Datalog with Constraints: A Foundation for Trust Management Languages

Ninghui Li and John C. Mitchell

Department of Computer Science, Stanford University
Gates 4B, Stanford, CA 94305-9045
{ninghui.li, jcm}@cs.stanford.edu

Abstract. Trust management (TM) is a promising approach for authorization and access control in distributed systems, based on signed distributed policy statements expressed in a policy language. Although several TM languages are semantically equivalent to subsets of Datalog, Datalog is not sufficiently expressive for fine-grained control of structured resources. We define the class of linearly decomposable unary constraint domains, prove that Datalog extended with constraints in any combination of such constraint domains is tractable, and show that permissions associated with structured resources fall into this class. We also present a concrete declarative TM language, RT_1^C, based on constraint Datalog, and use constraint Datalog to analyze another TM system, KeyNote, which turns out to be less expressive than RT_1^C in significant respects, yet less tractable in the worst case. Although constraint Datalog has been studied in the context of constraint databases, TM applications involve different kinds of constraint domains and have different computational complexity requirements.

1 Introduction

One main goal of computer security is to ensure that access to resources is restricted to parties with legitimate access permissions. Traditional *access control* mechanisms process requests from authenticated users of an operating system or a database system and make authorization decisions based on the identity of the requester. However, in decentralized, open, distributed systems, the resource owner and the requester often are unknown to one another, and access control based on identity may be ineffective. In the "trust-management" approach to distributed authorization, articulated in [4], access control decisions are based on *policy statements* made by multiple principals. Some statements are digitally signed to ensure their authenticity and integrity; these are called *credentials*. Some statements may be stored in local trusted storage and do not need to signed, we call these *access rules*. In a TM scenario, a *requester* submits a request, possibly supported by a set of credentials issued (signed) by other parties, to an *authorizer*, who specifies access rules governing access to the requested resources. The authorizer then decides whether to authorize this request by answering the *proof-of-compliance* question: "Do the access rules and credentials authorize the request?" Digitally signed credentials document authenticated attributes of entities. These attributes may be group membership, membership in a role within an organization, or being delegated of a permission or role. Access rules can specify what attributes are required to access a resource and other conditions of access, such as time or auditing requirements. There are good reasons to prefer TM languages that are declarative and have a formal foundation.

Several TM languages are based on DATALOG, e.g., Delegation Logic [14, 13], the RT (Role-based Trust-management) framework [15, 16], SD3 (Secure Dynamically Distributed DATALOG) [10], and Binder [6]. However, DATALOG has limitations as a foundation of TM languages. One significant limitation is the inability to describe structured resources. For example, a project manager may want to grant permission to read the entire document tree under a given URI, assign responsibility for associating public keys with all DNS names in a given domain, restrict network connections to port numbers in a limited range, or approve routine transactions with value below an upper limit. The permission to access all files and subdirectories under a directory "/pub/rt" represents permissions to access a potentially infinite set of resources that seems most naturally expressed using a logic programming language with function symbols. However, the tractability of DATALOG is a direct consequence of the absence of function symbols. Previous TM languages that can express certain structured resources, e.g., SPKI [7], have not had a formal foundation; some studies suggest that SPKI may be ambiguously specified and intractable [8, 1].

In this paper, we show that DATALOG extended with constraints (denoted by DATALOG$^{\mathcal{C}}$) can define access permissions over structured resources without compromising the properties of DATALOG that make it attractive for trust management, thus establishing a suitable logical foundation for a wider class of TM languages. DATALOG$^{\mathcal{C}}$ allows first-order formulas in one or more constraint domains, which may define file hierarchies, time intervals, and so on, to be used in the body of a rule, thus representing access permissions over structured resources in a declarative language. We study several constraint domains that are useful for representing structured resources, e.g., tree domains and range domains, and show that DATALOG$^{\mathcal{C}}$ with these domains can be evaluated efficiently. We also define a general class of tractable constraint domains, called linearly decomposable unary constraint domains and present a concrete declarative TM language, RT_1^C, that is based on DATALOG$^{\mathcal{C}}$. We show how to translate credentials in RT_1^C, which extends the DATALOG-based RT_1 language from the RT framework [15, 16] with constraints, into DATALOG$^{\mathcal{C}}$ over tractable constraint domains. We also use DATALOG$^{\mathcal{C}}$ to analyze another prominent TM systems KeyNote [3], and show that KeyNote uses constraint domains that are too expressive.

Constraint DATALOG has been studied extensively in the Constraint Database (CDB) literature [11, 12, 18–21]. However, TM applications involve constraint domains that are outside the scope of previous CDB research. Moreover, TM applications have different computational complexity requirements. In the CDB literature, tractability is often measured using data complexity, which considers the processing time for a fixed query (set of rules) as the size of the database (set of facts) grows. Data complexity is appropriate for CDB applications where the size of the input databases dominates the size of the queries by several orders of magnitude. However, expressing the access control policy in trust management and distributed access control requires both rules and facts. In particular, delegation, a characteristic feature of trust management, is represented using rules rather than facts. To guarantee that queries can be answered in time related to the complexity of the access control policy, TM applications require efficient computation as a function of the size of the set of rules and facts.

The rest of this paper is organized as follows. Some background on constraint Datalog appears in Section 2. Tractability of constraint domains is studied in Section 3, with RT_1^C in Section 4 and an analysis of KeyNote in Section 5. We conclude in Section 6.

