

# A Datalog Framework for Modeling Relationship-based Access Control Policies

Edelmira Pasarella

Universitat Politècnica de Catalunya  
Computer Science Department  
edelmira@cs.upc.edu

Jorge Lobo

Institució Catalana de Recerca i  
Estudis Avançats (ICREA)  
Universitat Pompeu Fabra  
jorge.lobo@upf.edu

## Abstract

Relationships like friendship to limit access to resources have been part of social network applications since their beginnings. Describing access control policies in terms of relationships is not particular to social networks and it arises naturally in many situations. Hence, we have recently seen several proposals formalizing different Relationship-based Access Control (ReBAC) models. In this paper, we introduce a class of Datalog programs suitable for modeling ReBAC and argue that this class of programs, that we called ReBAC Datalog policies, provides a very general framework to specify and implement ReBAC policies. To support our claim, we first formalize the merging of two recent proposals for modeling ReBAC, one based on hybrid logic and the other one based on path regular expressions. We present extensions to handle negative authorizations and temporal policies. We describe mechanism for policy analysis, and then discuss the feasibility of using Datalog-based systems as implementations.

## 1 Introduction

Lately, there has been a growing interest within the access control community in the concept of Relationship-based Access Control (ReBAC). ReBAC has been used in social networks almost since their beginnings with the well-known friendship relationship of Facebook as its prototypical example. Technical awareness of the concept was first reported

in [15], and perhaps the first formalization in the context of social networks was reported in [5]. Describing access control policies in terms of relationships is not particular to social networks. For example, a doctor can look at your medical records if he or she is *your family doctor*, or you can read a paper in a repository if you are one of *its reviewers*. At the core of the model there is a graph in which nodes represent users and resources, and arcs are labeled with relationships. Policies are described through paths among nodes in the graph (e.g., a-friend-of-a-friend represents a path of three nodes and two arcs). Recently, several papers have proposed different formalizations for ReBAC [4, 7, 8, 13, 17]. In this paper we argue that Datalog provides a very general framework for ReBAC modeling. To support our claim we work with two of the most sophisticated proposals, one based on hybrid logic and the other one based on path regular expressions, and show how complementary features of the two approaches can be captured in Datalog. The hybrid logic proposal has been developed in a series of papers that started with a modal logic as a modeling language [14], then it evolved into a model based on hybrid logic [4, 13], and more recently, an implementation embedded in the open source medical records system OpenMRS has been reported in [25]. This provides some maturity to the project. The second proposal follows the more explicit approach of defining a path specification language over the relationship graph to write policies. Results for path based ReBAC are more dispersed since more empha-

sis has been given to describing other parts of the access control systems (see for example [6, 17, 7]) and less to the formal characterizations of the expressibility. The work we have chosen for path specification, [8], is one of the most recent proposals and it incorporates features of earlier works with a more precise description of its expressibility. We then show how working under the Datalog framework we can easily extend the model (in ways that it would not be obvious to do formally in hybrid logic), we can also do policy analysis and have efficient implementations. Our contributions in this paper are the following:

1. We introduce a carefully selected subset of Datalog with equality constraints as a ReBAC policy specification language which ensures efficient implementations.
2. We then extend the hybrid logic HL of [4] to be able to express the path expressions of [8] and show a sound and complete translation of the extended HL policies into ReBAC Datalog policies.
3. We extend ReBAC Datalog policies to be able to express negative authorizations, all easily done formally because of Datalog.
4. We show how we can also use Datalog itself to find policy gaps and policy conflicts, and briefly discuss how to implement conflict resolution strategies.
5. We further extend the language to handle temporal policies.
6. We present precise complexity and expressibility results of the basic ReBAC Datalog which together with item (2) characterize the complexity of the (extended) hybrid logic for ReBAC.
7. We present evidence that policy evaluation can be done in the order of a few milliseconds using off-the-shelf Datalog engines with relationship graphs having hundred of thousands of arcs.

We end with some concluding remarks.

## 2 ReBAC Datalog policies

We are going to closely follow the terminology from the hybrid logic of [4] in our definitions, but first, we need to recall some basic notions of Datalog with constraints. For writing Datalog programs we need three disjoint (possibly infinite) sets  $C$ ,  $Var$  and  $P$  of constant symbols, variables and predicate symbols. There is a positive integer associated to each predicate symbol called its arity. A *term* in Datalog is any variable or constant symbol. An *atom* is an expression of the form  $p(t_1, \dots, t_k)$ , where  $p$  is a predicate symbol of arity  $k$  and  $t_1$  through  $t_k$  are terms. A *literal* is any atom  $p(t_1, \dots, t_k)$  or its negation  $\neg p(t_1, \dots, t_k)$ . A negated atom is called a *negative literal*; otherwise is called *positive*. If all the terms appearing in a literal/atom are constants the literal/atom is called *ground*. *Constraints* are expressions of the form  $t_1 = t_2$  or  $t_1 \neq t_2$  for any two terms  $t_1$  and  $t_2$ . Variables will be denoted using capital letters. A Datalog rule is an expression of the form:

$$c_1, \dots, c_k, L_1, \dots, L_m \rightarrow A \quad (1)$$

where the  $L_i$  are literals,  $A$  is an atom, the  $c_i$  are constraints, for  $k, m \geq 0$ . The expression  $c_1, \dots, c_k, L_1, \dots, L_m$  is called the *body* of the rule,  $A$  the *head*, and the rule a *definition* of the predicate that appears in  $A$ . An informal reading of a Datalog rule is that if there is a ground instance of the rule (i.e., all variables in the rule are replaced with constants) for which the constraints in the rule are valid, and we already know that every ground literal in the body is true then we can infer that the ground instance of  $A$  in the head of rule is true. A Datalog program is a finite set of Datalog rules.

The intended meaning of a Datalog program is given by a set of ground atoms  $M$  and is defined in terms of another set of ground atoms  $I$  given as input to the program. The set  $M$  contains the (ground) atoms in  $I$ , which are assumed to be true, plus the set of ground atoms that can be inferred to be true using the rules and the input  $I$ . Ground atoms outside  $M$  are assumed to be false. More formally, given a set of Datalog rules  $D$ , we call the set of all constants mentioned in  $D$  the *active language* of  $D$ . We denote

by  $Gr(D)$  the set of Datalog rules obtained by replacing in all possible ways the variables in the rules with constants in the active language of  $D$ . Note that  $Gr(D)$  will be empty if  $D$  does not mention any constant. Since ground atoms are also Datalog rules the same definitions of active language and  $Gr(\cdot)$  apply when we consider a Datalog program and an input. We will use for the interpretation of constraints the unique name assumption [24] in which all constants are assumed to be different from each other. Given a set of ground atoms  $M$ , and an atom  $A$ , we write  $M \models_{\text{Datalog}} A$  iff there exists a ground instance  $A'$  of  $A$  such that  $A' \in M$ . If  $A$  is ground and  $A \notin M$ , we write  $M \models_{\text{Datalog}} \neg A$ .

**Definition 2.1** *Given a Datalog program  $D$  and an input  $I$ , a set of ground atoms  $M$  is a model of  $I \cup D$  iff  $M$  is a minimal set (i.e., there is no a proper subset of  $M$ ) for which the following equation holds:*

$$M = \{A \mid c_1, \dots, c_k, L_1, \dots, L_m \rightarrow A \in Gr(I \cup D), \forall c_i : c_i \text{ is true, and } \forall L_i : M \models_{\text{Datalog}} L_i\}.$$

In general,  $I \cup D$  may have zero, one or more models. But as we will see later, policies will have a single model. In its most simplest form, a *query* to a Datalog program  $D$  with input  $I$  is to ask whether a ground atom  $A$  is true in every model of  $I \cup D$ . If this is the case we will write  $I \cup D \models_{\text{Datalog}} A$ ; we write  $I \cup D \models_{\text{Datalog}} \neg A$  if  $\neg A$  is not true in any model of  $I \cup D$ . The definition can be extended to non-ground atoms if  $I \cup D$  has a unique model  $M$ :  $I \cup D \models_{\text{Datalog}} A$  iff  $M \models_{\text{Datalog}} A$ . We can also have a conjunction of literals  $L_1, \dots, L_m$ ,  $m > 1$ , as a query and we write  $M \models_{\text{Datalog}} L'_1, \dots, L'_m$  as an answer if and only if  $L'_1, \dots, L'_m$  are ground instances of the literals  $L_1, \dots, L_m$  where variables are consistently replaced across the literals and  $\forall i M \models_{\text{Datalog}} L'_i$ .

**Protection states (see [4])** The underlying principle behind ReBAC is that from the point of view of specifying access control policies it is sufficient to have an abstract representation of the state of the system to protect built upon three fundamental concepts: the set of objects that form part of the system (e.g., users, resources), a set of properties that can be associated to individual objects, and a set of binary

relationships between these objects - a relationship graph where vertices are objects and edges are labeled with relationship names. Hence, a *protection state* in ReBAC Datalog will be described by a set of ground atoms where only two predicate symbols are used, a 3-ary predicate  $rel$  and a 2-ary predicate  $prop$ . The set of constants  $C$ , is partitioned into three disjoint sets, a set of nominal constants  $C_n$  representing names of objects, a set propositional constants  $C_p$ , representing properties, and a set of (binary) relationship names  $C_r$ . A ground atom of the form  $rel(n_1, r_1, n_2)$  can be member of a protection state only if  $n_1, n_2 \in C_n$  and  $r_1 \in C_r$ . A ground atom of the form  $prop(n_1, p_1)$  can be member of a protection state only if  $n_1 \in C_n$  and  $p_1 \in C_p$ . Intuitively speaking,  $C_n$  is the set of objects over which policies will be expressed. It contains the names of all the objects that can request access to resources, usually called *principals*, as well as the names of resources for which principals can request access to.  $C_r$  is the set of names of relationships that can be defined over these objects such as Alice is friend of Bob ( $rel(alice, friend, bob)$ : a principal-to-principal relationship), Bob owns Printer1 ( $rel(bob, own, printer_1)$ : a principal-to-resource relationship), or Alice is member of Department Alpha ( $rel(alice, member, alpha)$ : here Alpha is an abstract entity which is used only to simplify policy specifications, e.g. all members of Alpha have access to Printer1). A propositional name in  $C_p$  is meant to represent a property that a collection of objects may have, like being a medical doctor,  $prop(alice, doctor)$ , or a patient,  $prop(bob, patient)$ , or the property of being a Java program,  $prop(file.jar, java)$ , or a video file,  $prop(file.avi, video)$ .<sup>1</sup>

**Policies** Policy defines a new relation between principals and resources that *grants* the principals access to the resources. In ReBAC Datalog policies this relationship is defined by checking properties of the objects typically reachable through the relationship graph either from the principal making the request or the resource that the principal wants to access as

---

<sup>1</sup>Other representations could be used (e.g., to better represent numerical attributes such as age), but they might never express relationships between objects. Our model just simplifies the presentation.

