Isabelle/HOL, to be sound with respect to a first-order logic semantics.

1 Introduction

We introduce **TPL** – not only a **T**rust **P**olicy **L**anguage but also a trust management system geared to support and integrate today's existing trust schemes to create a trust infrastructure. A trust management system is a system that helps companies and organizations to automatically process trust decisions about electronic transactions they receive. TPL helps to specify and automatically implement a company's business policy for trust decisions.

TPL is designed in the context of the LIGHT^{est} project that aims to create a *Lightweight Infrastructure for Global Heterogeneous Trust management in support of an open Ecosystem of Stakeholders and Trust schemes*. The idea is that there are a number of trust schemes like the European eIDAS, but no scheme on which the whole world agrees on. To achieve this TPL supports different formats of electronic documents and transactions. It also allows authorities behind a trust scheme to define translations from other schemes. Translations can be automatically processed, but are only "recommendations": a policy designer can decide whether to accept trust translations. TPL also supports trust delegations.

* This work was supported by the LIGHT^{est} project, which is partially funded by the European Commission as an Innovation Act as part of the Horizon2020 program under grant agreement number 700321.

TPL has simple, clear and precise semantics as first-order clauses interpreted with respect to an environment representing TPL's interaction with the outside world. Despite the simplicity, the ATV's implementation – the Automated Trust Verifier connecting parsers and server lookups with logical evaluation – is complex. A concern is the reliability of trust decisions, i.e., that bugs in a component cannot lead to false positives. Thus TPL's architecture allows for boiling down this problem to the correctness of isolated components. For the logical decision of whether a decision follows from a policy, we offer a reliable logical *verification*: we feed the decision and policy, together with a logical representation of all documents and which signatures have been verified with respect to which keys, into the automatic theorem prover RP_X [17,16,18,19] to check that the given decision logically follows from the policy and the given documents. This is a double check by a very special "extra pair of eyes": the correctness of RP_X was formally proven using the theorem prover Isabelle. When a positive policy decision of the ATV is verified by RP_X, we believe, it is virtually impossible that it could be due to a flawed implementation of TPL's semantics.

In TPL's design we focused on three key qualities. The first is *modularity* in order to support arbitrary attribute-value based data formats. To connect a new data format to the ATV infrastructure, one only needs to write a parser from the concrete format to an abstract syntax representation. Thus there is no need to "adjust the world" – our system easily fits with existing schemes.

The second quality is *declarativity and expressive power*. TPL is inspired by Prolog (without the cut operator and negation) and thus policies are always formulated *positively*, i.e., under which conditions the policy is fulfilled. Nonetheless, TPL is Turing complete, i.e., every computable policy can be expressed; this programming aspect, in particular, allows generating templates for the most common kinds of policies. LIGHT^{est} has also produced graphical interfaces to TPL for users with different degrees of experience with policy specification [13,22].

The third quality is *formal precision and accountability*. Since we expect to deal with transactions of substantial value, it is crucial that there are no undefined corner cases or bugs in the implementation. It should be possible for an independent third party to easily review a decision. An example of such a review is the mentioned verification with RP_X. Another example could be the review in case of a legal dispute.

Parts of this paper are adapted from our technical report [12].

2 TPL by Example

We present TPL using the example of an online platform for auction houses. We do not consider peer-to-peer auction houses like eBay, but focus on platforms connecting traditional auction houses to the digital world. The auctions in question may easily range up to thousands of Euros for a single item, leading to the problem of ensuring that the successful bidder indeed pays the sum they have bid. The auction house wants no entrance barrier for new customers who just "stumbled" upon an item by an Internet search. On the other hand, they want

to avoid manipulations such as shill bidding (somebody anonymously bids on an item to push the price) and payment defaults.

In the analog world, the solution is that one has to bring references from other auction houses or a bank statement or be present at the auction, proving one's identity. We show how to transfer these aspects to the digital world using LIGHT^{est} in a way where one can benefit from the digital world's potential without losing the security and trust guarantees of analog auction houses.

The first step of digitalization is the creation of online catalogs, where a user can click on items they want to bid on and enter a maximum amount. This is basically an electronic version of the classical paper bidding form. After filling out this form, the user sends it as an HTTPS transaction to the auction house.

