Core TuLiP logic programming for trust management

Citation for published version (APA):

Czenko, M., & Etalle, S. (2007). Core TuLiP logic programming for trust management. In V. Dahl, & I. Niemelä (Eds.), *Proceedings of the 23rd International Conference on Logic Programming (ICLP 2007) 8-13 September 2007, Porto, Portugal* (pp. 380-394). (Lecture Notes in Computer Science; Vol. 4670). Springer.
https://doi.org/10.1007/978-3-540-74610-2_26

DOI:

[10.1007/978-3-540-74610-2_26](https://doi.org/10.1007/978-3-540-74610-2_26)

Document status and date:

Published: 01/01/2007

Document Version:

Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

[Link to publication](#)

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy

If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Core TuLiP

Logic Programming for Trust Management*

Marcin Czenko¹ and Sandro Etalle^{1,2}

¹ Department of Computer Science
University of Twente, The Netherlands

² University of Trento, Italy
{marcin.czenko,sandro.etalles}@utwente.nl

Abstract. We propose CoreTuLiP - the core of a trust management language based on Logic Programming. CoreTuLiP is based on a subset of modal logic programming, but enjoys the features of TM languages such as RT; in particular clauses are issued by different authorities and stored in a distributed manner. We present a lookup and inference algorithm which we prove to be correct and complete w.r.t. the declarative semantics. CoreTuLiP enjoys uniform syntax and the well-established semantics and is expressive enough to model scenarios which are hard to deal with in RT.

1 Introduction

Trust management (TM) [6,8,13,15] is an approach to access control in decentralised distributed systems where access control decisions are based on policy statements issued by multiple principals, and stored in a distributed manner. Policy statements are often digitally signed to ensure their authenticity and integrity; such statements are sometimes called *credentials* or *certificates*, and – in many cases – can be represented as datalog clauses. Indeed, like in logic programming, in TM credentials often have to be combined together to provide an authorisation *proof* (e.g. a proof that a given user has indeed access to a given resource).

One of the features of TM advocated in e.g. [16] (w.r.t. classical decentralised access control, but also w.r.t. logic programming) is distributed storage: in practice credentials may or may not be stored by the authority who *issues* them; therefore one of the prominent problems of TM is that of guaranteeing that – under reasonable circumstances – if there exists a proof of a certain (authorisation) statement, then it is also possible to *find* the credentials needed to construct the proof itself.

To date, one of the most successful TM systems is the RT family, defined by Li, Winsborough and Mitchell [15,16]. This family of languages enjoys a well-defined LP-based declarative semantics, a syntax similar to that of SDSI [8], and offers the possibility of storing credentials either by the *issuer* (the authority issuing them) and/or by the *subject* (the entity the credential “refers to”). The location where the credential is stored is determined by the so-called *type* of the credential. Li et al. show that if all

* This work was supported by the projects: Freeband I-Share, EU-NoE-ARTIST2, and EU-IST-IP-SERENITY (contract N 27587). Permanent address of both authors: University of Twente.

credentials are *well-typed* then there exists a terminating credential chain discovery algorithm which inquires whether a given statement is valid in the present state.

Although the RT family is successful in achieving its goals, we believe that it presents drawbacks which are worth investigating and improving. In particular, RT syntax is inflexible to the extent that to accommodate natural things such as separation of duty etc., one has to resort to a number of rather artificial extensions (RT₁ until RT^D, and RT^T), which are difficult to grasp and use. Secondly, it cannot be linked naturally with external languages. Finally, while it enjoys a declarative reading, this reading does not reflect the crucial type information.

One could speculate that to solve these problems one should simply translate RT into Logic Programming, and then use the latter to specify and prove authorisation statements. This is however inaccurate, as this translation would lose one of the essential elements that make RT a *trust management* language, in particular the information concerning where credentials should be stored and how they can be found when needed.

In this paper we present *CoreTuLiP*, which is the stripped-down version of the TuLiP (Trust management system based on Logic Programming) system we are developing at the University of Twente in the context of the I-Share project [12]. CoreTuLiP is basically a subset of (function-free) moded logic programming, with the essential additional feature that the clauses are not stored at a central authority, but are distributed across the different principals involved in the system. The mode information determines *where* a clause will be stored and a form of *well-modedness* is used to guarantee that, as the computation progresses, enough information is available to *find* the clauses needed to build a proof of the query being evaluated. Since credentials are distributed, CoreTuLiP is not amenable to SLD resolution, and requires a mix of top-down and bottom up reasoning. Here, we present a terminating algorithm which is able to answer well-moded queries, together with the soundness and completeness result. Finally, we show that RT₀, the core language of the RT family, is basically equivalent to a subset of CoreTuLiP. Doing so, we prove that it is possible to define a true trust management language which is as expressive as RT₀ also in terms of credential distribution without giving up the established LP formalism.

CoreTuLiP, is also based on LP, but has a more flexible underlying syntax than RT, and can easily accommodate extensions. For instance, it allows one to express thresholds and separation of duties, which require special additions to RT₀.

This paper is structured as follows: in Sect. 2 we introduce the basics of moded Logic Programming. In Sect. 3 we introduce CoreTuLiP. In Sect. 4 we present the Lookup and Inference Algorithm, and we show that it is sound and complete w.r.t. the standard LP semantics. In Sect. 5 we compare RT₀ with CoreTuLiP. Finally, in Sect. 6 we present the related work and then we conclude the paper and propose future research in Sect. 7.