2 Background on Constraint Datalog

Constraint Datalog is a restricted form of Constraint Logic Programming (CLP) [9], and is also a class of query languages for CDB.

2.1 Datalog

Datalog is a restricted form of logic programming with variables, predicates, and constants, but without function symbols. A Datalog *rule* has the form

$$R_0(t_{0,1},\ldots,t_{0,k_0}) :- \; R_1(t_{1,1},\ldots,t_{1,k_1}),\ \ldots\ ,\ R_n(t_{n,1},\ldots,t_{n,k_n})$$

where $R_0,\ldots,R_n$ are predicate (relation) symbols and each term $t_{i,j}$ is either a constant or a variable ($0 \le i \le n$ and $1 \le j \le k_i$). The formula $R_0(t_{0,1},\ldots,t_{0,k_0})$ is called the *head* of the rule and the sequence $R_1(t_{1,1},\ldots,t_{1,k_1}),\ \ldots\ ,\ R_n(t_{n,1},\ldots,t_{n,k_n})$ the *body*. If $n = 0$, then the body is empty and the rule is called a *fact*. A rule is *safe* if all variables occurring in the head also appear in the body. A Datalog *program* is a finite set of Datalog rules. Datalog is attractive for trust management because of the following reasons.

1. Datalog is declarative and is a subset of first-order logic; therefore, the semantics of a Datalog-based TM language is declarative, unambiguous, and widely understood.
2. Datalog has been extensively studied both in logic programming, and in the context of relational databases as a query language that supports recursion. TM languages based on Datalog can benefit from past results and future advancements in those fields.
3. The function-symbol-free property of Datalog ensures its tractability. For a safe Datalog program with fixed maximum number of variables per rule, construction of its minimal model takes time polynomial in the size of the program.
4. There are efficient goal-directed evaluation procedures for answering queries.

2.2 Constraint Domains and Constraint Databases

The notion of constraint databases was introduced in [11], and grew out of the research on DATALOG and CLP. It generalizes the relational model of data by allowing infinite relations that are finitely representable using constraints. Constraint databases find many applications in spatial and temporal databases. For recent surveys, see [12, 18].

Intuitively, a constraint domain is a domain of objects, such as numbers, points in the plane, or files in a file hierarchy, together with a language for speaking about these objects. The language is typically defined by a set of first-order constants, function symbols, and relation symbols.

Definition 1. A *constraint domain* Φ is a 3-tuple $(\Sigma, \mathcal{D}, \mathcal{L})$. Here Σ is a signature; it consists of a set of constants and a collection of predicate and function symbols, each with an associated "arity", indicating the number of arguments to the symbol. $\mathcal{D}$ is a Σ-structure; it consists of the following: a set D called the universe of the structure, a mapping from each constant to an element in D, a mapping from each predicate symbol in Σ of degree k to a k-ary relation over D, and a mapping from each function symbol in Σ of degree k to a function from D^k into D. $\mathcal{L}$ is a class of quantifier-free first-order formulas over Σ, called the *primitive constraints* of this domain.

Following common conventions, we assume that the binary predicate symbol $=$ is contained in Σ and is interpreted as identity in $\mathcal{D}$. We also assume that $\top$ (true) and $\perp$ (false) are in $\mathcal{L}$, and that $\mathcal{L}$ is closed under variable renaming.

The following are examples of classes of constraint domains that have been studied in the CDB literature; they are listed in order of increasing expressive power.

Equality constraint domains The signature Σ consists of a set of constants and one predicate $=$. A primitive constraint has the form $x = y$ or $x = c$, where x and y are variables, and c is a constant. DATALOG can be viewed as one specific instance of DATALOG$^{\mathcal{C}}$ with an equality constraint domain.

Order constraint domains The signature Σ has two predicates: $=$ and $<$. The Σ-structure is linearly ordered. A primitive constraint has the form $x\theta y$, $x\theta c$, or $c\theta x$ where θ is one of $=, <$.

Order and inequality constraint domains The signature Σ has predicates $\{=, \neq, <, >, \geq, \leq\}$. The Σ-structure is linearly ordered. A primitive constraint has the form $x\theta y$ or $x\theta c$, where θ is any predicate in Σ.

The structures in order constraint domains and order and inequality constraint domains can be integers, rational numbers, real numbers, or some subset of them.

Linear constraint domains The signature Σ has function symbols $+$ and $*$ and predicates $\{=, \neq, <, >, \geq, \leq\}$. A primitive constraint has the form $c_1x_1 + \cdots + c_kx_k\theta b$, where c_i is a constant and x_i is a variable for each $1 \leq i \leq k$, θ is any predicate in Σ, and b is a constant.

Polynomial constraint domains The signature Σ has the same functions symbols and predicate symbols as linear constraint domains. A primitive constraint has the form $p(x_1, \ldots, x_k)\theta 0$, where p is a polynomial in variables $x_1, \ldots, x_k$, and θ is any predicate in Σ.
Linear constraints and polynomial constraints may be interpreted over integers, rational numbers, or real numbers.

Definition 2. Let Φ be a constraint domain.