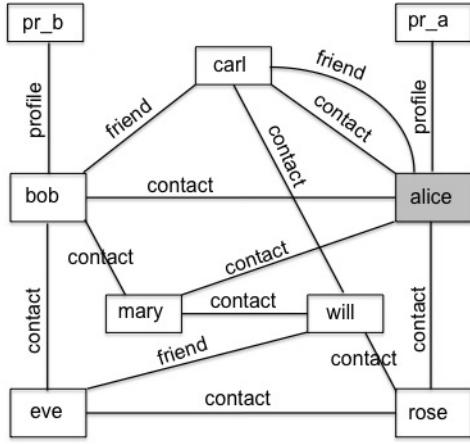


Figure 1: Partial view of the HHC protection state

well as conditions over the paths used to reach these objects. This means that policies define new relations in terms of the relationships in the graph that represents the protection state under consideration.

Before we formally introduce policies let us examine a few examples based on the following scenario. Assume there is a head hunter company, HHC, that has a ReBAC system to manage the access privileges of its clients to profiles of its pool of candidates. To this end, HHC uses the LinkedIn and Facebook profiles of its candidates and clients. Fig. 1 depicts a partial view of the protection state held by HHC. In this graph principals are the nodes `alice`, `bob`, `carl`, `eve`, `mary`, `rose` and `will` and the nodes `pr_b` and `pr_a` are resources. Arcs are labeled with the relationship names `profile` (of), `friend` (of) and `contact` (of). HHC has a special group of candidates qualified as senior advisors depicted inside dark squares in the graph. The principal `alice` is in this group. Hence, the protection state will contain atoms like `rel(bob, profile, pr_b)`, `rel(carl, friend, alice)`, or `prop(alice, senior_advisor)`, etc. HHC policies grant its clients (requesters) access to the professional profiles (resources) from its pool of candidates.

One of the simplest policies HHC could define is that any LinkedIn contact of the owner of a profile can access the profile. This policy can be expressed

in Datalog as follows:

**Policy1**

$$\text{rel}(\text{Res}, \text{profile}, \text{O}), \text{rel}(\text{Req}, \text{contact}, \text{O}) \rightarrow \text{grant}(\text{Req}, \text{Res})$$

Following this policy, if `rel(pr_b, profile, bob)` and `rel(eve, contact, bob)` are in the the protection state, eve is granted access to `pr_b`. It is easy to see that in the protection state depicted in Fig. 1, the access is granted, i.e., we are able to infer `grant(eve, pr_b)`. Expressing this simple policy in Datalog allows us to highlight very basics features of the model. First, the protection state  $I$  will be defined independently from the set of policies  $D$ , and will be the input to the program to answer queries. Second, typically an access request comes with at least two parameters: *who/what* is making the request and *what* resource is being requested. This fact is captured in our formalization by granting to a requester (`eve`) access to a resource (`pr_b`), if the query `grant(eve, pr_b)` is true in  $I \cup D$ :  $I \cup D \models_{\text{Datalog}} \text{grant}(\text{eve}, \text{pr}_b)$ .

The initial motivation behind ReBAC came from social networks where policies are expressed in terms of the relationships between owners of resources and requesters independent of the resources (think of the friend relationship in Facebook and the access that having that relationship grants). Nevertheless, requesters ask for access to resources; the ownership relation is kept as a “tacit condition.” Our policy makes explicit this “tacit condition” by reaching an owner of a resource through the relationship graph (e.g., `rel(Res, profile, O)`), and then, having identified the owner, checking conditions in the paths between the owner and the requester (e.g., `rel(Req, contact, O)`)<sup>2</sup>. Now, let’s assume HHC extends the access to any contact of a contact of the owner of the profile. This condition is modeled in Datalog by the rule below with the introduction of a new variable `Z`:

**Policy2**

$$\begin{aligned} \text{rel}(\text{Res}, \text{profile}, \text{O}), \text{rel}(\text{Req}, \text{contact}, \text{Z}), \\ \text{rel}(\text{Z}, \text{contact}, \text{O}) \rightarrow \text{grant}(\text{Req}, \text{Res}) \end{aligned}$$

<sup>2</sup>For the sake of explanation, we describe as if the rule body is evaluated from left to right, but positive literals can be evaluated in any order. Datalog engines aim to find the order that produces the most efficient evaluation.

From this policy rule and the protection state depicted in Fig. 1 we have that  $I \cup D \models_{Datalog} grant(will, pr\_b)$ . In this case,  $Z$  will be instantiated with `mary`. This policy alone grants access only to contacts that are at distance two of the owner of the profile. To keep access to direct contacts of the owner we need both policy rules. Several rules represent the disjunction of the rules, e.g., Policy1 or Policy2. Notice that neither `carl` nor `rose` has access to `pr_b`. To extend the chain to contacts at distance three we just need a new fresh variable, for instance,  $W$  and the rule will be:

```
rel(Res, profile, O), rel(Req, contact, Z),
rel(Z, contact, W), rel(W, contact, O) → grant(Req, Res)
```

In general, fresh variables memorize intermediate nodes reached along the traversal of chains in the relationship graph to later be recalled in another part of the rule. Observe that evaluating the rule from left to right, in the sub-query  $rel(Req, contact, Z)$ , the variable  $Req$  is bound since it occurs in the request and is “passed” to the program by a query such as  $grant(carl, pr\_b)$ . Then, if there exists an atom  $rel(carl, contact, o)$  in the protection state (like  $rel(carl, contact, will)$ ),  $Z$  will get bound to  $o$ , and hence, bound in the sub-query  $rel(Z, contact, W)$  and so on. This way of traversing relationships in the protection state can be followed to limit the traversal of the graph during policy evaluation to be through objects related to  $Req$  or  $Res$ .

Next, assume HHC wants to grant access to senior advisors’ profiles only when the requester has two different contacts in common with the advisor. This policy can be captured by the following rule:

Policy3

```
rel(Res, profile, O), prop(O, senior_advisor),
rel(Req, contact, Z1), rel(Req, contact, Z2),
rel(O, contact, Z1), rel(O, contact, Z2),
Z1 ≠ Z2 → grant(Req, Res)
```

This policy introduces two new features. One is an example of how properties over objects in the protection state are expressed - the second literal in the

body of the rule. The second one is the use of inequalities to express some counting over relationships that will not be possible without constraints. From Policy3 and the protection state in Fig. 1, we get  $I \cup D \models_{Datalog} grant(will, pr\_a)$ . To extend the policy to three or four contacts we merely need to add extra predicates to traverse the `contact` relation with new variables and then make sure that the variables get bound to different values by introducing more inequalities.

Suppose now that, to minimize conflicts of interest, HHC modifies Policy3, so that these two common contacts cannot both be personal friends of the senior advisor. The policy is modified as follows:

Policy4

```
rel(X, friend, Z1), rel(X, friend, Z2) → r(X, Z1, Z2)

rel(Res, profile, O), prop(O, senior_advisor),
rel(Req, contact, Z1), rel(Req, contact, Z2),
rel(O, contact, Z1), rel(O, contact, Z2),
Z1 ≠ Z2, ¬r(O, Z1, Z2) → grant(Req, Res)
```

The new feature in this policy is negation. The negative condition is defined in two steps. First, a new rule to describe the condition to be complemented is defined. Second, the negation of this condition is added to the policy rule. An important safety condition for the evaluation of negative sub-queries is that the values to check must be derived positively. This implies that all variable bindings in the negative conditions will be limited to values that are mentioned in the protection state (the active language). Hence, the negative sub-query  $\neg r(O, Z1, Z2)$  must be evaluated after all its variables have been bound by other sub-queries in the rule. Considering Fig. 1, we can see that  $I \cup D \models_{Datalog} grant(will, p\_a)$  because  $I \cup D \models_{Datalog} \neg r(alice, carl, rose)$ .

The last example introduces path traversals of unbounded length. HHC wants to grant access to a profile to any contact in the network of contacts of the candidate owning the profile. In this case, the condition over the network of contacts is that there must be a chain (of any length) with ending points the requester and the owner of the resource. This cor-

responds to checking whether for a requester  $u$  asking to get access to a resource  $r$  owned by  $o$ , the pair  $(u, o)$  belongs the transitive closure of the `contact` relation. The formalization in Datalog is the following:

Policy5.

$$\begin{aligned} \text{rel}(X, \text{contact}, Y) &\rightarrow r(X, Y) \\ r(X, Y) &\rightarrow \text{r}_{\text{tc}}(X, Y) \\ \text{r}(X, Z), \text{r}_{\text{tc}}(Z, Y) &\rightarrow \text{r}_{\text{tc}}(X, Y) \end{aligned}$$

$$\text{rel}(\text{Req}, \text{contact}, O), \text{r}_{\text{tc}}(\text{Req}, O) \rightarrow \text{grant}(\text{Req}, \text{Res})$$

The relation  $\text{r}_{\text{tc}}$  consists of all those pairs that appear in some path connecting  $u$  and  $o$  with all the arcs labeled `contact`. This relation is defined in Datalog as a recursive rule (i.e., a rule in which the predicate in the head of the rule also appears in the body). In Fig. 1, we have  $I \cup D \models_{\text{Datalog}} \text{grant}(\text{rose}, \text{pr\_b})$ .

In the rest of this section we formally define ReBAC Datalog policies. In particular, we define policies that cover all the features highlighted in Policy1–Policy5.