This paper introduces a number of example policies defining which forms the auction house accepts. One example policy in natural-language is the following:

Example Policy Rule 1 *The auction house accepts any form which is of the "Auction house 2019" format and contains a bid up to* 100 *Euro.*

As the first example of a TPL policy, let us consider how to write the above mentioned Example Policy Rule 1 in TPL:

Policy Rule Specification 1

```
accept(Form) :−
  extract(Form, format, theAuctionHouse2019Format),
  extract(Form, bid, Bid),
  Bid <= 100.
```

In this example, the variable Form is the transaction, here a bidding form in question in some concrete data format. extract is a predicate that can extract the attributes from the form: this is the interface to the parser for the respective data format. The first extraction generally is the check for the expected type of format, here the format used by the concrete auction house, identified by the constant theAuctionHouse2019Format. Next, we extract the bid field, which is bound to the variable Bid, and finally, we check that the value is below 100.

Semantically, the policy can be seen in two ways: (1) As a formal specification in first-order logic of Example Policy Rule 1. (2) As a program that can check if a form lives up to Example Policy Rule 1. In the following section, we will go in to detail with forms, formats and these semantics.

3 Syntax & Semantics

3.1 Formal Definition

The 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

```
TPLPolicy ::= Clause*

    Query ::= (Predication,)* Predication.

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

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

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

Fig. 1. Syntax in TPL specified by a grammar.

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 1.

3.2 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., $\mathsf{p(X, Y)} :- \mathsf{q(X)}, \mathsf{r(Y, X)}$ becomes $\forall \mathsf{X, Y}.\ \mathsf{p(X, Y)} \leftarrow \mathsf{q(X)} \wedge \mathsf{r(Y, X)}$.

Special care must be taken for built-in predicates, i.e. the interface to the environment, in particular, such as extract 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]

[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

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. The semantics of Prolog can be described as an interpreter – see e.g. Deransart, Ed-Dbali and Cervoni's textbook [5], 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.

3.3 Forms and Formats

Policies work on forms represented by a variety of concrete data formats, from X.509 certificates and DNS resource records 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, ...)}
```

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.

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>
```

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.

3.4 Implementation

For the LIGHT^{est} project, we implemented the Automated Trust Verifier (ATV), at the core of which is a TPL interpreter. The ATV is implemented in Java, using the ANTLR parser generator to implement the grammar from Figure 1.

Besides the interpreter core, the ATV implements the 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 (the form and the attribute) 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.

3.5 Built-In Predicates

This section describes the 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. 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.

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) represents a trusted scheme. Both parameters are input parameters. 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 form was properly signed using the given key.*

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*.*

In addition, our implementation comes with additional built-in predicates to support encoding of domains and concatenation of strings.

4 Using TPL

So far, our example auction house only accepts bids up to a certain number but puts no constraints on who may place a bid. For large bids the auction house needs to know who they are and that they can be trusted. This is achieved by issuing certificates to users, and publishing a list of trusted authorities who may issue certificates. Such a list is a trust list and is for example published by the European Union in the eIDAS framework. Therefore, we extend our example:

Example Policy Rule 2 *The auction house accepts any bid up to 1500 Euro, if it is signed by an eIDAS qualified signature.*

Thus, we need to perform the following checks: (1) Is the bid amount smaller than 1500 Euro? (2) Has the bidder's certificate been issued by an eIDAS qualified authority? (3) Did the bidder actually sign the bid?

Signatures and Signable Formats To verify signatures we use the built-in predicate verify_signature and signable formats: A signable format is a format for which a signature verification function is specified. For a form of the specified format and a public key, we can verify if the form is properly signed.

Trust Scheme lookups We need to verify the trust scheme membership of the bidder's issuer and thus have to obtain the associated trust list. Trust lists are discovered using a trust scheme claim which is inside the bidder's or issuer's certificate. In LIGHT^{est}this claim is represented by a domain name [21], e.g. the (fictional) URL qualified.trust.ec.eu for the trust scheme of qualified eIDAS authorities.

The trustlist built-in predicate (see Section 3.5) triggers a server lookup. It will succeed if a certain trust scheme exists, the trust list is available, and the desired certificate is on that list. It fails otherwise. It, therefore, acts as a requirement in a policy that the given certificate is on the claimed trust list.

To claim a trust scheme membership, a certificate includes a field trustScheme that states the trust scheme (represented as a domain) it claims to be in. In order to ensure that the domain actually belongs to our desired trust scheme, we use the built-in predicate trustscheme (see Sec.3.5).

Specifying the policy We translate Example Policy Rule 2 into a TPL rule:

Policy Rule Specification 2

```
accept(Form) :−
  extract(Form, format, theAuctionHouse2019format),
  extract(Form, bid, Bid), Bid <= 1500,
  extract(Form, certificate, Certificate),
  extract(Certificate, pubKey, PK),
  verify_signature(Form, PK),
```