1. A *constraint k-tuple*, or a *constraint*, (in variables $x_1, \ldots, x_k$) is a finite conjunction $\phi_1 \wedge \cdots \wedge \phi_N$, where each $\phi_i, 1 \leq i \leq N$, is a primitive constraint in Φ. Furthermore, the variables in each ϕ_i are all free and among $x_1, \ldots, x_k$.
2. A *constraint relation of arity k* is a finite set $r = \{\psi_1, \ldots, \psi_M\}$, where each $\psi_i, 1 \leq i \leq M$ is a constraint k-tuple over the same variables $x_1, \ldots, x_k$.
3. The *formula corresponding to* the constraint relation r is the disjunction $\psi_1 \vee \cdots \vee \psi_M$.
4. A *constraint database* is a finite collection of constraint relations.

Relational calculus, relation algebra, and DATALOG can all be enhanced with constraints as query languages for constraint databases. Our focus in this paper is DATALOG extended with constraints, DATALOG$^{\mathcal{C}}$.

2.3 Evaluation of Datalog$^{\mathcal{C}}$

A *constraint* (DATALOG) *rule* has the form:

$$R_0(x_{0,1}, \ldots, x_{0,k_0}) :- R_1(x_{1,1}, \ldots, x_{1,k_1}), \ldots, R_n(x_{n,1}, \ldots, x_{n,k_n}), \psi_0$$

where ψ_0 is a constraint in the set of all variables in the rule. When $n = 0$, the constraint rule is called a *constraint fact*. A constraint rule with n hypotheses may be applied to n constraint facts to produce m facts. The process of applying a rule to a set of facts requires a form of quantifier elimination, made precise in the following two definitions.

Definition 3. Given a rule of the form above and n facts of the form

$$R_i(x_{i,1}, \ldots, x_{i,k_i}) :- \psi_i(x_{i,1}, \ldots, x_{i,k_i})$$

where each ψ_i is a constraint, $1 \leq i \leq n$, a *constraint rule application* produces $m \geq 0$ facts

$$R_0(x_1, \ldots, x_k) :- \psi'_j(x_1, \ldots, x_k),$$

where each ψ'_j is a constraint, $1 \leq j \leq m$, and $\psi'_1(x_1, \ldots, x_k) \vee \cdots \vee \psi'_m(x_1, \ldots, x_k)$, or $\perp$ when $m = 0$, is equivalent to the formula

$$\exists * (\psi_1(x_{1,1}, \ldots, x_{1,k_1}) \wedge \cdots \wedge \psi_n(x_{n,1}, \ldots, x_{n,k_n}) \wedge \psi_0),$$

where "$*$" is the list of the variables that appear in the body but not the head of the rule.

Intuitively, a rule means that the head of the rule holds if the body holds, where variables that appear only in the body are implicitly existentially quantified. Therefore, the head of the rule holds if the displayed $\exists*$ formula is true. When the $\exists*$ formula is equivalent to a disjunction $\psi'_1(x_1,\ldots,x_k) \vee \cdots \vee \psi'_m(x_1,\ldots,x_k)$, then the rule reduces to a set of facts (rules with only constraints in the body).

The form of constraint rule application defined above is called *closed-form* because the outputs $\psi'_1,\ldots,\psi'_m$ are constraints in the same constraint domain as the input facts. Closed-form application requires quantifier elimination.

Definition 4. Let $x_1,\ldots,x_k$ be a set of variables, $* \subseteq \{x_1,\ldots,x_k\}$ some subset, and $\overline{*} = \{x_1,\ldots,x_k\} - *$ its complement. A constraint domain $(\Sigma,\mathcal{D},\mathcal{L})$ *admits quantifier elimination* if, for every formula $\exists*\psi(x_1,\ldots,x_k)$ with ψ any constraint (a conjunction of several constraints is still a constraint), it is possible to compute an equivalent quantifier-free disjunction of constraints $\psi'_1(\overline{*}) \vee \cdots \vee \psi'_m(\overline{*})$ with the same free variables.

Linear constraint domains (and other less expressive domains) admit quantifier elimination. On the other hand, the domain of polynomial constraints over integers does not admit quantifier elimination. This follows from the fact that it is undecidable to determine whether constraints of the form $p(x_1,\ldots,x_k) = 0$, known as Diophantine equations, have integer solutions or not [17]. The domain of polynomial constraints over real numbers admits quantifier elimination, but the complexity is very high.

The least fixpoint of a DATALOG$^{\mathcal{C}}$ program over any constraint domain that admits quantifier elimination may be computed by iterated rule application. The following algorithm terminates when all derivable new facts are already implied by previous results of the algorithm.

Definition 5 (The Datalog$^{\mathcal{C}}$ least fixpoint algorithm).

```
constraintFixpoint(Facts, Rules) {
  Results = Facts;  Changed = true;
  while (Changed) {
    Changed = false;
    foreach Rule = "R0(...) :- R1(...), ..., Rk(...), ψ0" in Rules
      foreach Tuple <(R1 :- ψ1), ..., (Rk :- ψk)>
                              constructed from Results {
        NewResults = constraintRuleApplication(Rule, Tuple)
        foreach Fact in NewResults {
          if (Fact is not implied by any fact in Results) {
            Results = Results ∪ {Fact};  Changed=true; } } }
  }
  return Results;
}
```

The set of facts produced by this algorithm is called the *constraint least fixpoint* of the program. Even when a constraint domain admits quantifier elimination, the least fixpoint algorithm may not terminate. An example arises in

DATALOG$^{\mathcal{C}}$ with linear constraints over the integers, which can express any computable function. More efficient least fixpoint algorithms exist. Also, resolution-style goal-directed evaluation procedures for DATALOG can be adapted to work with DATALOG$^{\mathcal{C}}$ [20].