2 Preliminaries on Logic Programs

The reader is assumed to be familiar with the terminology and the basic results of the semantics of logic programs [1,17]. Here, we refer to *function-free* (Datalog-like) logic programs and we adopt the notation of [1]. We denote atoms by $A, B, H, \dots$, queries by $A, B, C, \dots$ (following [1], queries are simply conjunctions of atoms, possibly empty), clauses by $c, d, \dots$, and programs by P . The empty query is denoted by $\square$. For any

syntactic object (e.g. atom, clause, query) o , we denote by $Var(o)$ the set of variables occurring in o . Given a *substitution* $\sigma = \{x_1/t_1, \dots, x_n/t_n\}$ we say that $\{x_1, \dots, x_n\}$ is its *domain* (denoted by $Dom(\sigma)$) and that $Var(\{t_1, \dots, t_n\})$ is its *range* (denoted by $Ran(\sigma)$). Further, we denote by $Var(\sigma) = Dom(\sigma) \cup Ran(\sigma)$. If $t_1, \dots, t_n$ is a permutation of $x_1, \dots, x_n$ then we say that σ is a *renaming*. The *composition* of substitutions is denoted by juxtaposition ($\theta\sigma(X) = \sigma(\theta(X))$). We say that a syntactic object (e.g. an atom) o is an *instance* of o' iff for some σ , $o = o'\sigma$; o is called a *variant* of o' , written $o \approx o'$ iff o and o' are instances of each other. A substitution θ is a *unifier* of objects o and o' iff $o\theta = o'\theta$. We denote by $mgu(o, o')$ any *most general unifier* (*mgu*, in short) of o and o' . Computations are sequences of derivation steps. The non-empty query $q : \mathbf{A}, \mathbf{B}, \mathbf{C}$ and a clause $c : H \leftarrow \mathbf{B}$ (renamed apart w.r.t. q) yield the resolvent $(\mathbf{A}, \mathbf{B}, \mathbf{C})\theta$, provided that $\theta = mgu(B, H)$. A *derivation step* is denoted by $\mathbf{A}, \mathbf{B}, \mathbf{C} \xRightarrow{\theta}_{P,c} (\mathbf{A}, \mathbf{B}, \mathbf{C})\theta$. c is called its *input clause*, and B is called the *selected atom* of q . A derivation is obtained by iterating derivation steps. A maximal sequence $\delta := \mathbf{B}_0 \xRightarrow{\theta_1}_{P,c_1} \mathbf{B}_1 \xRightarrow{\theta_2}_{P,c_2} \dots \mathbf{B}_n \xRightarrow{\theta_{n+1}}_{P,c_{n+1}} \mathbf{B}_{n+1} \dots$ of derivation steps is called an *SLD derivation of $P \cup \{\mathbf{B}_0\}$* provided that for every step the standardisation apart condition holds, i.e. the input clause employed at each step is variable disjoint from the initial query $\mathbf{B}_0$, and from the substitutions and the input clauses used at earlier steps. If δ is maximal and ends with the empty query ($\mathbf{B}_n = \square$) then the restriction of θ to the variables of $\mathbf{B}$ is called its *computed answer substitution* (*c.a.s.*, for short).

Moded Programs. Informally speaking, a *mode* indicates how the arguments of a relation should be used, i.e. which are the input and which are the output positions of each atom, and allows one to derive properties such as absence of run-time errors for Prolog built-ins and absence of floundering for programs with negation [3]. Most compilers encourage the user to specify a mode declaration.

Definition 1 (Mode). Consider an n -ary predicate symbol p . By a *mode* for p we mean a function m_p from $\{1, \dots, n\}$ to $\{In, Out\}$.

If $m_p(i) = In$ (resp. *Out*), we say that i is an *input* (resp. *output*) position of p (with respect to m_p). We assume that each predicate symbol has a *unique mode* associated to it; multiple modes may be obtained by simply renaming the predicates. We use the notation $(X_1, \dots, X_n)$ to indicate the mode m in which $m(i) = X_i$. For instance, (In, Out) indicates the mode in which the first (resp. second) position is an input (resp. output) position. To benefit from the advantage of modes, programs are required to be *well-moded* [3]: they have to respect some correctness conditions relating the input arguments to the output ones. We denote by $In(A)$ (resp. $Out(A)$) the sequence of terms filling in the input (resp. output) positions of A , and by $VarIn(A)$ (resp. $VarOut(A)$) the set of variables occupying the input (resp. output) positions of A .

Definition 2 (Well-Moded). A clause $H \leftarrow B_1, \dots, B_n$ is well-moded if $\forall i \in [1, n]$

$$\begin{aligned} VarIn(B_i) &\subseteq \bigcup_{j=1}^{i-1} VarOut(B_j) \cup VarIn(H), \text{ and} \\ VarOut(H) &\subseteq \bigcup_{j=1}^n VarOut(B_j) \cup VarIn(H). \end{aligned}$$

A query $\mathbf{A}$ is well-moded iff the clause $H \leftarrow \mathbf{A}$ is well-moded, where H is any (dummy) atom of zero arity. A program is well-moded if all of its clauses are well-moded.

Note that the first atom of a well-moded query is ground in its input positions and a variant of a well-moded clause is well-moded. The following Lemma, due to [2], shows the “persistence” of the notion of well-modedness.

Lemma 1. *An SLD-resolvent of a well-moded query and a well-moded clause that is variable-disjoint with it, is well-moded.* $\square$

3 Core TuLiP

We now introduce CoreTuLiP, which in first approximation is a variant of moded LP. In CoreTuLiP, there are two disjoint types of predicates: (user-defined) *credential predicates* and built-in *constraint predicates*. In CoreTuLiP,

- credential predicates have arity two;
- in an atom with a credential predicate, we call the term filling in the first argument position the *issuer*, and the one filling in the second argument position the *recipient*.
- a *credential* is a clause defining credential predicate. Then, the *issuer* of a credential is the term filling in the first argument position of the head.

(the relations with the notions of issuer and subject used in RT are discussed in Sect. 5). In the full version we are going to have credential predicates with more arguments and user defined predicates as well. These additions are, however, immaterial for this paper.

Example 1. To access a project document at the University of Twente (UT) one must be either a project member and a Ph.D. student at the UT or at one of the partner universities, or be approved by two different assistant professors from the UT. John and Jeroen are assistant professors at the UT. John states that a project member from one of the partner universities can access the document if she is approved by at least one project member who is also an associate professor at that university. Jeroen approves anyone who is also approved by a project leader at the UT. Sandro is a project leader at the UT.