For writing policies, in addition to the predicates used in protection states, there are three more types of predicates in the language: a set of binary predicates called *derived relationship predicates*,  $\{\text{nr}_1, \dots, \text{nr}_s\}$ , a corresponding set of binary predicates called *transitive closure relationship predicates*  $\{\text{tnr}_1, \dots, \text{tnr}_s\}$ , and a set of predicates of different arities called *global property predicates*  $\{g_1, \dots, g_t\}$ . We call  $\text{nr}_i$  the *basic predicate* of the transitive closure predicate  $\text{tnr}_i$ . We call *basic literal* any literal of the form  $\text{rel}(t_1, r, t_2)$ ,  $\neg\text{rel}(t_1, r, t_2)$ ,  $\text{prop}(t_1, p)$  and  $\neg\text{prop}(t_1, p)$ , where  $p \in C_p$ ,  $r \in C_r$ , and each  $t_i$  is either a variable or a constant in  $C_n$ . Similarly, we call derived relationship literals, transitive closure literals and global property literals to literals that use predicate symbols from the appropriate sets.

**Definition 2.2** A ReBAC policy  $D$ , comprises two sets of Datalog rules:

1. A non-empty ordered set  $\hat{D} = \{r_1, \dots, r_m\}$  such that the following conditions hold for every  $r_i$ :
  - (a) Every variable that appears either in a negative literal or in a (positive or negative)

global condition literal in the body of  $r_i$ , must also appear in the head or in a positive relationship, transitive closure or basic literal in the body of  $r_i$ .

- (b) If  $r_i$  defines a derived relationship predicate then every variable that appears in the head must also appear in either a derived relationship, transitive closure or basic positive literal in the body of  $r_i$ .
- (c) If a rule  $r_j$  defines either a derived relationship predicate or a global property predicate and the predicate appears in a literal in the body of  $r_i$ , then  $j < i$ .
- (d) Unless  $r_i$  defines `grant`, there is no other rule that defines the predicate defined by  $r_i$ .
- (e) The predicate `grant` does not appear in the body of  $r_i$ .
- (f)  $r_m$  defines the predicate `grant`.

2. A set  $\cup_{i=1}^s TR_i$ , where there is a set  $TR_i$  for each derived relationship predicate  $\text{nr}_i$  containing the rules:

$$\begin{aligned} \text{nr}_i(X, Y) &\rightarrow \text{tnr}_i(X, Y) \\ \text{nr}_i(X, Z), \text{tnr}_i(Z, Y) &\rightarrow \text{tnr}_i(X, Y) \end{aligned}$$

Condition (1a) is the safety condition for the evaluation of derived predicates discussed in the example (Policy4). Condition (1b) is also a safety condition. If variables appear in the head of a rule but not in the body then whenever a grounding of the rule body is true, it fixes the value of the variables in the head that appear in the body. The rest of the variables in the head can be bound to any constant independent of the active domain. Condition (1c) limits recursive definitions to the transitive closures. Condition (1d) limits disjunctive definitions to the predicate `grant`. Condition (1e) prevents `grant` to be defined recursively on itself and Condition (1f) makes sure the predicate `grant` is defined.

We recall that a Datalog program  $D$  is *hierarchical* if there exists an assignment of integers to the predicate symbols such that for every rule in  $D$  the integer assigned to the predicate in the head is larger than

the integers assigned to the predicates in the body.  $D$  is called *stratified* if there is an assignment such that for every rule in  $D$  the integer assigned to the predicate in the head is larger than or equal to the integers assigned to the predicates appearing in positive literals in the body and larger than the integers assigned to predicates appearing in negative literals. It is easy to see that any ReBAC policy  $D$  is always stratified and if it does not use transitive closure relations,  $D$  can be limited to be just  $\hat{D}$ , and hence,  $D$  is hierarchical. It is a well-known property of stratified Datalog programs that they have a unique model [20]. Hence, for any protection state  $I$  and ReBAC policy  $D$  there is a unique *intended* model  $M(D \cup I)$ .

**Definition 2.3** *Given a ReBAC policy  $D$  and a protection state  $I$ , we say that a permission request  $(u, r)$ , from a principal  $u$  to access a resource  $r$  is granted iff*

$$D \cup I \models_{\text{Datalog}} \text{grant}(u, r)$$

Effective mechanisms to answer Datalog queries exist and a lot of effort has gone to optimize these methods since Datalog is the core mathematical foundation of the relational database model and the database query language SQL. More about the complexity and implementation of query answering procedures will be discussed later in the paper.

### 3 EHL ReBAC Policies

The content of this section is mainly from Bruns et al. [4]. In [4] the authors introduced a hybrid logic **HL** for the specification of ReBAC policies. In this logic, from which we have borrowed the terminology for ReBAC Datalog, there are four disjoint sets of symbols, a set  $\mathcal{N}$  of *nominal symbols*, an infinite set  $\mathcal{V}$  of *variables*, a set  $\mathcal{I}$  of *labels* and a set  $\mathcal{P}$  of *propositional symbols*. We denote by  $n$ ,  $X$ ,  $i$  and  $p$  generic nominal symbols, variables, labels and propositional symbols respectively. Policies in **HL** represent properties involving a fixed number of arcs in a relationship graphs. Following [8], we extend the logic to also cover a subclass of properties that can refer to a finite but unbounded set of arcs described as simple regular expressions.

**Definition 3.1** *A formula in the extended hybrid logic **EHL** can be:*

1. *any nominal symbol  $n$ , variable  $X$  or proposition  $p$ ,*
2. *any term of one of the following forms:  $\neg\phi$ ,  $\phi_1 \wedge \phi_2$ ,  $@_n\phi$ ,  $@_X\phi$ , and  $\downarrow X\phi$ ,  $\pi\phi$ , given that  $\phi$ ,  $\phi_1$  and  $\phi_2$  are hybrid formulas and  $\pi$  a path expression having one of the following forms:*
  - (a)  $\epsilon$  representing the empty path
  - (b)  $\langle i \rangle$  or  $\langle -i \rangle$
  - (c)  $\pi_1\pi_2$ , for any two path expressions  $\pi_1, \pi_2$
  - (d)  $\pi^+$ , for any path expression  $\pi$

The definition of **HL** formulas [4] considers only simple path expressions of the form (b) above. *Models* in **EHL** are triples  $(S, \{R_i \subseteq S \times S | i \in \mathcal{I}\}, V)$ , where  $S$  is a non-empty set of nodes, and  $V : \mathcal{N} \cup \mathcal{P} \rightarrow 2^S$ , a total function with  $V(n)$  being a singleton set for any  $n \in \mathcal{N}$ . A *valuation*  $g : \mathcal{V} \rightarrow S$ , is a total function assigning variables to nodes. Let  $g[X \mapsto s]$  denote the valuation that maps  $X$  to  $s$  and any  $X' \neq X$  to  $g(X')$ . A nominal symbol  $n$  will denote the single object in  $V(n)$ . The pair  $(S, \{R_i | i \in \mathcal{I}\})$  can be interpreted as a labeled graph in which its vertexes are the nodes in  $S$  and the labeled arcs between the vertexes are defined by the  $R_i$  relations.

Let us revisit the scenario of policies Policy1–Policy5 from the point of view of models in **EHL**. In Fig. 1, the set of nodes  $S = \{alice, bob, carl, eve, mary, rose, will, pr\_a, pr\_b\}$  corresponds to the nominal symbols in  $\mathcal{N}$ , the relations are  $profile = \{(pr\_b, bob), (pr\_a, alice)\}$ ,  $contact = \{(alice, bob), (alice, carl), (bob, mary), (alice, mary), (alice, rose), (bob, eve), (eve, rose), (mary, will), (rose, will)\}$  and  $friend = \{(alice, carl), (bob, carl), (eve, will)\}$ . We assume that  $V(alice) = \{alice\}$ ,  $V(bob) = \{bob\}$ ,  $V(pr\_b) = \{pr\_b\}$  and  $senior\_advisor$  is a propositional symbol in  $\mathcal{P}$ . In this example,  $V(senior\_advisor) = \{alice\}$ , however, in general, for a propositional symbol  $p$ ,  $V(p)$  is not necessarily a singleton set. The pair  $(S, profile \cup contact \cup friend)$  is called a social graph in [4]. Given an **EHL** model  $M$ , a node  $s \in S$  and a

valuation  $g$ , a satisfiability relation  $\models$  over EHL formulas is defined inductively as follows:

- Definition 3.2**
1.  $M, s, g \models X$  iff  $g(X) = s$
  2.  $M, s, g \models n$  iff  $V(n) = \{s\}$
  3.  $M, s, g \models p$  iff  $s \in V(p)$
  4.  $M, s, g \models \neg\phi$  iff  $M, s, g \not\models \phi$
  5.  $M, s, g \models \phi_1 \wedge \phi_2$  iff  $M, s, g \models \phi_1$  and  $M, s, g \models \phi_2$
  6.  $M, s, g \models \phi_1 \vee \phi_2$  iff  $M, s, g \models \phi_1$  or  $M, s, g \models \phi_2$
  7.  $M, s, g \models @_n\phi$  iff  $M, s^*, g \models \phi$  and  $V(n) = \{s^*\}$
  8.  $M, s, g \models @_X\phi$  iff  $M, g(X), g \models \phi$
  9.  $M, s, g \models \downarrow X\phi$  iff  $M, s, g[X \mapsto s] \models \phi$
  10.  $M, s, g \models \pi\phi$  iff  $M, s', g \models \phi$  for some  $(s, s') \in \mathcal{R}_\pi$ , where  $\mathcal{R}_\pi$  is inductively defined as follows:
    - (a)  $\mathcal{R}_\epsilon = \emptyset$
    - (b)  $\mathcal{R}_{\langle i \rangle} = R_i$
    - (c)  $\mathcal{R}_{\langle -i \rangle} = R_i^{-1}$
    - (d)  $\mathcal{R}_{\pi_1\pi_2} = \mathcal{R}_{\pi_1} \circ \mathcal{R}_{\pi_2}$ , where  $\circ$  denotes relation composition.
    - (e)  $\mathcal{R}_{\pi^+} = \text{trans}(\mathcal{R}_\pi)$ , the transitive closure of  $\mathcal{R}_\pi$ .