In the CDB literature, most complexity results are about data complexity, which is a measure of running time for a fixed query as the size of the input database grows. Some constraint domains that can be evaluated in closed-form with DATALOG with PTIME data complexity are: equality constraints, order and inequality constraints over dense linear order domains [11], and integer periodicity constraints ($x \equiv_k y$, $x \equiv_k c$) for fixed set of k's [21].

As mentioned in the introduction, a more restrictive DATALOG$^{\mathcal{C}}$ complexity measure is appropriate for TM applications.

Definition 6. A constraint domain Φ is *tractable*, if evaluating any DATALOG$^{\mathcal{C}}$ program with constraints in Φ has time complexity polynomial in the size of the program, when the size of each rule is bounded by a fixed value. One good measure of rule size is the sum of all the arities of the predicates in a rule.

3 Tractable Constraint Domains in Trust Management

In TM languages, it is useful to appeal to constraints from several domains. It is straightforward to define multi-sorted DATALOG$^{\mathcal{C}}$, following the standard definition of multi-sorted first-order logic. In order to keep each constraint domain separate from the others, we assume that when constraint domains are combined, each domain is given a separate sort, all predicate symbols are only applicable to arguments from the appropriate constraint domain, and each variable belongs to only one sort. It is straightforward to verify, by inspection of the algorithm in Definition 5, that any multi-sorted combination of tractable domains remains tractable.

Theorem 1. *A multi-sorted* DATALOG$^{\mathcal{C}}$ *program with constraints in several domains can be evaluated in time polynomial in the size of the program if all involved constraint domains are tractable.*

We now give several classes of constraint domains that are useful in TM.

Tree domains Each constant of a tree domain takes the form $\langle a_1, \ldots, a_k \rangle$. Imagine a tree in which every edge is labelled with a string value. The constant $\langle a_1, \ldots, a_k \rangle$ represents the node for which $a_1, \ldots, a_k$ are the strings on the path from root to this node. A primitive constraint is of the form $x = y$ or $x\theta\langle a_1, \ldots, a_k \rangle$, in which $\theta \in \{=, <, \leq, \prec, \preceq\}$, $x < \langle a_1, \ldots, a_k \rangle$ means that x is a child of the node $\langle a_1, \ldots, a_k \rangle$, and $x \prec \langle a_1, \ldots, a_k \rangle$ means that x is a descendant of $\langle a_1, \ldots, a_k \rangle$.

Range domains Range domains are syntactically sugared order domains. A primitive constraint has the form $x = y$, $x = c$ or $x \in (c_1, c_2)$, in which c is a constant, each of c_1 and c_2 is either a constant or a special symbol "$*$", meaning unbounded. And when c_1 is not $*$, "(" can also be "["; similarly, ")" can be "]" when c_2 is not $*$.

Discrete domains with sets This is syntactically sugared version of equality domains. A primitive constraint has the form $x = y$, or $x \in \{c_1, \ldots, c_\ell\}$, in which $c_1, \ldots, c_\ell$ are constants.

The following is an example that uses three sorts: one tree domain and two range domains.

Example 1. An entity A grants to an entity B the permission to connect to machines in the domain "stanford.edu" at port number 80, and allows B to further delegate any part of the permission, the validity period of this grant is from time t_1 to time t_3. To represent this, we need to use a tree domain for DNS names, a range domain for port number, and another range domain for time. The above grant and delegation can be represented using the following constraint fact and rule.

$$\begin{gathered}\text{grantConnect}(A,\, B,\, h,\, p,\, v) :-\ h \prec \langle\text{edu,stanford}\rangle,\ p = 80,\ v \in [t_1, t_3].\\ \text{grantConnect}(A,\, x,\, h,\, p,\, v) :-\ \text{grantConnect}(B,\, x,\, h,\, p,\, v),\\ h \prec \langle\text{edu,stanford}\rangle,\ p = 80,\ v \in [t_1, t_3].\end{gathered}$$

If B grants to another entity D the permission to connect to the host "cs.stanford.edu" and any machine in the domain "cs.stanford.edu" at any port number, with validity period from t_2 to t_4. Then we have:

$$\text{grantConnect}(B,\, D,\, h,\, p,\, v) :-\ h \preceq \langle\text{edu,stanford,cs}\rangle,\ v \in [t_2, t_4].$$

From the above, we can conclude the following, assuming that $t_1 \leq t_2 \leq t_3 \leq t_4$:

$$\text{grantConnect}(A,\, D,\, h,\, p,\, v) :-\ h \preceq \langle\text{edu,stanford,cs}\rangle,\ p = 80,\ v \in [t_2, t_3].$$

3.1 Hierarchical Domains Are Tractable

We first show that tree domains are tractable, using a specialized property of unary statements about tree orderings.

Definition 7. A constraint domain is *unary* if each primitive constraint either has the form $x = y$, where x and y are variables, or contains only one variable. We call a unary primitive constraint a *basic constraint*.

Definition 8. A unary constraint domain is *hierarchical* if, for any two basic constraints $\phi_1(x)$ and $\phi_2(x)$, either $\phi_1(x) \wedge \phi_2(x)$ is unsatisfiable or one of the constraints implies the other.

It is not difficult to verify that tree domains are hierarchical.