```
  check_eIDAS_qualified(Certificate).

check_eIDAS_qualified(Certificate) :−
  extract(Certificate, format, eIDAS_qualified_certificate),
  extract(Certificate, issuer, IssuerCertificate),
  extract(IssuerCertificate, trustScheme, TrustSchemeClaim),

  trustscheme(TrustSchemeClaim, eIDAS_qualified),
  trustlist(TrustSchemeClaim, IssuerCertificate, TrustListEntry),

  extract(TrustListEntry, pubKey, PkIss),
  verify_signature(Certificate, PkIss).
```

When this is added to a TPL specification containing Policy Rule Specification 1, then any form that lives up to the requirements of either rule is accepted. Policy Rule Specification 2 requires that the format of the form is the auction house format, and extracts the bid to check that it is at most 1500. After that it extracts the bidder's certificate. This a form, and the policy extracts the public key of the bearer, given in the pubKey field. Then the verification of the signature of the form is done with respect to the public key using the verify_signature predicate. Afterward, the policy checks that the certificate is eIDAS qualified. This is done in a separate predicate. From the bidder's certificate, it extracts the issuer's certificate, given in IssuerCertificate. From the IssuerCertificate it then extracts the TrustSchemeClaim, which is a domain name used to address the trust scheme and to verify the issuer's trust scheme claim. The policy checks that the trust membership claim is really eIDAS qualified. This is done using the trustscheme predicate. A lookup is then done using the trustlist predicate, which discovers and retrieves the trust list and verifies that the IssuerCertificate is on the list. Lastly, the issuer's public key is extracted from the trust list entry and then used with verify_signature to verify the signature on the bidder's certificate. The TrustListEntry must contain at least the public key of the issuer, such that it can be verified to be the same as the issuer key recorded in the certificate.

This shows that policies can be specified on an abstract level avoiding specifying the whole interaction with the Internet and the checks that need to be performed on the response to authenticate it.

4.1 Allowing Trust Translation

Trust schemes can define translations, i.e. they might consider other schemes equivalent to them. We extend our example policy accordingly:

Example Policy Rule 3 *The auction house accepts any bid of at most 1000 Euro with a signature from a scheme outside eIDAS if the scheme is deemed equivalent to eIDAS via a translation scheme of eIDAS.*

We introduce the notation of equivalence modulo a translation relying on the trust translation schemes provided by the authority of the target scheme.

The used example policy is similar to Policy 2, but the trustscheme predicate is changed to trustschemeX which allows trust translation and is defined explicitly in TPL: trustschemeX checks that a trust scheme membership claim belongs either directly to the scheme we are trusting, or belongs to an equivalent scheme:

```
trustschemeX(Claim, TrustedScheme) :−
  trustscheme(Claim, TrustedScheme).

trustschemeX(Claim, TrustedScheme) :−
  encodeX(Claim, TrustedScheme, Domain),
  lookup(Domain, Entry),
  extract(Entry, translation, equivalent).
```

For a claim for a foreign scheme and the name of a trusted scheme, the built-in predicate encodeX generates a domain for the trust translation scheme. Suppose Claim is a (hypothetical) Swiss scheme located at example.admin.ch and the TrustedScheme is eIDAS_qualified. Then the URL should point to e.g. admin.ch._translation.qualified.trust.ec.eu (i.e. it should escape the domain of the original scheme, and select the corresponding Translation scheme of eIDAS qualified). This domain should refer to the entry about the Swiss scheme at eIDAS. The entry is then used to discover information which can be used to verify equivalence. In the example case, we check if the translation field is set to equivalent.

4.2 Delegation

An important concept is *delegation*: A *mandator* can *delegate* rights to a *proxy*, who then acts on behalf of the mandator. This allows us to extend the auction house service even further:

Example Policy Rule 4 *The auction house accepts any bid of at most 1000 with a signature from a proxy. The proxy must be within the eIDAS trust scheme.*

Within the delegation we have several fields where the mandator can define what the proxy is allowed to do [20]. In this case, the mandator must allow the proxy to place bids. Put in practice, the mandator could also set a maximum amount up to which the proxy is allowed to place bids. Thus the fields must be verified in order to place bids. Further, for the public key, it is checked that it is within the eIDAS trust scheme. Lastly, the policy checks at the delegation provider that the delegation is still valid and that nobody tampered with the delegation. This leads us to the specification of the delegation in Policy Rule Specification 3.

Policy Rule Specification 3