Items 1-6, 10b and 10c are standard in modal logics. Items 7-9 are the hybrid operators. Informally speaking,  $@_t$  jumps to the node named by  $t$ , i.e.  $@_t\phi$  holds if  $\phi$  holds at the node identified by  $t$ . In the case of Fig. 1,  $@_{alice}\text{senior\_advisor}$  holds because after jumping to node *alice*, it holds that  $alice \in V(\text{senior\_advisor})$ .

The term  $\downarrow X$  binds the variable  $X$  to the current node, i.e.,  $M, s, g \models \downarrow X\phi$  holds if  $\phi$  holds at  $s$  but with the valuation  $g$  now interpreting  $X$  as  $s$  ( $g$  is replaced with  $g[X \mapsto s]$ ). In the case of Fig. 1,  $@_{bob}\langle friend \rangle \downarrow X\phi$ , jumps to node *bob*, then through the relation *friend* arrives to node *carl*, therefore the variable  $X$  is bound to *carl* and, thus, if  $X$  occurs in the sub-formula  $\phi$ , it refers to *carl*. For another example, let us consider under Fig. 1 the formula

$@_{bob}\langle contact \rangle \downarrow X_1\langle contact \rangle \downarrow X_2\langle contact \rangle \downarrow X_3\phi$ . The evaluation starts at the node *bob* and it holds if there exists a chain of contacts of length 3 and the sub-formula  $\phi$  holds with variables  $X_1, X_2$  and  $X_3$  bound to the nodes in the chain: *mary*, *will* and *rose* are examples of such nodes. The usual notions of free and bound variables in a formula are defined based on the bindings produced by  $\downarrow$ . Item 3.2.10e corresponds to the notion of closure for regular expressions.

As in ReBAC Datalog, policies are evaluated in the context of a concrete model  $M$  (corresponding to a protection state), and a request  $(u, r)$ .

**Definition 3.3** A policy is an EHL formula that may have at most *Res* and *Req* as free variables and is a Boolean combination of formulas of the form  $@_{Res}\phi_1$  or  $@_{Req}\phi_2$ .

**Definition 3.4** Given a policy  $\phi$ , a permission request  $(u, r)$  is granted in a model  $M$  iff

$$M, s, g[Req \mapsto u, Res \mapsto r] \models \phi$$

for some  $s \in S$  and valuation  $g$ .

Since *Res* and *Req* are the only variables that can occur free in  $\phi$ ,  $s$  and  $g$  are irrelevant for granting the permission. Thus, from the rest of the paper we will write  $M, [X_1 \mapsto s_1, \dots, X_m \mapsto s_m] \models \phi$ , when the only free variables in  $\phi$  are  $X_1, \dots, X_m$ . In the presentation of the logic in [4], the owner of the resource and not the resource itself is used in the policies since  $M$  is presented as a “social graph”, nodes are restricted to be principals, and policies are assumed to be associated to a particular resource for which the owner is known. However, the authors recognize that more general settings can be defined and refer to the general case described here as *heterogeneous* protection states. Having an action in the request is also common but we will discuss this later in the paper.

Some examples of EHL policies adapted from [4] are:

$$@_{Res}\langle -profile \rangle \langle contact \rangle Req \quad (2)$$

that grants access to any contact of the owner of the resource.

$$@_{Res}\langle -profile \rangle \langle contact \rangle (Req \vee \langle contact \rangle Req) \quad (3)$$

that grants access to a contact or a contact of a contact of the owner of the resource.

$$@_{Res} \langle -profile \rangle \langle contact \rangle (Req \wedge senior\_advisor) \quad (4)$$

that grants access to a contact of the owner if he or she is a senior advisor.

$$@_{Res} \langle -profile \rangle \langle contact \rangle (Req \wedge \neg Bob) \quad (5)$$

that grants access to a contact of the owner who can't be Bob.

$$@_{Res} \langle -profile \rangle (\langle friend \rangle Req \wedge \neg \langle friend \rangle \neg Req) \quad (6)$$

that grants access to a friend of the owner if he or she is the only friend.

A salient feature of the original **HL** language (and thus, of **EHL** and ReBAC Datalog) is the ability to express graded modalities. Given a positive integer  $k$ , one can write  $\langle i \rangle_k \phi$  as a shorthand for:

$$\begin{aligned} \downarrow X \langle i \rangle \downarrow Y_1 (\phi \wedge @_X \langle i \rangle \downarrow Y_2 (\neg Y_1 \wedge \phi \wedge \\ \dots @_X \langle i \rangle \downarrow Y_k (\neg Y_1 \wedge \neg Y_{k-1} \wedge \phi) \dots)) \end{aligned}$$

which informally says that the formula holds in a node  $s$  iff there are at least  $k$   $R_i$ -successors of  $s$  at which  $\phi$  holds. For example, a formula granting access to a requester that has at least three contacts in common with the profile's owner is:

$$@_{Res} \langle -profile \rangle \langle contact \rangle_3 (\langle contact \rangle Req) \quad (7)$$

This essentially the same encoding of counting through inequalities done in Policy3.

The following policy is adapted from [8]:

$$@_{Res} \langle -profile \rangle (\langle member\_of \rangle \langle -supervise \rangle)^+ Req \quad (8)$$

that grants permission to any supervisor in the management chain to access profiles owned by members of the groups under her management line.

## 4 From EHL to ReBAC Data-log

Given an **EHL** policy defined over sets  $\mathcal{N}$ ,  $\mathcal{V}$ ,  $\mathcal{I}$  and  $\mathcal{P}$ , an **EHL** model  $M = (S, \{R_i \subseteq S \times S | i \in \mathcal{I}\}, V)$ ,

and a policy  $\phi$ , we want to find an equivalent ReBAC Datalog policy  $[\phi]$  and protection state  $[M]$ .

Without loss of generality, we assume that all bound variables in  $\phi$  are named differently. We also assume that the model has been fixed. Hence, when we refer to  $S$ ,  $R_i$  or  $V$  in any of the definitions we are referring to the nodes, relations and the function  $V$  of this model. The following equivalences of **HL** formulas are easy to verify:

1.  $@_{t_1} @_{t_2} \phi \equiv @_{t_2} \phi$ ;
2.  $\neg @_t \phi \equiv @_t \neg \phi$ ;
3.  $@_t (\phi_1 \vee \phi_2) \equiv (@_t \phi_1 \vee @_t \phi_2)$ ; and
4.  $\neg \downarrow X \phi \equiv \downarrow X \neg \phi$ ,

for any  $t$ ,  $t_1$  and  $t_2$  nominal symbols or variables. Using these equivalences and De Morgan's laws we can normalize **EHL** formulas by pushing all negations to be in front of nominal symbols, variables, propositional symbols or non-empty path expressions, as well as removing multiple occurrences of  $@$  in front of any formula. A formula is called *normal conjunctive* if it does not contain disjunctions, all the negations appear in front of nominal symbols, variables or non-empty path expressions and there are no redundant  $@$ -operators. A formula is in *disjunctive form* if it is a disjunction of normal conjunctive formulas. It is easy to see that every formula has an equivalent formula in disjunctive form. For the rest of the presentation we assume that all **EHL** formulas are in disjunctive form. Let the sets  $C = C_n \cup C_p \cup C_r$ ,  $\text{Var}$ ,  $P$  of constant symbols, variables and predicate symbols be such that  $\mathcal{N} \subseteq C_n$ ,  $\mathcal{I} \subseteq C_r$ ,  $\mathcal{P} \subseteq C_p$  and  $\mathcal{V} \subseteq \text{Var}$ . Without loss of generality, we assume that for any nominal symbol  $n$ ,  $V(n) = \{n\}$ .<sup>3</sup> In what follows, for the sake of readability, we will use italics in **EHL** formulas and continue using math serif font for Datalog. Intuitively, each normal conjunctive sub-formula occurring in a disjunctive formula representing a policy, can be seen as a partial definition of the policy. This intuition gives us insights about how to proceed in order to translate an **EHL** policy into a ReBAC program. Given an **EHL** formula  $\phi$ , we define the program

---

<sup>3</sup>This is, the syntax of the constant in the language is the same as value in the model (Herbrand-like).

$[\phi]$  in three steps. Firstly, we provide a mechanism to translate each normal conjunctive sub-formula of  $\phi$  into pairs where the first component is a set of literals and the second component is a set of ReBAC rules. Second, for each normal conjunctive formula in  $\phi$ , we associate a definition of the binary predicate `grant` using each individual translation. Finally, we join all these `grant` definitions with the translation of the EHL model into a Datalog protection state to get the ReBAC policy and Input to evaluate queries. The next two definitions formalize these steps.