<p>(c₁) <i>access_document</i>(ut, X) :-
 <i>project_member</i>(ut, X),
 <i>prof</i>(ut, A₁),
 <i>prof</i>(ut, A₂),
 A₁ ≠ A₂,
 <i>approve_access</i>(A₁, X),
 <i>approve_access</i>(A₂, X).</p> <p>(c₂) <i>access_document</i>(ut, X) :-
 <i>phd_student</i>(P, X),
 <i>project_partner</i>(ut, P),
 <i>project_member</i>(P, X).</p> <p>(c₃) <i>approve_access</i>(john, X) :-
 <i>approve_access</i>(A, X),
 <i>associate_prof</i>(P, A),
 <i>project_partner</i>(ut, P),
 <i>project_member</i>(P, A),
 <i>project_member</i>(P, X).</p> | <p>(c₄) <i>approve_access</i>(jeroen, X) :-
 <i>approve_access</i>(L, X),
 <i>project_leader</i>(ut, L).</p> <p>(c₅) <i>project_leader</i>(ut, sandro).</p> <p>(c₆) <i>phd_student</i>(ut, marcin).</p> <p>(c₇) <i>approve_access</i>(sandro, rico).
 <i>approve_access</i>(jeffrey, rico).</p> <p>(c₈) <i>associate_prof</i>(tud, jeffrey).</p> <p>(c₉) <i>project_member</i>(ut, john).
 <i>project_member</i>(ut, charles).
 <i>prof</i>(ut, john).
 <i>prof</i>(ut, jeroen).
 <i>project_partner</i>(ut, ut).</p> <p>(c₁₀) <i>project_partner</i>(ut, tud).
 <i>project_member</i>(tud, jeffrey).
 <i>project_member</i>(tud, rico).</p> |
|--|---|

In TM, credentials are always *issued* by some authority (for the sake of simplicity here we identify authorities with the set of ground terms). In Example 1, the credential $\text{prof}(ut, john)$ is *issued* by ut (University of Twente), and has $john$ as the recipient. With this credential, ut states that $john$ is one of the professors at the University of Twente. In a practical setting, this credential is signed by ut , and ut and $john$ are placeholders for the implementation dependent identifiers (like public keys or URIs). Under these assumptions, it is natural to expect that the issuer of a credential should be a ground term.

Definition 3. Let $cl : H \leftarrow B_1, \dots, B_n$ be a clause. We say that cl is well-formed if it is well-moded and $\text{issuer}(H)$ is a ground term.

Modes and Decentralised Storage. Credential predicates have three *legal modes*: (In, In) , (In, Out) , and (Out, In) . The reason why the mode (Out, Out) is considered illegal is that it would allow queries with completely uninstantiated arguments like $\text{prof}(X, Y)$, in which neither the issuer nor the recipient is specified. Unlike in LP, such queries cannot be answered in a TM system because the system does not know where to look for relevant credentials, which could be issued and stored by any authority. By requiring that at least one of the arguments be input, and that the credentials be *traceable* (see below) we will be able to find the credentials we need to construct the proofs we need. For constraint predicates the only legal mode is the one “all-input” $(In, \dots, In)$.

A peculiar feature of trust management systems is that credentials are stored in a distributed way. For instance, in Example 1, the credential $\text{prof}(ut, john)$ which is issued by ut could be stored by either ut or $john$. Storing it by $john$ has the advantage that $john$ does not have to fetch the credential at ut every time he needs it, which in a highly distributed system may be costly. We call the *depository* of a credential the authority where the credential is stored. In CoreTuLiP, it is the mode of the credential’s head which determines its depository (here, we allow only one mode per relation symbol, so credentials will be stored at one place only; by allowing multiple modes we lift this limitation in the extended system). Returning to Example 1, if $\text{mode}(\text{prof})$ is either (In, In) or (In, Out) , then the credential $\text{prof}(ut, john)$ will be stored at ut , otherwise (if the mode is (Out, In)), $john$ will store it. Storing the credential at some other place would make it unfindable. The definition below generalises this concept.

Definition 4 (Traceable, Depository). We say that a clause $cl : H \leftarrow B_1, \dots, B_n$ is traceable if it is well-formed and one of the following conditions holds:

1. $\text{mode}(H) \in \{(In, In), (In, Out)\}$ – in this case $\text{issuer}(H)$ is the depository of the rule,
2. $\text{mode}(H) = (Out, In)$, and $\text{recipient}(H) (= In(H))$ contains a ground term – in this case $\text{recipient}(H)$ is the depository of the rule,
3. $\text{mode}(H) = (Out, In)$ and $\text{recipient}(H)$ contains a variable. In this case we require that there exists a prefix $B_1, \dots, B_k$ of the body such that
 - $\text{mode}(B_1) = \dots = \text{mode}(B_k) = (Out, In)$,
 - $In(H) = In(B_1)$,
 - $In(B_{i+1}) = Out(B_i)$, and is a variable, for $i \in [1, k-1]$,
 - $Out(B_k)$ contains a ground term,

In this case, we say that $\text{issuer}(B_k)(= \text{Out}(B_k))$ is the depositary of the rule.

The third case is complex, but it has the advantage of permitting the storage of a credential at a third party (neither the issuer, nor the recipient). Consider again Example 1, with the following mode assignments: $\text{access_document}:(In, In)$, $\text{project_member}:(In, In)$, $\text{prof}:(In, Out)$, and the remaining predicates are moded (Out, In) . In this case credentials $c_1 - c_4$ and c_9 are stored at *ut*. Credential c_5 is stored by *sandro*, c_6 by *marcin*, c_7 by *rico*, and c_8 by *jeffrey*. *tud* stores all the credentials c_{10} .

We can now introduce the concept of a state.

Definition 5. A state $\mathcal{P}$ is a finite collection of pairs (a, P_a) where P_a is a collection of traceable credentials and a is the depositary of these credentials.

The declarative semantics of a state is simply given in terms of logic programming as follows (where for simplicity we assume that all constraints are user-defined)

Definition 6. Let $\mathcal{P}$ be the state $\{(a_1, P_1), \dots, (a_n, P_n)\}$, and A be an atom

- We denote by $P(\mathcal{P})$ the set of clauses $P_1 \cup \dots \cup P_n$. We call $P(\mathcal{P})$ the LP-counterpart of state $\mathcal{P}$.
- We say that A is true in state $\mathcal{P}$ iff $P(\mathcal{P}) \cup \mathcal{C} \models A$, where $\mathcal{C}$ is a first order theory determining the meaning of credential predicates.