```
checkQualifiedDelegation(Document, Mandate) :−
        checkMandate(Document, Mandate),
        checkMandatorKey(Document, Mandate),
        checkValidDelegation(Document, Mandate),
```

```
        extract(Document, bid, Bid), Bid <= 1000.

checkMandate(Document, Mandate) :−
        extract(Mandate, format, delegation),
        extract(Mandate, proxyKey, PkSig),
        verify_signature(Document, PkSig),
        extract(Mandate, purpose, place_bid).

checkMandatorKey(Document, Mandate) :−
        extract(Mandate, issuer, MandatorCert),
        extract(MandatorCert, trustScheme, TrustSchemeClaim),
        trustscheme(TrustSchemeClaim, eIDAS_qualified),
        trustlist(TrustSchemeClaim, MandatorCert, TrustListEntry),
        extract(TrustListEntry, pubKey, PkIss),
        verify_signature(MandatorCert, PkIss).

checkValidDelegation(Document, Mandate) :−
        extract(Mandate, delegationProvider, DP),
        lookup(DP, DPEntry),
        extract(DPEntry, fingerprint, HMandate),
        verify_hash(Mandate, HMandate).
```

5 Verification

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 [17,16,18,19] which is, with exception of its parser, verified in Isabelle/HOL [14]. 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 interpreter can 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 $p \wedge (\neg q_q \vee \cdots \vee \neg q_n) \wedge b$ encoded in the input format of RP_x.
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 prove $p \wedge b \models q_1 \wedge ... \wedge q_n$. This is equivalent to proving that $p \wedge (\neg q_q \vee \cdots \vee \neg q_n) \wedge b$ 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 [15] 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 $p \wedge (\neg q_q \vee \cdots \vee \neg q_n) \wedge b$ in ${RP_x}'s$ input format is still left unverified and so is the parser of RP_x. Notice also that the verification of RP_x's soundness and completeness is only with respect to unsatisfiability in Herbrand models. This is not a problem though since it implies its soundness and completeness with respect to arbitrary models, but this has yet to be formally proven for RP_x.

6 Discussion and Related work

Blaze et al. [4] coined the term trust management system and introduced Policy Maker, one of the first such systems. PolicyMaker was refined to create KeyNote [3,2]. The relation between access control policies and trust policies was early recognized. Herzberg et al. [8] sees trust policy languages as an extension of access control mechanisms, thereby, as Li et al. [10] point out, generalizing authorization. Due to the similarity, a popular idea used in access control languages is often used for trust policy languages, namely logic programming.

For a large number of works, including ours, policies are always formulated positively: every policy rule describes under which conditions one is trusted and the decision is negative when no policy rule is fulfilled. This makes it a lot simpler than languages including negative rules such as Dong and Dulay's [6] Shinren: While it is convenient to also formulate negative constraints, the integration into the reasoning process results in a rather complicated semantics with a nine-valued logic and requires policy rules to be annotated with priorities. We believe that it is enough to limit the use of negation to black listing, i.e. checking that an entity is not on a black list which can be part of a server lookup with a built-in predicate. Note also that pure Horn clauses are Turing complete, i.e. every computable trust policy decision can be expressed in TPL.

Several of these languages borrow from logics of knowledge and belief such as Li et al.'s DL [10], Becker et al.'s SecPAL [1] as well as Gurevich and Neeman's DKAL [7]. In particular, they contain a modal operator, *says*, so that the fact that an agent stated a formula is itself a formula. This allows for easily relating the reasoning of participants but leaves the area of classical logic due to modal interpretation in different worlds. Basing a policy language on logic also allows us to achieve proof certification, i.e. checking that a policy decision indeed follows

logically from a policy, reducing the chance of a false policy decision due to an implementation error. Jim [9] used this idea to allow for a simple proof checker to check the policy decisions made by a more complicated program. Jim's unverified proof checker's trustworthiness came from simplicity rather than being verified.

7 Conclusion

We have presented TPL, a trust policy language and trust management system. We have shown the language's syntax and semantics as well as the idea of using formats to represent transactions abstractly. We also showed how the language supports signatures, translation and delegation. By basing the semantics on first-order logic, we have also achieved a way to verify policy decisions, by way of a prover that is formalized sound and complete in the Isabelle proof assistant.