**Definition 4.1** Given a variable  $X \in \text{Var}$ , for any conjunctive normal EHL formula  $\phi$ ,  $[\phi]^X$  defines inductively a set  $B$  of constraints and literals, and a set  $R$  of Datalog rules in a pair  $(B, R)$  as follows:

1.  $[X']^X = (\{X' = X\}, \emptyset)$
2.  $[n]^X = (\{n = X\}, \emptyset)$
3.  $[p]^X = (\{\text{prop}(X, p)\}, \emptyset)$  iff  $p \in \mathcal{P}$
4.  $[\neg\phi]^X = (\{X' \neq X\}, \emptyset)$ , iff  $\phi \equiv X'$ ;  
 $[\neg\phi]^X = (\{n \neq X\}, \emptyset)$ , iff  $\phi \equiv n$  and  $V(n) = \{n\}$ ;  
otherwise  
 $[\neg\phi]^X = (\{\neg\bar{\phi}(\bar{V}, X)\}, \{B \rightarrow \bar{\phi}(\bar{V}, X)\} \cup R')$  iff  
 $[\phi]^X = (B, R')$ ,  $\bar{V}$  are the free variables appearing in  $\phi$ , and  $\bar{\phi}$  is a new global property predicate symbol of arity equal to the cardinality of  $\bar{V}$  plus 1.
5.  $[\phi_1 \wedge \phi_2]^X = (B_1 \cup B_2, R_1 \cup R_2)$  iff  $[\phi_1]^X = (B_1, R_1)$  and  $[\phi_2]^X = (B_2, R_2)$
6.  $[@_n\phi]^X = (\{n = Y\} \cup B, R)$  iff  $[\phi]^Y = (B, R)$ ,  $Y$  is a new fresh variable from  $\text{Var}$
7.  $[@_{X'}\phi]^X = (B \cup \{X' = Z\}, R)$ ,  $Z$  is a new fresh variable from  $\text{Var}$ , and  $[\phi]^Z = (B, R)$
8.  $[\downarrow X'\phi]^X = (\{X' = X\} \cup B, R)$  iff  $[\phi]^X = (B, R)$
9. For  $[\pi\phi]^X$ , when
  - (a)  $\pi \equiv \epsilon$ , then  $[\pi\phi]^X = [\phi]^X$
  - (b)  $\pi \equiv \langle i \rangle$ , then  $[\pi\phi]^X = (\{\text{rel}(X, i, Y)\} \cup B, R)$  if and only if  $[\phi]^Y = (B, R)$  and  $Y$  is a new fresh variable from  $\text{Var}$

(c)  $\pi \equiv \langle -i \rangle$ , then  $[\pi\phi]^X = (\{\text{rel}(Y, i, X)\} \cup B, R)$  if and only if  $[\phi]^Y = (B, R)$  and  $Y$  is a new fresh variable from  $\text{Var}$

(d)  $\pi \equiv \pi_1\pi_2$ , then  $[\pi\phi]^X = (B_{\pi_1 Y} \cup B_{\pi_2\phi}, R_{\pi_1 Y} \cup R_{\pi_2\phi})$ , where  $[\pi_1 Y]^X = (B_{\pi_1 Y}, R_{\pi_1 Y})$  and  $[\pi_2\phi]^Y = (B_{\pi_2\phi}, R_{\pi_2\phi})$

(e)  $\pi \equiv \pi_1^+$ , then  $[\pi\phi]^X = (\{\text{pi}_+(X, Y)\} \cup B_\phi, R_\phi \cup R_{\pi_1 Y} \cup R_{\text{tc}})$  if and only if

i.  $[\phi]^Y = (B_\phi, R_\phi), [\pi_1 Y]^X = (B_{\pi_1 Y}, R_{\pi_1 Y})$  and  $Y$  is a new fresh variable from  $\text{Var}$

ii.  $\text{pi}$  is a new derived relationship predicate symbol and  $\text{pi}_+$  its corresponding transitive closure predicate, and

$$\begin{aligned} R_{\text{tc}} &= \{B_{\pi_1 Y} \rightarrow \text{pi}(X, Y), \\ &\quad \text{pi}(X, Y) \rightarrow \text{pi}_+(X, Y), \\ &\quad \text{pi}(X, Z), \text{pi}_+(Z, Y) \rightarrow \text{pi}_+(X, Y)\} \end{aligned}$$

**Definition 4.2** For any EHL policy  $\phi = \phi'_1 \vee \dots \vee \phi'_m$  in disjunctive form and an EHL model  $M$ . Let  $\phi'_i = @_X_i \phi_i$  and  $[\phi'_i]^{X_i} = (B_i, R_i)$ ,  $i \in \{1, \dots, m\}$ .  $[\phi]$  and  $[M]$  define the following Datalog program and its input:

$$\begin{aligned} [\phi] &= \bigcup_{i=1}^m (\{B_i \rightarrow \text{grant}(\text{Res}, \text{Req})\} \cup R_i) \\ [M] &= \{\text{prop}(s, p) : p \in \mathcal{P}, s \in V(p)\} \cup \{\text{rel}(s, i, s') : (s, s') \in R_i\} \end{aligned}$$

As we see, we are considering the  $\vee$  operator separately and use Def. 4.1 and Def. 4.2 to get the translations for policies. Note that  $X_i$  is either `Res` or `Req` for every  $X_i$  in the definition. The next example illustrates several of the steps in the translation.

**Example 4.1** Let us consider a very simple EHL path expression formula that grants access to a profile to any direct or indirect contact of the owner of the profile:

$$\phi = @_R e s \langle -\text{profile} \rangle \langle \text{contact} \rangle^+ \text{Req}$$

This policy is already in disjunctive form with a single conjunctive formula,  $\phi_1 = \langle -\text{profile} \rangle \langle \text{contact} \rangle^+ \text{Req}$  and  $X_1 = \text{Res}$ . Thus,

$$[\phi_1]^{\text{Res}} = [\overbrace{\langle -\text{profile} \rangle}^{\pi_1} \overbrace{\langle \text{contact} \rangle}^{\pi_2}]^+ \text{Req} = (\mathcal{B}_{\phi_1}, \mathcal{R}_{\phi_1})$$

$$(\mathcal{B}_{\phi_1}, \mathcal{R}_{\phi_1}) \stackrel{\text{Def. 4.1.9d}}{=} (\mathcal{B}_{\pi_1 Y} \cup \mathcal{B}_{\pi_2 \text{Req}}, \mathcal{R}_{\pi_1 Y} \cup \mathcal{R}_{\pi_2 \text{Req}}) \quad (9)$$

where  $(\mathcal{B}_{\pi_1 Y}, \mathcal{R}_{\pi_1 Y}) = [\pi_1 Y]^{\text{Res}}$  and  $(\mathcal{B}_{\pi_2 \text{Req}}, \mathcal{R}_{\pi_2 \text{Req}}) = [\pi_2 \text{Req}]^Y$

$$(\mathcal{B}_{\pi_1 Y}, \mathcal{R}_{\pi_1 Y}) \stackrel{\text{Def. 4.1.9c}}{=} (\{\text{rel}(Y_1, \text{profile}, \text{Res}), Y = Y_1\}, \emptyset) \quad (10)$$

since  $[Y]^{Y_1} \stackrel{\text{Def. 4.1.1}}{=} (\{Y = Y_1\}, \emptyset)$

$$(\mathcal{B}_{\pi_2 \text{Req}}, \mathcal{R}_{\pi_2 \text{Req}}) \stackrel{\text{Def. 4.1.9e}}{=} (\{\text{pi}_+(Y, Y_2), Y_2 = \text{Req}\}, \mathcal{R}_{\pi_2 Y_2}) \quad (11)$$

since  $[\text{Req}]^{Y_2} \stackrel{\text{Def. 4.1.1}}{=} (\{Y_2 = \text{Req}\}, \emptyset)$ . Additionally,

$$[\pi_2 Y_2]^Y \stackrel{\text{Def. 4.1.9b, Def. 4.1.1}}{=} (\{\text{rel}(Y, \text{contact}, Y_3), Y_3 = Y_2\}, \emptyset).$$

Hence

$$\begin{aligned} \mathcal{R}_{\text{tc}} &= \{\text{rel}(Y, \text{contact}, Y_3), Y_3 = Y_2 \rightarrow \text{pi}(Y, Y_2) \\ &\quad \text{pi}(Y, Y_2) \rightarrow \text{pi}_+(Y, Y_2) \\ &\quad \text{pi}(Y, Y_3), \text{pi}_+(Y_3, Y_2) \rightarrow \text{pi}_+(Y, Y_2)\} \end{aligned} \quad (12)$$

and

$$(\mathcal{B}_{\pi_2 \text{Req}}, \mathcal{R}_{\pi_2 \text{Req}}) = (\{\text{pi}_+(Y, Y_2), Y_2 = \text{Req}\}, \mathcal{R}_{\text{tc}}) \quad (13)$$

From (10), (12) and (13) we obtain that the pair  $(\mathcal{B}_{\phi_1}, \mathcal{R}_{\phi_1})$  in (9) can be rewritten as

$$\begin{aligned} &(\{\text{rel}(Y_1, \text{profile}, \text{Res}), Y = Y_1, \text{pi}_+(Y, Y_2), Y_2 = \text{Req}\}, \\ &\quad \{\text{rel}(Y, \text{contact}, Y_3), Y_3 = Y_2 \rightarrow \text{pi}(Y, Y_2) \\ &\quad \quad \text{pi}(Y, Y_2) \rightarrow \text{pi}_+(Y, Y_2) \\ &\quad \quad \text{pi}(Y, Y_3), \text{pi}_+(Y_3, Y_2) \rightarrow \text{pi}_+(Y, Y_2)\}) \end{aligned}$$

and finally, rewriting the pair above we have that  $(\mathcal{B}_{\phi_1}, \mathcal{R}_{\phi_1})$  equals to

$$\begin{aligned} &(\{\text{rel}(Y, \text{profile}, \text{Res}), \text{pi}_+(Y, \text{Req})\}, \\ &\quad \{\text{rel}(Y, \text{contact}, Y_2), \rightarrow \text{pi}(Y, Y_2) \\ &\quad \quad \text{pi}(Y, Y_2) \rightarrow \text{pi}_+(Y, Y_2) \\ &\quad \quad \text{pi}(Y, Y_3), \text{pi}_+(Y_3, Y_2) \rightarrow \text{pi}_+(Y, Y_2)\}) \end{aligned}$$

Consequently, by Def.4.1 and Def.4.2 the ReBAC Datalog program associated to  $\phi$ ,  $[\phi]$ , is

$$\begin{aligned} &\{\text{rel}(Y, \text{profile}, \text{Res}), \text{pi}_+(Y, \text{Req}) \rightarrow \text{grant}(\text{Req}, \text{Res}), \\ &\quad \text{rel}(Y, \text{contact}, Y_2), \rightarrow \text{pi}(Y, Y_2) \\ &\quad \quad \text{pi}(Y, Y_2) \rightarrow \text{pi}_+(Y, Y_2) \\ &\quad \quad \text{pi}(Y, Y_3), \text{pi}_+(Y_3, Y_2) \rightarrow \text{pi}_+(Y, Y_2)\} \end{aligned}$$

Given a protection state, a policy and a permission request, the next theorem establishes the relationship between granting permissions in EHL and query answering in ReBAC Datalog programs.