4 The Lookup and Inference AlgoRithm (LIAR)

The goal of an authorisation system is to check whether a fact is true in a given state. Since the state $\mathcal{P}$ can be very large and distributed across different agents, it is essential to have an algorithm which takes care of computing whether a given query is true in $\mathcal{P}$ without having to collect the entire $P(\mathcal{P})$. An extra difficulty comes from the fact that clauses might easily be mutually recursive, and that cases 2 and 3 of Definition 4 make it impossible to follow a straightforward top-down reasoning. In this section we present a suitable algorithm. Before we proceed we need the following definitions.

Definition 7 (Connected). We say that two atoms A and B that have mode (Out, In) are connected if $\text{recipient}(A)$ is ground and $\text{recipient}(A) = \text{recipient}(B)$.

Let A be an atom and S be a set of atoms. We adopt the following conventions:

- (i) We write $A \stackrel{\sim}{\in} S$ iff $\exists A' \in S$, such that $A' \approx A$ (i.e. A' is a renaming of A).
- (ii) We write $A \not\stackrel{\sim}{\in} S$ iff $\nexists A' \in S$ such that $A' \approx A$.
- (iii) We write $A \stackrel{\theta}{\hookrightarrow} S$ iff $\exists A' \stackrel{\sim}{\in} S$ standardised apart w.r.t. A such that $\gamma = \text{mgu}(A, A')$ and $A\theta \approx A\gamma$.

Definition 8. Let A be an atomic well-moded query. We define the Lookup and Inference AlgoRithm (LIAR) which given a state $\mathcal{P}$ and a query A as an input returns the (possibly empty) sets of atoms FACTSTACK and GOALSTACK. The algorithm is reported in Fig. 1.

INPUT : A . /* A is the initial atomic query */

Init:

CLSTACK : $\{A \leftarrow A\}$;

FACTSTACK = GOALSTACK = VISITED = $\emptyset$;

SATISFIED = **FALSE** ;

REPEAT

Phase 1 (Top-down resolution):

CHOOSE:

$c : H \leftarrow B, C, D \in \text{CLSTACK}$ and

$B' \subseteq \text{FACTSTACK}$, such that the following conditions hold:

(i) B and B' unify with mgu θ ,

(ii) $C\theta$ is well-moded,

(iii) $C\theta \notin \text{GOALSTACK}$,

(iv) **IF** $\text{mode}(C) = (Out, In)$ **THEN** $\text{recipient}(C\theta) \notin \text{VISITED}$ **ENDIF**

ADD $C\theta$ to GOALSTACK;

IF $\text{mode}(C) \in \{(In, Out), (In, In)\}$ **THEN**

FETCH at $\text{issuer}(C\theta)$ all clauses $\{c_1, \dots, c_n\}$ whose head unifies
with $C\theta$ with mgus $\{\gamma_1, \dots, \gamma_n\}$ respectively ;

FOR EACH $c_i \gamma_i \in \{c_1 \gamma_1, \dots, c_n \gamma_n\}$ **DO**

IF $c_i \gamma_i \notin \text{CLSTACK}$ **THEN** ADD $c_i \gamma_i$ to CLSTACK **ENDIF**

END FOR EACH

ELSEIF $\text{mode}(C) = (Out, In)$ **THEN**

FETCH all clauses $\{c_1, \dots, c_n\}$ stored at $\text{recipient}(C\theta)$ whose head
has mode (Out, In) ;

ADD $\text{recipient}(C\theta)$ to VISITED;

FOR EACH $c_i \in \{c_1, \dots, c_n\}$ **DO**

IF $c_i \notin \text{CLSTACK}$ **THEN** ADD c_i to CLSTACK **ENDIF**

END FOR EACH

ENDIF

Phase 2 (Bottom-up model-building):

REPEAT

CHOOSE: $H \leftarrow B \in \text{CLSTACK}$ and $B' \subseteq \text{FACTSTACK}$,
such that B and B' unify with mgu θ ;

IF $H\theta \notin \text{FACTSTACK}$ **THEN** ADD $H\theta$ to FACTSTACK **ENDIF**;

IF $\text{mode}(H) = (Out, In)$ AND $\text{issuer}(H\theta) \notin \text{VISITED}$ **THEN**

ADD to CLSTACK the clause:

$\text{dummy}(X, \text{issuer}(H\theta)) \leftarrow \text{dummy}(X, \text{issuer}(H\theta))$

ENDIF

UNTIL nothing can be added to FACTSTACK;

IF A is ground and $A \in \text{FACTSTACK}$ **THEN** SATISFIED = **TRUE** **ENDIF**

UNTIL SATISFIED **OR** nothing can be added to FACTSTACK and CLSTACK;

OUTPUT = FACTSTACK;

Fig. 1. The Lookup and Inference AlgoRithm (LIAR). We assume that *dummy* is a reserved predicate symbol, with mode (Out, In) . Statements in boxes are optional and included only for optimisation purposes.

The algorithm maintains three *stacks*: CLSTACK contains the set of clauses collected so far, FACTSTACK contains the set of atomic logical consequences inferred from CLSTACK, and GOALSTACK contains the set of atomic goals already processed (to handle loops). Additionally, the VISITED stack contains the set of entities that have been visited during the processing. Initially, CLSTACK contains a single clause constructed from the initial atomic query A ; the other stacks are empty. The algorithm is divided in two phases. Phase 1 contains the credential discovery. First, it selects a *new* well-moded atom $C\theta$ from the body of a clause in CLSTACK and then, depending on its mode, it fetches the *new* credentials from either $issuer(C\theta)$ or $recipient(C\theta)$. The fetched credentials are then added to the CLSTACK. Notice that, when $mode(C) = (Out, In)$, *all* clauses whose head has mode (Out, In) must be fetched from $recipient(C\theta)$, and not only the clauses whose head unifies with $C\theta$. This is because in this case one does not know which credentials may be needed to prove $C\theta$, yet. To overcome this problem, the algorithm overestimates and fetches all credentials with the right mode being stored at $recipient(C\theta)$. In Phase 2, the model of the set of clauses in the CLSTACK is build bottom-up. Newly inferred facts are added to the FACTSTACK. For the facts having mode (Out, In) , the algorithm adds a *dummy* clause to CLSTACK, so that the “subject traceable” chains can be discovered properly. The algorithm extends naturally to queries containing more than one atom. The following results show that LIAR algorithm is sound and complete w.r.t. the standard LP semantics, i.e. the centralised algorithm based on the SLD resolution. All proofs are reported in [9]. We need the following lemma.