Theorem 2. *Hierarchical domains are tractable.*

Proof. Consider the algorithm in Definition 5 and the process of constraint rule application. The key step is quantifier elimination, i.e., finding a formula equivalent to $\exists * \psi(x_1, \ldots, x_k)$, in which $\psi(x_1, \ldots, x_k)$ is a conjunction of primitive constraints and $* \subseteq \{x_1, \ldots, x_k\}$. In hierarchical constraint domains, this can be done as follows. First, we transform ψ to an equivalent constraint that is free of equality constraints. For every constraint $x_i = x_j$ in ψ, we remove $x_i = x_j$ and replace every occurrence of x_j in ψ with x_i. Next, if any variable x_i has two basic constraints, by the property of hierarchical domains, either their conjunction is unsatisfiable, in which case $\exists * \psi(x_1, \ldots, x_k)$ is equivalent to $\perp$, or one of them implies the other, in which case we can remove the less restrictive one. Repeating the above step until either we know that $\exists * \psi(x_1, \ldots, x_k)$ is not satisfiable, or we have a constraint that has at most one basic constraint per variable. In the latter case, we simply remove the constraints about variables occurring in $*$ (since any one basic constraint is satisfiable) and get an constraint equivalent to $\exists * \psi(x_1, \ldots, x_k)$.

Following this process for quantifier elimination, the fixpoint computation for any hierarchical domains does not introduce any new basic constraints. If the algorithm begins with a set of constraint rules that have total size N (and fixed rule size), there are at most polynomial number of different constraint facts as possible results, giving us a computational complexity of PTIME. ∎

3.2 Linearly Decomposable Domains Are Tractable

Range domains are not hierarchical. The conjunction of two basic constraints $x \in (c_1, *)$ and $x \in (*, c_2)$ results in a new constraint $x \in (c_1, c_2)$, which is not equivalent to either.

Definition 9. A unary constraint domain is said to be *linearly decomposable* if there exists a constant d such that, given any set C of basic constraints about one variable x, there exists a set C' of basic constraints about x such that $|C'| \leq d|C|$, where $|C|$ is the sum of the sizes of constraints in C for some appropriate notion of size (e.g., number of symbols in a constraint), and the conjunction of any subset of $C \cup C'$ can be represented by the disjunction of constraints in C'. We say that C' is a decomposition of C.

Clearly, all hierarchical domains are linearly decomposable. Range domains are also linearly decomposable. For example, a set of constraints $C = \{x \in (*, 10], x \in [5, *), x \in [1, 5]\}$ can be decomposed into $C' = \{x \in (*, 1), x \in [1, 4], x \in [5, 5], x \in (5, 10], x \in (10, *)\}$. Discrete domains with sets are also linearly decomposable, as each constraint $x \in \{c_1, \ldots, c_\ell\}$ is equivalent to the disjunction of ℓ constraints $x = c_1, \cdots, x = c_\ell$. This is linear because the size of the original constraint is $\Theta(\ell)$.

Theorem 3. *Linearly decomposable domains are tractable.*

Proof. Given a $\text{D}\textsc{atalog}^{\mathcal{C}}$ program, one can collect all the basic constraints in it, rename them so that all the constraints are about the same variable, and compute a linear decomposition of them. During quantifier elimination, a conjunction of multiple constraints on one variable can be replaced with a disjunction of constraints in C'. The fixpoint computation does not need to introduce any new basic constraints beyond those in C', and the size of C' is bounded by dN. The rest follows from the proof of Theorem 2.

3.3 Not All Unary Domains Are Tractable

The key reason that linearly decomposable domains are tractable is that although new basic constraints are introduced by the conjunction of existing constraints, the number of these new constraints are still linear in the total size of the original constraints. The tractability result in Theorem 3 can be generalized to the case of polynomially decomposable domains. We now show that some unary constraint domains are not polynomially decomposable and are intractable.

Example 2. The universe of the constraint domain is all the subsets of

$$A = \{a_{11}, \cdots, a_{1n}, a_{21}, \cdots, a_{2n}, \cdots, a_{n1}, \cdots, a_{nn}\}$$

and the only predicates are $=$ and $\subseteq$. We show a program that has n^2 constraint rules and total size n^4:

$$\begin{array}{ll} \{p_1(x) :- x \subseteq A - \{a_{1i}\}. & \mid 1 \leq i \leq n\} \\ \{p_2(x) :- p_1(x),\ x \subseteq A - \{a_{2i}\}. & \mid 1 \leq i \leq n\} \\ \cdots & \\ \{p_n(x) :- p_{n-1}(x),\ x \subseteq A - \{a_{ni}\}. & \mid 1 \leq i \leq n\} \end{array}$$

The constrain least fixpoint is

$$\{p_n(x) :- x \subseteq A - \{a_{1i_1}, a_{2i_2}, \cdots, a_{ni_n}\} \mid 1 \leq i_1 \leq n, \ldots, 1 \leq i_n \leq n\},$$

which has size n^n. In this example, answering a single query is still tractable, computing the fixpoint is not.