**Theorem 4.1** Given an EHL policy  $\phi$  in disjunctive form, an EHL model  $M$  and a permission request  $(u, r)$

$$M, [Req \mapsto u, Res \mapsto r] \models \phi \text{ iff } [M] \cup [\phi] \models_{\text{Datalog}} \text{grant}(u, r)$$

**Proof sketch:** the proof is based on the following lemma:

**Lemma 4.1** Let  $\phi$  be a normal conjunctive EHL formula,  $M = (S, \{R_i \subseteq S \times S | i \in \mathcal{I}\}, V)$  a model,  $s$  a node in  $S$  and  $g : V \rightarrow S$  an assignment such that  $M, s, g \models \phi$ . Let  $X \in \text{Var}$  be a fresh variable not appearing in  $\phi$ . Then,

$$[M] \cup \{X = s, B \rightarrow q(\bar{V}, X)\} \cup R \models_{\text{Datalog}} q(\bar{a}, X),$$

where  $\bar{V}$  is the set of free variables in  $\phi$ ,  $\bar{a}$  is the assignment of  $\bar{V}$  in  $g$ ,  $q$  is a fresh predicate and  $[\phi]^X = (B, R)$

This lemma works over general normal conjunctive formulas without the restriction imposed in policies by EHL over free variables. Hence we are able to do an inductive proof based on the structure of  $\phi$ . The case in which  $\phi \equiv \pi^+ \phi'$  requires a second induction to cover the transitive closure.

## 5 From ReBAC Datalog to EHL

There are two types of ReBAC Datalog policies that cannot be expressed within EHL. An example of the

first type of policies is the following:

$$\begin{aligned} \text{rel}(X, i, Y), \text{prop}(Y, p_1) &\rightarrow r(X, Y) \\ r(X, Y) &\rightarrow \text{tr}(X, Y) \\ r(X, Z), \text{tr}(Z, Y) &\rightarrow \text{tr}(X, Y) \\ \text{tr}(\text{Req}, \text{Res}) &\rightarrow \text{grant}(\text{Req}, \text{Res}) \end{aligned}$$

In this policy, a property is checked on every object in the path between Res and Req. Such conditions cannot be imposed in a path expression. To limit the expressibility of ReBAC Datalog to path expressions and avoid this type of policies, we need *simple* definitions of derived relationships. We need to limit the literals that can appear in the body of a derived relationship definition to be either positive `rel` literals or transitive closure relationship literals – no negation and no basic or global property literals.

An example of the second type of policies is the following:

$$\text{rel}(X, i, Y) \rightarrow \text{grant}(\text{Req}, \text{Res}) \quad (14)$$

This says that access is granted if  $R_i$  in the protection state is not empty. To exclude this type of policies we need to limit the variables that appear in any ReBAC Datalog rule as follows:

**Definition 5.1** For a Datalog rule of the form (1) we say that:

1. A variable that appears in a literal  $L_k$ ,  $k \leq m$ , is seeded iff it also appears either in A or in a literal  $L_i$ ,  $i < k$ .
2. A negative literal is well-seeded iff all its variables are seeded.
3. A positive literal is well-seeded iff at least one of its variables is seeded.

The rule is well-seeded iff the literals in its body can be re-arrange so that all of them become well-seeded and the variables appearing in the constraints are seeded.

The rule in Eq.(14) is not well-seeded since neither of the variables, X or Y, appears in the head or in a predicate in the body together with another well-seeded variable (or constant). Now we have the following proposition.

**Proposition 5.1** A ReBAC Datalog policy that only uses simple derived relationship definitions and all its rules are well-seeded can be translated to an EHL formula. Furthermore, if the policy does not use transitive closure relationships it can be translated into an HL formula.

**Proof sketch:** the transformation starts from the `grant` rules and is more or less straightforward if it is done using a well-seeded order traversal of the literals in the rule by binding a variable with  $\downarrow$  the first time the variable is encountered in the rule.

We skip the transformation due to space limitations. In addition, there is no equivalent EHL policies for most of the extensions discussed in the following section.

## 6 Extensions

Permissions are usually granted not to simply access a resource but to do something with it. For example, Alice may want access to a file to read and modify it. Hence the granularity of the permissions should be at the level of the operation. We can represent requests as a triple  $(u, r, a)$ , where  $u$  is the principal requesting access to the resource  $r$  and  $a$  is the *action* the principal wants to apply to the resource. If the set of actions is part of  $S$  in the protection state, there could be an “implements” relations over resources and actions and we can allow three free variables in an EHL policy  $\phi$ :  $\text{Req}, \text{Res}, A$ . A request  $(u, r, a)$  is granted under the policy  $\phi$  iff

$$M, [\text{Req} \mapsto u, \text{Res} \mapsto r, A \mapsto a] \models \phi$$

and the `grant` Datalog rules will be of the form

$$B \rightarrow \text{grant}(\text{Req}, \text{Res}, A)$$

For example, the policy that let any friend of the owner of a resource Res to copy Res is written as follows:

$$\text{rel}(\text{Res}, \text{implements}, \text{copy}),$$

$$\text{rel}(\text{O}, \text{owns}, \text{Res}), \text{rel}(\text{O}, \text{friend}, \text{Req}) \rightarrow \text{grant}(\text{Req}, \text{Res}, \text{copy})$$

Similar to permission granting rules, negative authorizations can be defined by a formula  $\phi'$  such that access is denied when:

$$M, [Req \mapsto u, Res \mapsto r, A \mapsto a] \models \phi'$$

The Datalog rule of a negative authorization will be of the form

$$B \rightarrow \text{deny}(\text{Req}, \text{Res}, A)$$

Having negative authorizations introduces two problems. One is what to do if a request is neither granted nor denied. The second is what to do with conflicting decisions. The first issue of policy coverage is a semantic issue. We could have a meta-rule to cover the missing cases but this meta-rule may hide the gaps of what it could be an incomplete policy otherwise. In addition, if meta-rules are used one needs to re-examine the need of complicating the policy specification with negative and positive authorizations since one could, in principle, specify one type of policy and let the meta-rule cover the other type (like in *any request that is not granted is denied*). A more practical problem is to discover policy gaps. So far, we have used Datalog programs to answer ground queries (e.g., `grant(u, r, a)`). By typing the objects in a protection state and adding them as part of the input, we can also ask existentially quantified queries and do gap analysis with the rule:

$$\begin{aligned} &\text{prop}(\text{Req}, \text{principal}), \text{prop}(\text{Res}, \text{resource}), \\ &\quad \text{prop}(A, \text{action}), \end{aligned}$$

$$\neg\text{grant}(\text{Req}, \text{Res}, A), \neg\text{deny}(\text{Req}, \text{Res}, A) \rightarrow \text{gap}(\text{Req}, \text{Res}, A)$$

and the query:

$$D \models_{\text{Datalog}} \exists \text{Req} \exists \text{Res} \exists A (\text{gap}(\text{Req}, \text{Res}, A))$$

For analysis, we are assuming that propositions exist in  $D$  typing the constants in the active domain.

There are three complexity characterizations for query evaluation in Datalog and logic programs. In one characterization, called data complexity, the complexity is characterized in terms of the input size (in our case, the protection state) while the Datalog program (in our case the ReBAC policies) and

the query are fixed. If, on the other hand, the input is fixed and the program and the query size is what matters, the complexity of query evaluation is called program complexity. If both the program and the input are considered part of the problem size the characterization is called program+data complexity. Most of the time in database applications data complexity is considered sufficient since the size of the data represented by the input is much larger than the size of the program. We will show in the next section why this is also a reasonable assumption for ReBAC policies.

Efficient procedures (PTIME data complexity) exist not only to decide if the answer is yes or no, but also to obtain values for the existentially quantified variables in a query like the one to check for gaps.

Conflicts can be an indication of policy errors. However, including policy conflict resolution rules in the semantics of policy evaluation is a common practice since many times it facilitates policy specification. A typical policy conflict resolution rule is denies-overrides-allows. This can be easily incorporated into Datalog policies by rewriting each granting access rule as follows:

$$B, \neg\text{deny}(\text{Req}, \text{Res}, A) \rightarrow \text{grant}(\text{Req}, \text{Res}, A)$$

There are many conflict resolution strategies that can be borrowed from other Datalog models – the interested reader can find in [18] an extensive study of authorizations overrides meta-policies and how to express them in terms of logic programs.

**History-based Policies** It is common to find examples of access control policies that depend on the occurrence of past events. In the context of ReBAC, motivated by access control policies found in community-based collaborations, Fong et al [13] has extended HL with linear past temporal operators. Two examples from [13] are:

- A user who has been reported for using inappropriate language twice is suspended for further editing.
- A user who has already created two distinct objects that have since remained untouched by any

member of the community (including herself) is not allowed to further create new objects.

Handling history-based policies in the context of Datalog has been discussed in [22]. This is achieved by adding a time argument to all the predicates and allowing a limited class of time constraints over time variables. To illustrate how it works we will encode the second example above:

$$\begin{aligned}
& T_1 \leq T, T_2 \leq T, \\
& \text{rel}(U, \text{own}, O_1, T_1), \text{rel}(U, \text{own}, O_2, T_2), \\
& \neg \text{twoEd}(O_1, O_2, T), O_1 \neq O_2 \rightarrow \text{deny}(U, O, \text{create}, T) \\
& T_1 \leq T, T_2 \leq T, \text{rel}(O_1, \text{edited}, U_1, T_1), \\
& \quad \text{rel}(O_2, \text{edited}, U_2, T_2) \rightarrow \text{twoEd}(O_1, O_2, T)
\end{aligned}$$

The intuition behind the rules is that the  $T_i$  variables will be instantiated with time values, and events like creation of objects, or modifications of objects will be incorporated into the protection state (these events can be captured each time a request to execute these operations is granted/denied) and the state will evolve over time. Hence, given two objects  $o_1$  and  $o_2$ , and a fixed time  $t$ ,  $\text{twoEd}(o_1, o_2, t)$  will hold if there are time points  $t_1$  and  $t_2$  before (or equals to)  $t$  for which  $\text{rel}(u, \text{edited}, o_1, t_1)$  and  $\text{rel}(u, \text{edited}, o_2, t_2)$  are part of the corresponding states.