Lemma 2. *Let $\mathcal{P}$ be a state and FACTSTACK be the result of the algorithm execution for some well-moded query. Let A be an atom in FACTSTACK. Then A is ground.*

The soundness result is rather straightforward.

Theorem 1 (soundness). *Let $\mathcal{P}$ be a state and FACTSTACK be the result of executing LIAR on $\mathcal{P}$ and a well-moded query. Then $\forall A \in \text{FACTSTACK}, P(\mathcal{P}) \models A$.*

Proof. It is easy to see that, by construction, if an atom A is added to FACTSTACK, then $\text{CLSTACK} \models A$. Since $\forall c \in \text{CLSTACK}$ c is an instance of a clause $c' \in P(\mathcal{P})$, it follows that $P(\mathcal{P}) \models A$. $\square$

The following completeness result guarantees among other things that – after executing LIAR on a state $\mathcal{P}$ and some well-moded query – for any goal $A \in \text{GOALSTACK}$ it holds that if there exists a successful SLD derivation of A in $P(\mathcal{P})$ with c.a.s. θ then $A \xrightarrow{\theta} \text{FACTSTACK}$.

Theorem 2 (completeness). *Let $\mathcal{P}$ be a state and then FACTSTACK, GOALSTACK be the result of executing LIAR on $\mathcal{P}$ and a given well-moded goal.*

Then $\forall C \in \text{GOALSTACK}$, if $\exists$ a successful SLD derivation $\delta : C \xrightarrow{\theta}_{P(\mathcal{P})} \square$ then $C \xrightarrow{\theta} \text{FACTSTACK}$.

5 CoreTuLiP vs. RT₀

In this section we are going to compare CoreTuLiP with the well-established RT₀ trust management language. We are going to show that – in most respects – CoreTuLiP is

at least as expressive as RT_0 . To this end, we first present a slightly simplified (yet expressively equivalent) version of RT_0 as given in [16]: A *principal* is a uniquely identified individual or process. A principal can define a *role*, which is indicated by a principal's name followed by a *role name*, separated by a dot. For instance $a.r$, and $alice.pictures$ are roles. For the sake of uniformity, we depart from [16] in using names starting with a lowercase letter (sometimes with subscripts) to indicate role names. A role denotes a set of principals – the members of the role. RT_0 allows a principal to issue four kinds of statements:

- *Simple Member*: $a.r \leftarrow d$. “ a asserts that d is a member of $a.r$.”
- *Simple Inclusion*: $a.r \leftarrow b.r_1$. “ a asserts that $a.r$ includes (all members of) $b.r_1$.”
- *Linking Inclusion*: $a.r \leftarrow a.r_1.r_2$. “ a asserts that $a.r$ includes $b.r_2$ for every b that is a member of $a.r_1$.”
- *Intersection Inclusion*: $a.r \leftarrow b_1.r_1 \cap b_2.r_2$. “ a asserts that $a.r$ includes every principal who is a member of both $b_1.r_1$ and $b_2.r_2$.”

An RT_0 *policy* (indicated by $\mathcal{S}$) is a set of RT_0 statements. Its semantics is defined by translating it into a *semantic program*, $SP(\mathcal{S})$, which is a Prolog program with only one ternary predicate m . Given an RT_0 statement c , the *semantic program* of c , $SP(c)$, is defined as follows:

$$\begin{aligned} SP(a.r \leftarrow d) &= m(a, r, d). \\ SP((a.r \leftarrow b.r_1)) &= m(a, r, X) :- m(b, r_1, X). \\ SP((a.r \leftarrow a.r_1.r_2)) &= m(a, r, X) :- m(a, r_1, Y), m(Y, r_2, X). \\ SP((a.r \leftarrow b_1.r_1 \cap b_2.r_2)) &= m(a, r, X) :- m(b_1, r_1, X), m(b_2, r_2, X). \end{aligned}$$

SP extends to the set of statements in the obvious way: $SP(\mathcal{S}) = \{SP(c) \mid c \in \mathcal{S}\}$. Intuitively, $m(a, r, d)$ indicates that d is a member of the role $a.r$. Finally, given an RT_0 policy $\mathcal{S}$, the semantics of a role $a.r$ is defined in terms of atoms entailed by the semantic program: $\llbracket a.r \rrbracket_{SP(\mathcal{S})} = \{d \mid SP(\mathcal{S}) \models m(a, r, d)\}$.

The Type System of RT_0 . To ensure traceability, RT_0 comes with a type system [16]. In the original presentation, each role name has two types: an issuer-side type and a subject-side type. Here – also for the sake of simplicity – we assume that each role has just one of the following three type values: *issuer-traces-all* (ITA), *issuer-traces-def* (ITD), and *subject-traces-all* (STA). To extend the results we present here to the full version (i.e. including all possible combinations of RT_0 types), we need to extend Core TuLiP in a straightforward way by allowing predicates with multiple modes.

Concerning storage, if a role name r is issuer-traces-all or issuer-traces-def, then principal a has to store all the credentials defining $a.r$. When a role name r is subject-traces-all then a credential of the form $a.r \leftarrow e$, must be stored by every subject of this credential. The successful discovery requires that each credential in the policy $\mathcal{S}$ be *well-typed*. For the sake of simplicity, we use the following definition of well-typed credentials (equivalent to [16]).

Definition 9. Let c be an RT_0 credential. We say that c is well-typed iff the combination of type value assignments appears as a valid entry in Table 1.