3.4 Discussion

There are tractable constraint domains that are not unary; for example, order and inequality constraints over densely ordered structures. In this paper, we limit our attention to unary constraint domains. Unary domains are not very interesting from the point of view of constraint satisfaction. However, we find them attractive for the following reasons. First, DATALOG$^{\mathcal{C}}$ with unary domains strictly generalizes DATALOG, yet preserves the features of DATALOG that makes it attractive for trust management. Second, DATALOG$^{\mathcal{C}}$ with unary domains can express most useful assertions in trust management, because describing permissions or attributes of entities typically does not involve constraints relating two variables in ways other than equality. Third, DATALOG$^{\mathcal{C}}$ with unary domains is easier to understand and to implement than more complicated domains. Ease of understanding is an important advantage, since authors of TM policy statements need to understand their meanings.

4 RT_1^C: A Declarative TM language based on Datalog$^{\mathcal{C}}$

In this section, we introduce RT_1^C, a constraint-based extension to the RT_1 language in the RT framework [15, 16], as a concrete example of declarative TM languages based on DATALOG$^{\mathcal{C}}$. Each statement in RT_1^C can be translated into an equivalent rule in DATALOG$^{\mathcal{C}}$ with linearly decomposable domains.

4.1 Overview of the *RT* Framework

The *RT* framework is a family of Role-based Trust-management languages. The basic concepts of *RT* include entities and roles. *Entities* can issue statements and make requests. *RT* assumes that one can determine which entity issued a particular statement or request. Public/private key pairs clearly make this possible. We use A, B, and D, sometimes with subscripts, to denote entities.

A *role* in *RT* takes the form of an entity followed by a role name, separated by a dot. The simplest kinds of role names, used in RT_0, are identifiers. We use R, often with subscripts, to denote role names. A *role* is similar to a group; it defines a set of entities who are members of this role. Each entity A has the authority to define who are the members of each role of the form $A.R$, and A does so by issuing statements. Each statement defines one role to contain either an entity, another role, or certain other expressions that evaluate to a set of entities. A role may be defined by multiple statements. Their effect is union.

We now describe four kinds of statements for defining roles in RT_1; for simplicity, we assume that role names are simple identifiers.

- *Type-1*: $A.R \longleftarrow B$
  A and B are (possibly the same) entities, and R is a role name. This means that A defines B to be a member of A's R role.
- *Type-2*: $A.R \longleftarrow B.R_1$
  This statement means that A defines its R role to include (all members of) B's R_1 role.
- *Type-3*: $A.R \longleftarrow A.R_1.R_2$
  We call $A.R_1.R_2$ a *linked role*. This means that A defines its R role to include (members of) every role $B.R_2$ in which B is a member of $A.R_1$ role.
- *Type-4*: $A.R \longleftarrow A_1.R_1 \cap A_2.R_2 \cap \cdots \cap A_\ell.R_\ell$
  This means that A defines its R role to include the intersection of the ℓ roles.

Following is an example from [16], illustrating the use of these statements.

Example 3. A fictitious Web publishing service, EPub, offers a discount to anyone who is both an ACM member and a preferred customer of EOrg, the parent organization of EPub. EOrg considers students of all universities to be preferred customers, and delegate the authority over the identification of students to entities that EOrg believes are legitimate universities. EOrg additionally delegates the authority over identifying universities to a fictitious Accrediting Board for Universities, ABU. Alice is an ACM member and a student of StateU, which is accredited by ABU.

$$
\begin{array}{l}
\text{EPub.discount} \longleftarrow \text{EOrg.preferred} \cap \text{ACM.member} \\
\text{EOrg.preferred} \longleftarrow \text{EOrg.university.student} \\
\text{EOrg.university} \longleftarrow \text{ABU.accredited} \\
\text{ABU.accredited} \longleftarrow \text{StateU} \\
\text{StateU.student} \longleftarrow \text{Alice} \\
\text{ACM.member} \longleftarrow \text{Alice}
\end{array}
$$

In the above example, role names are simple identifiers. In RT_1, more generally, role names can have parameters. Parameterized roles can represent access permissions that take parameters identifying resources and access modes, role templates (e.g., leader of a project), relationships between entities (e.g., manager of an employee), and attributes that have fields (e.g., digital driver licenses, digital diplomas).

4.2 RT_1^C

RT has *application domain specification documents (ADSDs)* and statements. Each ADSD defines a vocabulary, which is a suite of related data types and role identifiers (role ids for short).

RT_1^C has several categories of types: integer types, float types, enumeration types, string types, tree types. Integer types, float types, and ordered enumeration types correspond to range domains. Unordered enumeration types and string types correspond to discrete domains with sets. And tree types correspond to tree domains. Each type category has a syntax for defining *value sets*, for each value set S, $x \in S$ corresponds to a basic constraint in the corresponding constraint domain. In an ADSD, to declare a role id, one needs to declare the parameters. Each parameter has a name and a data type.

An RT_1^C *statement* has the same structure as an RT_0 statement. The difference is that each role name takes the form of $r(h_1, \ldots, h_n)$, in which r is a role identifier, and for each i in $1..n$, h_i takes one of the following three forms: $f = c$, $f \in S$, and $f = \mathit{ref}$, in which f is the name of one of r's parameters that has type τ, c is a constant of type τ, S is a value set of type τ, and *ref* is a reference to another parameter in the same statement, also of type τ.

We now describe how to translate RT_1^C statements into DATALOG$^{\mathcal{C}}$ rules. Each type is mapped to a constraint domain, and each role id r is mapped to a corresponding predicate symbol $\overline{r}$. Role names in RT_1^C have named parameters; these can be easily translated into unnamed (position-based) parameters by choosing an order among parameters.