A *time constraint*  $C$  is any expression of the form  $T_1 \oplus T_2 \pm c$ , where  $T_1$  and  $T_2$  are different time variables,  $c$  is a non-negative real number and  $\oplus$  is one of  $\{=, \leq, <\}$ . These binary relations are interpreted under the standard order of time. Several constraints can appear in a rule but all the time variables in the constraints must also appear either in the head of the rule or in a literal in the body. In addition, if  $T$  is the time variable appearing in the head, and  $C_1, \dots, C_n$  all the constraints appearing in the body, then for any variable  $T_i$  that appears in the constraints, it must be the case that  $C_1, \dots, C_n \models_{\text{Datalog}} T_i \leq T$ . This ensures that policy evaluations do not depend on “future” states. In the non-temporal case, policies were evaluated in a protection state. In the case of temporal policies, all the ground atoms belonging to the same temporal protection state will be extended with an extra-argument which will be a time constant - the same constant in all the atoms. Note that there

is no way to specify absolute values for the  $T_i$ 's in the rules, all times are relative to  $T$  which is also a variable. Similar to [13], policy compliance is defined in terms of traces. A trace  $\mathcal{T}$ , is a (possibly infinite) sequence of temporal states  $\langle S_0, S_1, \dots \rangle$ , such that constants  $t_i, t_j$  associated to the atoms in states  $S_i, S_j$  are such that  $t_i \leq t_j$  if  $i \leq j$ . Intuitively,  $\mathcal{T}$  represents the history of the protection state evolution over time. How the evolution happens over time is not relevant for our discussion. Given a set of temporal policy rules  $P$  and a trace  $\mathcal{T}$ , a permission request  $(\text{req}, \text{res}, a)$  is granted at time  $t$  iff

$$P \cup \mathcal{T} \models_{\text{Datalog}} \text{grant}(\text{req}, \text{res}, a, t)$$

The crucial point here is that conditions in any rule refer to properties that must be true either at the same state where the head of the rule is true or in an earlier state, and when a permission is requested it is assumed that the request is to grant the permission at the current time, i.e., the time when the request is made. The results in [22] also show how effective monitors that only keep the historical data required to evaluate the rules can be implemented instead of having copies of multiple states. Each update step executed by the monitor takes time proportional to the size of the update made to the protection state. This is in contrast to the results in [13] in which the steps take time proportional to the size of the state. The same monitors from [22] can be used for historical ReBAC if the only time variable that can appear in the rules representing path expressions is the variable that appears in the head (and thus there are not temporal constraints in the recursive rules). In other words path expressions refers to paths in a single protection state.

## 7 Datalog as an implementation

In contrast to policy analysis where time is not so much an issue, the complexity of access control decisions must consider the effect of the policy, i.e., the Datalog rules. Program complexity in Datalog is EXPTIME-complete [11]. In terms of ReBAC Datalog that would mean that fixing a protection state,

there is a policy that takes exponential time to evaluate with respect to the size of the the policy itself + the fixed size of the protection state. This result applies even if the Datalog rules are well-seeded and no transitive closure relationships are used. Therefore, the result also applies to HL policies. The hardness part of the EXPTIME complexity proof depends on the fact that there are no limitations in the arity of the predicate relations that can define the Datalog program - the standard proof uses an encoding of a deterministic Turing machine that halts in less than  $2^{n^k}$  steps and uses predicates of arity in the order of  $O(n)$ . These are very large programs. In ReBAC Datalog policies, all predicates of arity  $> 3$  appear in global condition literals. If we assume a constant  $k$  exists that limits the maximal arity of any predicate the complexity reduces to NP. The intractability persists because of the inequalities. Inequalities permit to encode the Hamiltonian path problem [23]. The encoding of the problem for a path of length  $n$  uses  $n$  different variables in the inequalities of a single rule. Again, this is a very large program. If we can also assume that the number of different variables that appear in the inequalities of a single rule does not exceed a constant  $k$  we obtain tractability. Furthermore, the result is tight.

**Proposition 7.1** *ReBAC Datalog programs with all predicates with arity  $\leq k$  and rules with constraints that used  $\leq k$  variables is program+data complete for P.*

This follows directly from the facts that (1) Datalog programs that are limited to use  $\leq k$  variables per rule is data+program complete for P [28], and (2) that using the result that Stratified Datalog with negation is data complete for P and program complete for EXPTIME [1] together with the same techniques from [28], one can show that stratified Datalog programs with negation that are limited to use  $\leq k$  variables per rule are also data+program complete for P. These proofs rest on the fact that any intermediate result needed to evaluate the rules is no more than polynomially larger than the input size. In ReBAC Datalog programs, the size of any derived relation is a polynomial function on the size of the protection state. More precisely, if the number of constants in

the protection state is  $m$ , the size of a derived relation can be bound to  $O(m^k)$ , assuming  $k$  to be the maximal predicate arity. Take, for example, `grant(X, Y)`. The maximum number of different values that  $X$  or  $Y$  can take is  $m$ . Hence, the number of ground atoms is bound by  $m^2$ . The number of relations defined by policies (i.e., the number of different predicate names appearing in the head of at least one rule is limited by the number  $l$  of program rules, therefore, an evaluation of the ReBAC program can be done in  $O(lm^{k^2})$ ). The square is added as an upper bound of rule evaluation in case there are recursive rules. In practice, this number is much smaller, and for a given request `grant(u, r, a)`,  $l$  will be determined by how well we can index the rules based on  $u$ ,  $r$ , and  $a$ , to pull out the subset of rules that apply to the specific request. One could use the principal matching rules concept from [8] or the user-to-user relationship-based access control model of [7] to organize policies and create an indexing.

There is a syntactic characteristic of the program rules that is used to ensure that intermediate results are kept small: we have already observed how the propagation of information through variable bindings happens in the rules. Take, for example, the rule:

```
rel(Res, profile, O), rel(Req, contact, Z),
    rel(Z, contact, O) → grant(Req, Res)
```

In terms of database operations, the evaluation of the rule requires two joins. We know that at the moment of evaluation, values for the variables `Req` and `Res` will be fixed. Therefore, the evaluation of `rel(Res, profile, O)` will produce a single value for `O`. The expected number of values for `Z` returned by the evaluation of `rel(Req, contact, Z)` can be estimated by the typical values of contact list sizes given that `Req` is fixed. Similarly, the expected number of values for `Z` in the evaluation `rel(Z, contact, O)` can be estimated. This is called the selectivity of the evaluation, the smaller the expected number of values, the higher the selectivity. Given that the selection operations in databases can be done much faster than the joins, modern database systems do query planning before query evaluation to find the right order to evaluate the joins. If, for example, the order is first to do

the join  $\text{rel}(\text{Req}, \text{contact}, Z), \text{rel}(Z, \text{contact}, O)$ , before doing the second join with  $\text{rel}(\text{Res}, \text{profile}, O)$  a projection over  $O$  is done in the relation obtained from the join  $\text{rel}(\text{Req}, \text{contact}, Z), \text{rel}(Z, \text{contact}, O)$  and the joint relation can be discarded before doing the (semi) join with  $\text{rel}(\text{Res}, \text{profile}, O)$ . In this case there can never be a relation with more than  $m^2$  tuples during the computation. In contrast, creating the  $(\text{Res}, O, \text{Req}, Z)$  joint table could in principal generate a relation with  $m^4$  tuples. This dependency of shared variables is known as a Sideway Information Passing (SIP) optimization and it is fundamental for the Magic Sets optimization technique applied to recursive Datalog rules. Given that the evaluation of an access control decision in the Datalog program is always answering a ground query this optimization will be very effective, essentially transforming the query answering into a goal oriented procedure. This means that the search space will be very likely limited to nodes in the graph that are reachable from the constants passed as arguments in the query which, in many cases, will be much smaller than  $m$ . Furthermore, SIPs are useful for implementing and maintaining view materialization - this is a pre-computation of rule evaluations that generalizes the concept of catching suggested in [9].

There are several Datalog systems available to test implementations. Nevertheless, we are not presenting experimental evaluations since [21] already reports an evaluation and comparison of a few systems that includes experiments with rule sets with exactly the characteristics of ReBAC Datalog policies. Instead what we will do is to present the relevant results and put them in context with the experimental evaluation of a Java implementation of a subset of EHL policy evaluator reported in [25].

Since the publication of [21] there have been several new releases of the systems and the results of the experiments have been updated twice using the newer versions. The discussion below is based on the 2011 report [12]. The machine where all the experiments were conducted was a dual core 3GHz Dell Optiplex 755 with 4 gigabytes of main memory. It was running Ubuntu 7.10 with kernel 2.6.22. Although the experiments were ran using four different Datalog systems and no a single one outperformed the others in all

the evaluations, we will only report the results obtained using Ontobroker [21] since it is the system that better performed in the majority of the tests. Ontobroker is also written in Java. We start reviewing the results of evaluating the following set of rules:

$$\begin{aligned} b1(X, Z), b2(Z, Y) &\rightarrow a(X, Y) \\ c1(X, Z), c2(Z, Y) &\rightarrow b1(X, Y) \\ c3(X, Z), c4(Z, Y) &\rightarrow b2(X, Y) \\ d1(X, Z), d2(Z, Y) &\rightarrow c1(X, Y) \end{aligned}$$

The base relations that would correspond to the protection state were  $c2, c3, c4, d1$  and  $d2$ , representing atoms of the form  $\text{rel}(X, c2, Y), \text{rel}(X, c3, Y), \text{rel}(X, c4, Y), \text{rel}(X, d1, Y), \text{rel}(X, d2, Y)$ . We will discuss the results for experiments that were conducted using  $50K$  and  $250K$  randomly generated arcs from a fixed set of 1000 nodes. For the query  $a(X, Y)$ , in which both variables were free, with  $50K$  arcs the time to evaluate the query was 8.807sec. With  $250K$  the evaluation took 59.259sec. At first glance, these times do not look encouraging. Nevertheless, if in the query we bind the first argument (e.g.,  $a(1, Y)$ ) the time to answer the query with  $50K$  arcs reduces to 7msec. With  $250K$  arcs the time reduces to 21msec. Tests with the second argument bound (e.g.,  $a(x, 2)$ ) resulted in similar performance of  $50K$  arcs, but only 5msec for  $250K$ . This difference is explained by the fact that Ontobroker does query analysis and builds a cost model to decide what optimizations to use including the order to do the joint operators, the algorithm to use for the execution of each of the join operations as well as selectivity analysis. This improvement of at least three orders of magnitude shows the effect of limiting the search to reachable objects. For ReBAC, we can take the best of the times since queries  $\text{grant}(u, r)$ , will have both arguments bound.