Table 1. Well-Typed RT_0 credentials

					$a.r \leftarrow a.r_1.r_2$									
					r_1	ITA			ITD			STA		
					r_2	ITA	ITD	STA	ITA	ITD	STA	ITA	ITD	STA
r	$a.r \leftarrow b.r_1$					ITA	OK							
	r_1	ITA	ITD	STA		ITD	OK	OK	OK		OK			OK
		STA				STA								OK

					$a.r \leftarrow b_1.r_1 \cap b_2.r_2$									
					r_1	ITA			ITD			STA		
					r_2	ITA	ITD	STA	ITA	ITD	STA	ITA	ITD	STA
r	ITA	OK	OK	OK		ITA	OK				OK			
	ITD	OK	OK	OK		ITD	OK	OK	OK	OK	OK	OK	OK	
	STA			OK		STA			OK		OK	OK	OK	

For example, take the credential $c : a.r \leftarrow a.r_1.r_2$ and assume first that $type(r)$ and $type(r_1)$ is ITD, and $type(r_2)$ is STA. Then, after checking with Table 1, we see that c is well-typed w.r.t. this type value assignment. On the other hand, if $type(r) = type(r_1) = type(r_2) = ITD$, then c is not well-typed as there is no valid entry for this type value assignment in Table 1. Note that simple member credentials (of the form $a.r \leftarrow b$) are always well-typed.

Three Sorts of Goals. If the set of credentials $\mathcal{S}$ is *well-typed* then there exists a terminating algorithm supporting three *sorts* of goals.

1. “given $a.r$, list all principals in $\llbracket a.r \rrbracket_{SP(\mathcal{S})}$ ”; this goal can be answered provided that r is issuer-traces-all.
2. “given $a.r$ and b , check if b is a member of $\llbracket a.r \rrbracket_{SP(\mathcal{S})}$ ”; this goal can be answered in all cases.
3. “given b , list all roles $a.r$ such that b is a member of $\llbracket a.r \rrbracket_{SP(\mathcal{S})}$ ”; this goal can be answered only partially: given b , the system is able to find all *subject traceable* roles $a.r$ (i.e. the roles $a.r$ where $type(r) = STA$) such that b is a member of $\llbracket a.r \rrbracket_{SP(\mathcal{S})}$.

5.1 Translating RT_0 into CoreTuLiP

We now demonstrate that CoreTuLiP is – in most cases – more expressive than RT_0 by showing that an arbitrary RT_0 policy can be translated in a straightforward way into an equivalent CoreTuLiP state. First, we define a mapping T from RT_0 to CoreTuLiP.

Definition 10. Let c be an RT_0 credential. Then $T(c)$ is defined as follows:

$$\begin{aligned}
 T(a.r \leftarrow d) &= r(a, d). \\
 T(a.r \leftarrow b.r_1) &= r(a, X) :- r_1(b, X). \\
 T(a.r \leftarrow a.r_1.r_2) &= \begin{cases} r(a, X) :- r_2(Y, X), r_1(a, Y). & \text{if } type(r_1) \neq ITA, \\ r(a, X) :- r_1(a, Y), r_2(Y, X). & \text{otherwise.} \end{cases} \\
 T(a.r \leftarrow b_1.r_1 \cap b_2.r_2) &= \begin{cases} r(a, X) :- r_2(b_2, X), r_1(b_1, X). & \text{if } type(r_1) \neq ITA, \\ r(a, X) :- r_1(b_1, X), r_2(b_2, X). & \text{otherwise.} \end{cases}
 \end{aligned}$$

Concerning the mode of the predicates, we have: if the RT_0 type of r is *ITA* then the mode of r in $T(c)$ is (In, Out) , if the RT_0 type of r is *ITD* then the mode of r in $T(c)$ is (In, In) , and if the RT_0 type of r is *STA* then the mode of r in $T(c)$ is (Out, In) . $\square$

The following theorem shows that, from the view point of the declarative semantics, $\mathcal{S}$ and $T(\mathcal{S})$ are equivalent (recall that m is the fixed predicate symbol used in $SP(\mathcal{S})$). The proof can be found in [9].

Theorem 3. *Let $\mathcal{S}$ be an RT_0 policy. Then $SP(\mathcal{S}) \models m(a, r, d)$ iff $T(\mathcal{S}) \models r(a, d)$.*

This shows that each RT_0 policy can be translated into a declaratively equivalent CoreTuLiP state. Now, to prove the full equivalence we still have to prove two things, namely that (a) if an RT_0 credential is stored at principal a then its corresponding CoreTuLiP statement is stored at a as well, and that (b) the CoreTuLiP system can answer the same goals the RT_0 system can. We start with the following proposition proven in [9].

Proposition 1. *Let c be an RT_0 credential.*

- (a) *If c is stored at a then $T(c)$ is also stored at a .*
- (b) *If c is a well-typed then $T(c)$ is traceable.*

At last, we have to show how RT_0 goals can be transformed into (legal, i.e. well-moded) CoreTuLiP queries. Since RT does not have a formal notation to express goals we have to be a bit verbose.

Remark 1 (Translating RT_0 goals). Let $\mathcal{S}$ be a well-typed RT_0 policy and $T(\mathcal{S})$ its CoreTuLiP equivalent. Let us consider the different sorts of goals supported by RT_0 .

Sort 1: the general goal of this sort is “given $a.r$, list all principals in $\llbracket a.r \rrbracket_{SP(\mathcal{S})}$ ”. This is translated into the query $r(a, X)$. This goal can be answered in RT_0 only if the role r has type *ITA*. But in this case the mode of r in $T(\mathcal{S})$ is (In, Out) , and the query $r(a, X)$ is well-moded w.r.t. it. Therefore, we can conclude that goals of sort 1 can be safely expressed in CoreTuLiP.

Sort 2: the general goal of this sort is “given $a.r$ and b , check if b is a member of $\llbracket a.r \rrbracket_{SP(\mathcal{S})}$ ”. This is translated into the query $r(a, b)$, which being ground is always well-moded. Therefore, we can conclude that goals of sort 2 can be safely expressed in CoreTuLiP.