We argued that TPL has the qualities of "modularity", "declarativity and expressive power" and "formal precision and accountability". By being modular TPL allows for heterogeneity. By providing declarativity and expressive power, TPL ensures that it has the expressibility needed to write the policies needed by users. By providing formal precision and accountability, TPL ensures that businesses and organizations can feel safe about the correctness of the automatic trust decisions. TPL is a central component of LIGHT^{est} and with the above qualities we believe that TPL can help LIGHT^{est} achieve its goal of providing a lightweight infrastructure for global heterogeneous trust management in support of an open ecosystem of stakeholders and trust schemes. We hope that this will provide a step towards wider adoption of TPL and trust management systems.

Acknowledgement. Andreas Viktor Hess suggested many improvements.

References

1. Becker, M.Y., Fournet, C., Gordon, A.D.: SecPAL: Design and semantics of a decentralized authorization language. Journal of Computer Security **18**(4), 619–665 (2010)
2. Blaze, M., Feigenbaum, J., Ioannidis, J., Keromytis, A.D.: The keynote trust-management system version 2 (1999)
3. Blaze, M., Feigenbaum, J., Keromytis, A.D.: Keynote: Trust management for public-key infrastructures (position paper). In: Security Protocols, 6th International Workshop, Cambridge, UK, April 15-17, 1998, Proceedings. pp. 59–63 (1998)
4. Blaze, M., Feigenbaum, J., Lacy, J.: Decentralized trust management. In: 1996 IEEE Symposium on Security and Privacy, May 6-8, 1996, Oakland, CA, USA. pp. 164–173 (1996)
5. Deransart, P., Ed-Dbali, A., Cervoni, L.: Prolog - The Standard: Reference Manual. Springer (1996)
6. Dong, C., Dulay, N.: Shinren: Non-monotonic trust management for distributed systems. In: Trust Management IV - 4th IFIP WG 11.11 International Conference, IFIPTM 2010, Morioka, Japan, June 16-18, 2010. Proceedings. pp. 125–140 (2010)

7. Gurevich, Y., Neeman, I.: DKAL: Distributed-Knowledge Authorization Language. In: Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, USA, 23-25 June 2008. pp. 149–162 (2008)
8. Herzberg, A., Mass, Y., Mihaeli, J., Naor, D., Ravid, Y.: Access control meets public key infrastructure, or: Assigning roles to strangers. In: 2000 IEEE Symposium on Security and Privacy, Berkeley, California, USA, May 14-17, 2000. pp. 2–14 (2000)
9. Jim, T.: SD3: A trust management system with certified evaluation. In: 2001 IEEE Symposium on Security and Privacy, Oakland, California, USA May 14-16, 2001. pp. 106–115 (2001)
10. Li, N., Feigenbaum, J., Grosof, B.N.: A logic-based knowledge representation for authorization with delegation. In: Proceedings of the 12th IEEE Computer Security Foundations Workshop, CSFW 1999, Mordano, Italy, June 28-30, 1999. pp. 162–174 (1999)
11. Mödersheim, S., Katsoris, G.: A sound abstraction of the parsing problem. In: IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014. pp. 259–273 (2014)
12. Mödersheim, S., Schlichtkrull, A.: The LIGHTest foundation. Tech. rep., Technical University of Denmark (2018)
13. Mödersheim, S.A., Ni, B.: GTPL: A Graphical Trust Policy Language. In: Open Identity Summit 2019 (2019)
14. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science, vol. 2283. Springer (2002)
15. Paulson, L.C.: Computational logic: its origins and applications. Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences **474**(2210), 20170872 (2018)
16. Schlichtkrull, A., Blanchette, J.C., Traytel, D.: 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
17. Schlichtkrull, A., Blanchette, J.C., Traytel, D.: 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. pp. 152–165 (2019)
18. Schlichtkrull, A., Blanchette, J.C., Traytel, D., Waldmann, U.: Formalization of Bachmair and Ganzinger's ordered resolution prover. Archive of Formal Proofs **2018** (2018), `https://www.isa-afp.org/entries/Ordered_Resolution_Prover.html`
19. Schlichtkrull, A., Blanchette, J.C., Traytel, D., Waldmann, U.: 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
20. Wagner, G., Omolola, O., More, S.: Harmonizing Delegation Data Formats. In: Open Identity Summit 2017. pp. 25–34. Gesellschaft für Informatik, Bonn (2017)
21. Wagner, G., Wagner, S., More, S., Hoffmann, M.: DNS-based Trust Scheme Publication and Discovery. In: Open Identity Summit 2019. pp. 49–58. Gesellschaft für Informatik, Bonn (2019)
22. Weinhardt, S., Omolola, O.: Usability of policy authoring tools: A layered approach. In: International Conference on Information Systems Security and Privacy (2019)