1. From $A.r(h_1, \ldots, h_n) \longleftarrow D$ to

   $\overline{r}(A, D, x_1, \ldots, x_k) :- \psi$

   In which k is the arity of r and ψ is a conjunction of primitive constraints corresponding to parameters $h_1, \ldots, h_n$. A parameter like $f_j = c$ is translated into a basic constraint $x = c$. A parameter like $f_i \in S$ is translated into a corresponding basic constraint. And a parameter like $f_j = \mathit{ref}$ is translated into an equality constraint involving two variables.
2. From $A.r(h_1, \ldots, h_n) \longleftarrow B.r_1(s_1, \ldots, s_m)$ to

   $\overline{r}(A, y, x_1, \ldots, x_k) :- \overline{r_1}(B, y, x_{1,1}, \ldots, x_{1,k_1}), \psi$

   In which k and k_1 are the arities of r and r_1, ψ is a constraint corresponding to the parameters $h_1, \ldots, h_n, s_1, \ldots, s_m$.
3. From $A.r(h_1, \ldots, h_n) \longleftarrow A.r_1(s_{1,1}, \ldots, s_{1,m_1}).r_2(s_{2,1}, \ldots, s_{2,m_2})$ to

   $\overline{r}(A, y, x_1, \ldots, x_k) :- \overline{r_1}(A, z, x_{1,1}, \ldots, x_{1,k_1}), \overline{r_2}(z, y, x_{2,1}, \ldots, x_{2,k_2}), \psi$

In which ψ is a constraint corresponding to the parameters in the statement.

4. From $A.r(h_1, \ldots, h_n) \longleftarrow A_1.r_1(s_{1,1}, \ldots, s_{1,m_1}) \cap \cdots \cap A_\ell.r_\ell(s_{\ell,1}, \ldots, s_{\ell,m_\ell})$ to

$$\overline{r}(A, y, x_1, \ldots, x_k) :- \overline{r_1}(A_1, y, x_{1,1}, \ldots, x_{1,k_1}), \cdots, \overline{r_\ell}(A_\ell, y, x_{\ell,1}, \ldots, x_{\ell,k_\ell}), \psi$$

In which ψ is a constraint corresponding to the parameters in the statement.

As shown in [15, 16], the *RT* framework supports for flexible delegation relationships and distributed credential chain discovery. RT_1 requires that every variable in a statement must appear in the body, to guarantee that the resulting DATALOG rule is safe. As a result, one cannot represent granting the permissions of connecting to any port number in a range to an entity. In RT_1^C, this restriction is not needed anymore. The addition of constraints enables one to represent permissions involving ranges and structured resources. Using DATALOG$^{\mathcal{C}}$ as the foundation of RT_1^C provides a sound semantics foundation and tractability guarantee.

5 Using Datalog$^{\mathcal{C}}$ to Analyze KeyNote

KeyNote [3] is a TM system that is based on PolicyMaker [4]. A KeyNote *assertion* is essentially a delegation from its issuer to its licensees, which in the simplest case is a single entity. A KeyNote *request* is characterized by a list of fields, which are name/value pairs. An assertion also has *conditions* written in an expression language, which refers to fields in requests. The intuitive meaning of an assertion is that, if the licensees support a request, and the request satisfies the conditions, then the issuer supports the request as well. KeyNote can be roughly captured by DATALOG$^{\mathcal{C}}$ with several very expressive constraint domains. One domain is integers with function symbols $\{+, -, *, /, \%, \hat{}\}$, predicates $\{=, \neq, <, >, \leq, \geq\}$, and any quantifier-free first-order formula as a primitive constraint. The fragment of that domain without function symbols $\{/, \%, \hat{}\}$ is polynomial constraints over integers, which, as we discussed in Section 2.3, does not admit quantifier elimination.

Theorem 4. *It is undecidable to compute the set of all requests that a set of KeyNote assertions authorizes.*

Note that the above theorem does not rule out that possibility to determine whether any specific request is authorized by a set of assertions. In fact, this only involves arithmetic computation and comparison. The above result means that there does not exist an algorithm to perform analysis of all the requests being authorized by a set of assertions. In fact, this is so even when there is only one assertion with a single entity as the licensees, and the question is just whether the assertion authorizes any request at all. We view this as a significant disadvantage, because it would be desirable to evaluate and analyze the effect of security assertions.

We want to point out that examples given in [3] do not use the expressive power that leads to undecidability. In fact, we have not encountered any TM example both in our research and in literature that requires such expressive power;

therefore, we argue that the expression language in KeyNote is too expressive. On the other hand, it has been shown that the delegation structure in KeyNote is too limited in TM applications [14, 15].

Related Work

Several TM languages were designed based on DATALOG without constraints. DATALOG with periodicity constraints is used in [2] in an access control language that supports periodic temporal constraints; however, this work does not deal with representation of structured resources and the general tractability of different constraint domains.