Sort 3: the general goal of this sort is “given b , list all $a.r$ such that b is a member of $\llbracket a.r \rrbracket_{SP(\mathcal{S})}$ ”. Such goals have no corresponding CoreTuLiP translation. The technical reason behind this limitation is purely of syntactic nature: the translation would be higher-order query $(X(Y, b))$, where X and Y are variables). There are two reasons why we believe that this limitation of CoreTuLiP w.r.t. RT_0 is hardly relevant in practice: first, RT_0 allows to express the query, but it is not able to give a complete answer in any case: it can only find all such $a.r$ which are also *subject traceable*. Secondly (also because RT_0 is not able to provide a full answer), this kind of goal is not used in practice on their own, but only as a subgoal of the goals of Sort 2. $\square$

The syntactic inability of CoreTuLiP to express goals of Sort 3 is actually a conscious design choice we made to keep the syntax manageable (to express goals of this sort we

would need a “polymorphic” mode system in which the actual mode of an atom does not only depend on its predicate symbol but also on some of its arguments). Actually, our LIAR algorithm would be able to answer such queries as well.

Summarising, Theorem 3, Proposition 1, and Remark 1 allow us to say that Core-TuLiP is at least as expressive as RT_0 , with the small exception of the goals of Sort 3. In proving this, we have made the restrictive assumption that an RT_0 role name has just one of the following three types: ITA (issuer-traces-all), ITD (issuer-traces-def), or STA (subject-traces-all). The extension to the full version (i.e. including all possible combinations of RT_0 types) can be done in a straightforward way by extending Core TuLiP so that it allows predicates with multiple modes.

5.2 A Flexible Syntax

As we said already, CoreTuLiP is simply the core language of the TM system we are developing. The full language will allow credentials with more than two arguments and user defined predicates. Nevertheless, CoreTuLiP is already expressive enough to express complex policies (like thresholds or separation of duty) that in RT require the adoption of special operators (which are present in more expressive members of the RT family RT_1 , RT_2 , RT^T , or RT^D). Consider for instance the following statement taken from [15] “*a* says that an entity is a member of *a.r* if one member of *a.r*₁ and two *different* members of *a.r*₂ all say so”. This policy cannot be expressed in RT_0 , and to express this in RT one needs to use the so-called *manifold roles*, which extend the notion of roles by allowing role members to be *collections* of entities (rather than just principals). This is done in RT^T by defining the operators $\odot$ and $\otimes$. A *type-5* credential of the form $a.r \leftarrow b_1.r_1 \odot b_2.r_2$ says that $\{s_1 \cup s_2\}$ is a member of *a.r* if *s*₁ is a member of *b*₁.*r*₁ and *s*₂ is a member of *b*₂.*r*₂. A *type-6* credential $a.r \leftarrow b_1.r_1 \otimes b_2.r_2$ has a similar meaning, but it additionally requires that $s_1 \cap s_2 = \emptyset$. With these two additional types, one can express the above statement using the following three credentials:

$$\begin{aligned} a.r &\leftarrow a.r_4.r \\ a.r_4 &\leftarrow a.r_1 \odot a.r_3 \\ a.r_3 &\leftarrow a.r_2 \otimes a.r_2 \end{aligned}$$

In CoreTuLiP, on the other hand, this policy can be expressed quite naturally with the following one line:

$$r(a, X) :- r_1(a, Y), r(Y, X), r_2(a, Z_1), r_2(a, Z_2), Z_1 \neq Z_2, r(Z_1, X), r(Z_2, X).$$

Notably, to express this, we don’t have to use manifold-like structures which are, in our opinion, rather hard to grasp.

6 Related Work

The term “trust management” was first coined by Blaze, Feigenbaum, and Lacy [7] as an answer to the inadequacy of traditional authorisation mechanisms based on identity-based public-key systems like X.509. X.509 strongly depends on a centralised hierarchy

of certificate authorities (CA), which significantly restricts the possible application area in the distributed heterogeneous systems. In their work, Blaze et al. identify various requirements that a trust management system should satisfy and they introduce *PolicyMaker* - a trust management engine fulfilling these requirements. A direct descendant of PolicyMaker is KeyNote [5]. The reader can find a comparison of PolicyMaker and KeyNote in [6].

Clark et al. developed SDSI/SPKI [8] that combines local namespaces and the corresponding *name certificates* of SDSI [20] with *authorisation certificates* of SPKI [10]. They present a TM language that they prove to be expressive enough for many access control scenarios. All these TM systems avoid the problem of *credential storage*. In all these approaches it is assumed that all required credentials can be found when needed and as such they are not well-prepared to face the reality of the today's highly distributed world. In [21] Winsborough and Li identify the features a "good" language for credentials should have, one of those being the support for distributed storage. The RT family of Trust Management Languages [16,15] is the first in which the problem of credential discovery is given an extensive treatment. In particular, in [16], a type system is introduced in order to restrict the number of possible credential storage options. In Sect. 5 we compare the type system of RT_0 with our approach to the credential discovery problem.

Li, Groszof, and Feigenbaum in [14] developed a logic-based language, called Delegation Logic (DL), to represent policies, credentials, and requests for distributed authorisations. Monotonic version of Delegation Logic – called DILP – is based on Logic Programming language Datalog. DILP extends Datalog with constructs that feature delegation depth and a wide variety of complex principals.

Bertino et al. [4] introduce an XML-based trust negotiation language $\mathcal{X}$ -TNL used for expressing credentials and disclosure policies in the *Trust- $\mathcal{X}$* system. Though Trust- $\mathcal{X}$ policies are formalised using logic rules, it also does not directly support credential discovery.

The well-known *eXtensible Access Control Markup Language* (XACML) [19] supports the distributed policies and also provides a profile for role based access control (RBAC). However, in XACML, it is the responsibility of the *Policy Decision Point* (PDP) – an entity handling access requests – to know where to look for the missing attribute values in the request.

7 Conclusions

In this paper we introduce CoreTuLiP, a true Trust Management Language which enjoys the advantages of LP syntax and of its declarative semantics. CoreTuLiP forms the basis for the TuLiP TM language we are developing at the UT (which will include user-defined predicates and will enjoy most features of moded logic programs, including interface with other languages, debugging facilities, etc.). The main purpose of CoreTuLiP is to provide a theoretical basis for the further developments.