In comparison with work on constraint databases, Chomicki et al. [5] state "Recent developments in constraint databases, in particular the research on aggregation and spatiotemporal applications, suggest a need for *middle-ground* formalisms that preserve some of the expressive power of constraint databases and constraint query languages, while at the same time generalizing in a natural way the basic assumptions underlying the classical relational model of data." In [5], Chomicki et al. study constraint databases with variable independence conditions, which is a property of constraint relations. Our work to find tractable domains is also a search for useful middle-grounds, but our motivations are different, namely, usefulness in trust management, simplicity, and tractability. These motivations led us to take a different approach; we study properties of constraint domains, rather than properties of constraint relations. Moreover, properties like hierarchical and linearly decomposable are not limited to one class of constraint domains; our approach is thus similar to yet different from that in [19], in which Revesz studies the complexity of DATALOG$^{\mathcal{C}}$ with various limited form of linear constraints. We believe that DATALOG$^{\mathcal{C}}$ with unary constraint domains provides a useful middle-ground that generalizes DATALOG in a natural and useful way while preserving many nice properties of DATALOG.

6 Conclusion and Future Directions

Trust management (TM) languages need a declarative and formal foundation. Although DATALOG has been the best logical foundation for distributed access control decisions to date, DATALOG does not meet the practical need for policies about common structured resources. Our work with the RT family of TM languages [15, 16], and demonstration applications such as a distributed scheduling system and web-based file-sharing system, underscore the need for a more expressive logical foundation. DATALOG with constraints is a promising and expressive alternative that eliminates some deficiencies of DATALOG without sacrificing any of the attractive features that make DATALOG appealing for trust management.

In this paper, we identify a class of constraint domains called linearly decomposable unary domains, prove that DATALOG with any combination of such constraint domains is tractable, and show that permissions associated with structured resources, including tree domains and range domains, fall into this class.

To illustrate the value of constraint DATALOG for designing TM languages, we present a declarative TM language, RT_1^C, based on constraint DATALOG. We also use DATALOG to analyze KeyNote, which turns out to be less expressive than RT_1^C in significant respects, yet less tractable in the worst case.

Further study is needed on the tractability of unary constraint domains and non-unary constraint domains useful for trust management. We showed that linearly decomposability is a sufficient condition for tractability; however, we have not identified necessary and sufficient conditions for a unary constraint domain to be tractable. Other constraint domains worthy of investigation include strings with constraints involving regular expressions.

References

1. Olav Bandmann and Mads Dam. A note on SPKI's authorization syntax. In *Pre-Proceedings of 1st Annual PKI Research Workshop*, April 2002. Available from http://www.cs.dartmouth.edu/~pki02/.
2. Elisa Bertino, Claudio Bettini, Elena Ferrari, and Pierangela Samarati. An access control model supporting periodicity constraints and temporal reasoning. *ACM Transactions on Database Systems*, 23(3):231–285, 1998.
3. Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis. The KeyNote trust-management system, version 2. IETF RFC 2704, September 1999.
4. Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 164–173. IEEE Computer Society Press, May 1996.
5. Jan Chomicki, Dina Goldin, Gabriel Kuper, and David Toman. Variable independence in constraint databases, November 2001. In final review for IEEE Transactions on Knowledge and Data Engineering.
6. John DeTreville. Binder, a logic-based security language. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 105–113. IEEE Computer Society Press, May 2002.
7. Carl Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian Thomas, and Tatu Ylonen. SPKI certificate theory. IETF RFC 2693, September 1999.
8. Jonathan R. Howell. *Naming and sharing resources acroos administrative boundaries*. PhD thesis, Dartmouth College, May 2000.
9. Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–580, 1994.
10. Trevor Jim. SD3: A trust management system with certified evaluation. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 106–115. IEEE Computer Society Press, May 2001.
11. Paris C. Kanellakis, Gabriel M. Kuper, and Peter Z. Revesz. Constraint query languages. *Journal of Computer and System Sciences*, 51(1):26–52, August 1995. Preliminary version appeared in *Proceedings of the 9th ACM Symposium on Principles of Database Systems (PODS)*, 1990.
12. Gabriel Kuper, Leonid Libkin, and Jan Paredaens, editors. *Constraint Databases*. Springer, 2000.
13. Ninghui Li, Benjamin N. Grosof, and Joan Feigenbaum. A practically implementable and tractable Delegation Logic. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 27–42. IEEE Computer Society Press, May 2000.

14. Ninghui Li, Benjamin N. Grosof, and Joan Feigenbaum. Delegation Logic: A logic-based approach to distributed authorization. *ACM Transaction on Information and System Security (TISSEC)*, February 2003. To appear.
15. Ninghui Li, John C. Mitchell, and William H. Winsborough. Design of a role-based trust management framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 114–130. IEEE Computer Society Press, May 2002.
16. Ninghui Li, William H. Winsborough, and John C. Mitchell. Distributed credential chain discovery in trust management. To appear in *Journal of Computer Security*. Extended abstract appeared in *Proceedings of the Eighth ACM Conference on Computer and Communications Security (CCS-8)*, November 2001.
17. Yuri V. Matiyasevich. *Hilbert's Tenth Problem*. The MIT Press, 1993.
18. Peter Z. Revesz. Constraint databases: A survey. In L. Libkin and B. Thalheim, editors, *Semantics in Databases*, number 1358 in LNCS, pages 209–246. Springer, 1998.
19. Peter Z. Revesz. Safe Datalog queries with linear constraints. In *Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming (CP98)*, number 1520 in LNCS. Springer, 1998.
20. David Toman. Memoing evaluation for constraint extensions of Datalog. *Constraints: An International Journal*, 2:337–359, 1997.
21. David Toman and Jan Chomicki. Datalog with integer periodicity constraints. *Journal of Logic programming*, 35:263–290, 1994.