CoreTuLiP enjoys the advantages of trust management languages: for instance, statements may be issued by multiple authorities and be stored by authorities different from the issuing one. To deal with the problem of finding the credentials which are needed for

a proof, we define the notion of traceable credentials and present a Lookup and Inference AlgoRithm (LIAR), which we show to be correct and complete w.r.t. the standard declarative semantics. We also compare CoreTuLiP with RT_0 and show that each RT_0 credential and goal translates in a straightforward way into Core TuLiP (with the small exception of the goals of Sort 3).

The theoretical relevance of this paper is that we show that it is possible to define a true TM language without leaving the well-established LP formalism. The practical relevance lies in the much greater flexibility, extendibility and accessibility that LP language enjoys with respect to – for instance – RT. As we have discussed, to accommodate various needs, the language RT_0 has developed a relatively large number of extensions, which are in our opinion often hard to grasp. We thought that this was the price we had to pay to have a true TM language, but CoreTuLiP shows that this can be done otherwise.

Future Work. CoreTuLiP can be extended in several directions. First we plan to investigate extending CoreTuLiP to support non-stratified negation. This is connected to our previous work on $RT_{\ominus}$ [18], where we extend RT_0 with *negation-in-context*. Secondly, we are going to add support for integrity constraints for TM systems [11]. We also plan to provide an implementation for CoreTuLiP, possibly supporting the XACML standard.

Acknowledgements. We would like to thank Pieter Hartel and Jeroen Doumen from the University of Twente for their feedback and valuable comments to this paper.

References

1. Apt, K.R.: From Logic Programming to Prolog. Prentice-Hall, Englewood Cliffs (1997)
2. Apt, K.R., Luitjes, I.: Verification of Logic Programs with Delay Declarations. In: Alagar, V.S., Nivat, M. (eds.) AMAST 1995. LNCS, vol. 936, pp. 66–90. Springer, Heidelberg (1995)
3. Apt, K.R., Marchiori, E.: Reasoning about Prolog programs: from Modes through Types to Assertions. Formal Aspects of Computing 6(6A), 743–765 (1994)
4. Bertino, E., Ferrari, E., Squicciarini, A.C.: Trust- χ : A Peer-to-Peer Framework for Trust Establishment. IEEE Trans. Knowl. Data Eng. 16(7), 827–842 (2004)
5. Blaze, M., Feigenbaum, J., Ioannidis, J., Keromytis, A.: The KeyNote Trust-Management System, Version 2. IETF RFC 2704 (September 1999)
6. Blaze, M., Feigenbaum, J., Ioannidis, J., Keromytis, A.: The Role of Trust Management in Distributed Systems Security. In: Vitek, J., Jensen, C. (eds.) Secure Internet Programming. LNCS, vol. 1603, pp. 185–210. Springer, Heidelberg (1999)
7. Blaze, M., Feigenbaum, J., Lacy, J.: Decentralized Trust Management. In: Proc. of the 17th IEEE Symposium on Security and Privacy, pp. 164–173. IEEE Computer Society Press, Los Alamitos (1996)
8. Clarke, D., Elien, J.E., Ellison, C., Fredette, M., Morcos, A., Rivest, R.L.: Certificate Chain Discovery in SPKI/SDSI. Journal of Computer Security 9(4), 285–322 (2001)
9. Czenko, M.R., Etalle, S.: Core TuLiP. Technical Report TR-CTIT-07-22, Centre for Telematics and Information Technology. University of Twente, Enschede (March 2007)
10. Ellison, C., Frantz, B., Lampson, B., Rivest, R., Thomas, B., Ylonen, T.: SPKI Certificate Theory. IETF RFC 2693 (September 1999)

11. Etalle, S., Winsborough, W.H.: Integrity Constraints in Trust Management – Extended Abstract. In: Ahn, G.-J. (ed.) Proc. 10th ACM Symp. on Access Control Models and Technologies (SACMAT), pp. 1–10. ACM Press, New York (2005), Extended version available at CoRR: <http://arxiv.org/abs/cs.CR/0503061>
12. Freeband Communication. I-Share: Sharing Resources in Virtual Communities for Storage, Communications, and Processing of Multimedia Data. URL: <http://www.freeband.nl/project.cfm?language=en&id=520>
13. Jim, T.: SD3: A Trust Management System with Certified Evaluation. In: Proceedings of the 2001 IEEE Symposium on Security and Privacy, pp. 106–115. IEEE Computer Society Press, Los Alamitos (2001)
14. Li, N., Grosz, B., Feigenbaum, J.: Delegation Logic: A Logic-based Approach to Distributed Authorization. ACM Transactions on Information and System Security (TISSEC) 6(1), 128–171 (2003)
15. Li, N., Mitchell, J., Winsborough, W.: Design of A Role-based Trust-management Framework. In: Proc. 2002 IEEE Symposium on Security and Privacy, pp. 114–130. IEEE Computer Society Press, Los Alamitos (2002)
16. Li, N., Winsborough, W., Mitchell, J.: Distributed Credential Chain Discovery in Trust Management. Journal of Computer Security 11(1), 35–86 (2003)
17. Lloyd, J.W.: Foundations of Logic Programming, 2nd edn. Springer, Heidelberg (1993)
18. Czenko, M., Tran, H., Doumen, J., Etalle, S., Hartel, P., den Hartog, J.: Nonmonotonic Trust Management for P2P Applications. In: Proc. of the 1st International Workshop on Security and Trust Management. Electronic Notes in Theoretical Computer Science, vol. 157, pp. 113–130. Elsevier, Amsterdam (2006)
19. OASIS.: eXtensible Access Control Markup Language (XACML) Version 2.0 (February 2005), URL: <http://www.oasis.org>
20. Rivest, R., Lampson, B.: SDSI - A Simple Distributed Security Infrastructure (October 1996), Available at <http://theory.lcs.mit.edu/~rivest/sdsi11.html>
21. Winsborough, W.H., Li, N.: Towards Practical Automated Trust Negotiation. In: POLICY, pp. 92–103. IEEE Computer Society Press, Los Alamitos (2002)