It is difficult to make a direct comparison to the results reported in [25] for several reasons. One is that the number of arcs used in [25] is 2 orders of magnitude larger ( $30000K$ ) than for the experiments in [12]. Furthermore, in [25] the arcs were not randomly generated, and the machine was more powerful: it had 8 cores of faster CPUs and 4 times more memory. They report having averages of 37msec for the policies most similar to the program above. These 37msec are an

average over policy evaluations that could require the executions of no joints at all and up to a maximum of three joints. This is in contrast to the query  $a(.)$  that has four joins. Evaluations with larger data sets can be done but it is worth noting that database sizes do not correlate directly with time to execute queries - not only the second argument bound query evaluation ran faster for the  $250K$  set than the  $50K$ , but the time that took to run queries of the form  $b2(X, Y)$  and  $b1(X, Y)$  with one of the arguments bound using the  $50K$  set and the  $250K$  set took about the same time in each case, less than 4msec for  $b1$  and less than 20msec for  $b2$ .

[12] also reports experiments over the evaluation of transitive closure rules:

$$\begin{aligned} \text{par}(X, Y) &\rightarrow \text{tc}(X, Y) \\ \text{par}(X, Z), \text{tc}(Z, Y) &\rightarrow \text{tc}(X, Y) \end{aligned}$$

The results here are also remarkable. The largest input size consisted of 2000 nodes and 1M  $\text{par}$  arcs randomly generated. Two types of input were generated, for graphs with and without cycles. For queries with no bindings ( $\text{tc}(X, Y)$ ) the times for evaluation were 87.3sec for data with no cycles and 200.9sec for data with cycles. Binding the first argument made very little difference, 86.5sec and 197.17sec respectively. But if the second argument was bound the results were 25msec for no cycles and 16msec for data with cycles. This demonstrates the effects of the Magic set optimization that re-writes the programs to take advantage of the bound arguments and the SIP derived from the rules syntax.

[25] does not have implementation for path expressions. The observation to make is that despite of the fact that the system in [25] was specially developed for EHL its performance is not particularly better than using an off-the-shelf Datalog system that also includes regular path evaluations, giving evidence of the excellent performance of Datalog systems contrary to the belief that they are not suitable for high throughput access control implementations.

A final observation about implementations: there is a result in parallel complexity that may explain some of the experimental results for the transitive closure above. A Datalog program is called linear if

and only if each rule has at most one occurrence of the predicate in the head appearing in the body. Recall that a decision problem is in the NC complexity class if it can be solved in polylogarithmic time on a parallel computer with a polynomial number of processors. It is known that the data complexity of linear Datalog is in NC [26] and amenable to parallelization. Note that except for negation, ReBAC programs are linear. Among the optimization considered by Ontobroker is the use multiple cores and threading to parallelize query evaluation.

## 8 Final remarks

Research on access control policy languages has been extensive and logic programming has been a popular modeling choice [18, 2, 19, 16, 3]. But writing correct policies and developing correct and intuitive implementations of policy management systems are not easy tasks [10]. The attention ReBAC has received in the access control research community comes from the fact that it provides an expressive yet tractable model to intuitively capture the meaning of the “subjective” policies people may have in mind. The goal of this paper has been to show the benefits of using Datalog as a developing framework. Modeling ReBAC in Datalog is natural since Datalog is a good language to describe and talk about properties of graphs which is the essence of ReBAC. From a practical point of view there are two good reasons for choosing Datalog: Datalog specifications are easier to implement, and implementation techniques have been around for many years. These are complemented by extensive results in computational complexity which we were able to use almost directly to establish the expressibility and complexity results of ReBAC Datalog policies (and by Propositions 5.1 & 7.1, the complexity of HL and EHL policy evaluation). This does not mean that Datalog must be the syntax the policy author uses to write policies. ReBAC Datalog can be thought as target compilation language of a more user-friendly language for authoring.

There is a striking similarity between the definitions of properties and relationships in HL and the definitions of concepts and roles in Description

Logics (DL). This has been our motivation for the “meta-relation”  $\text{rel}$ , as in  $\text{rel}(O, \text{friend}, R)$ , instead of  $\text{friend}(O, R)$ . This is a typical domain-independent representation of DL roles in Datalog. Since hash indexes can be built in relation columns, accessing the related items of a particular object can be done very efficiently. There is a lot of research in the DL community to develop fast deduction algorithms for very large data sets (see, for example, [27]). Developing a ReBAC model based on one of the tractable DLs is an avenue of research worth exploring. But what is more important to note is that many advances for high throughput Datalog systems have been driven by the interest of the Semantic Web community of using Datalog-like languages for Ontology reasoning. Even if a specialized ReBAC policy evaluator is developed all the experience gained developing high throughput Datalog systems cannot be ignored and will be of tremendous impact.

**Acknowledgments** Edelmira Pasarella was partially supported by the Spanish Ministry for Economy and Competitiveness (MINECO) and the European Union (FEDER funds) under Grant Ref.: TIN2013-46181-C2-1-R (COMMAS). Jorge Lobo was partially supported by the Secretaria d’Universitats i Recerca de la Generalitat de Catalunya, the Maria de Maeztu Units of Excellence Programme and the Spanish Ministry for Economy and Competitiveness (MINECO) under Grant Ref.: TIN2016-81032-P.

## References

- [1] Krzysztof R Apt and Howard A Blair. Arithmetic classification of perfect models of stratified programs. *Fundamenta Informaticae*, 13(1):1–17, 1990.
- [2] Steve Barker. Protecting deductive databases from unauthorized retrieval and update requests. *Data & Knowledge Engineering*, 43(3):293–315, 2002.
- [3] Moritz Y Becker, Cédric Fournet, and Andrew D Gordon. Secpal: Design and semantics of a decentralized authorization language. *Journal of Computer Security*, 18(4):619–665, 2010.
- [4] Glenn Bruns, Philip WL Fong, Ida Siahian, and Michael Huth. Relationship-based access control: its expression and enforcement through hybrid logic. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, pages 117–124. ACM, 2012.
- [5] Barbara Carminati, Elena Ferrari, and Andrea Perego. Enforcing access control in web-based social networks. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):6, 2009.
- [6] Yuan Cheng, Jaehong Park, and Ravi Sandhu. Relationship-based access control for online social networks: Beyond user-to-user relationships. In *Privacy, Security, Risk and Trust (PAS-SAT), 2012 International Conference on and 2012 International Conference on Social Computing (SocialCom)*, pages 646–655. IEEE, 2012.
- [7] Yuan Cheng, Jaehong Park, and Ravi S. Sandhu. An access control model for online social networks using user-to-user relationships. *IEEE Trans. Dependable Sec. Comput.*, 13(4):424–436, 2016.
- [8] Jason Crampton and James Sellwood. Path conditions and principal matching: a new approach to access control. In *Proceedings of the 19th ACM symposium on Access control models and technologies*, pages 187–198. ACM, 2014.
- [9] Jason Crampton and James Sellwood. Relationships, paths and principal matching: A new approach to access control. *arXiv preprint arXiv:1505.07945*, 2015.
- [10] Lorrie Faith Cranor and Simson Garfinkel. *Security and usability: designing secure systems that people can use.* ” O’Reilly Media, Inc.”, 2005.
- [11] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys (CSUR)*, 33(3):374–425, 2001.

- [12] Paul Fodor, Senlin Liang, and Michael Kifer. Openrulebench: Report 2011. 2011.
- [13] Philip WL Fong, Pooya Mehregan, and Ram Krishnan. Relational abstraction in community-based secure collaboration. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 585–598. ACM, 2013.
- [14] Philip WL Fong and Ida Siahaan. Relationship-based access control policies and their policy languages. In *Proceedings of the 16th ACM symposium on Access control models and technologies*, pages 51–60. ACM, 2011.
- [15] Carrie Gates. Access control requirements for web 2.0 security and privacy. *IEEE Web*, 2(0), 2007.
- [16] Yuri Gurevich and Itay Neeman. Dkal: Distributed-knowledge authorization language. In *Computer Security Foundations Symposium, 2008. CSF’08. IEEE 21st*, pages 149–162. IEEE, 2008.
- [17] Hongxin Hu, Gail-Joon Ahn, and Jan Jorgensen. Multiparty access control for online social networks: model and mechanisms. *Knowledge and Data Engineering, IEEE Transactions on*, 25(7):1614–1627, 2013.
- [18] Sushil Jajodia, Pierangela Samarati, Maria Luisa Sapino, and VS Subrahmanian. Flexible support for multiple access control policies. *ACM Transactions on Database Systems (TODS)*, 26(2):214–260, 2001.
- [19] Lalana Kagal, Tim Finin, and Anupam Joshi. A policy based approach to security for the semantic web. *The Semantic Web-ISWC 2003*, pages 402–418, 2003.
- [20] Phokion G Kolaitis. The expressive power of stratified logic programs. *Information and Computation*, 90(1):50–66, 1991.
- [21] Senlin Liang, Paul Fodor, Hui Wan, and Michael Kifer. Openrulebench: an analysis of the performance of rule engines. In *Proceedings of the 18th international conference on World wide web*, pages 601–610. ACM, 2009.
- [22] Jorge Lobo, Jiefei Ma, Alessandra Russo, Emil Lupu, Seraphin Calo, and Morris Sloman. Refinement of history-based policies. In *Logic programming, knowledge representation, and non-monotonic reasoning*, pages 280–299. Springer, 2011.
- [23] Christos H Papadimitriou and Mihalis Yannakakis. On the complexity of database queries. In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 12–19. ACM, 1997.
- [24] Raymond Reiter. Towards a logical reconstruction of relational database theory. In *On conceptual modelling*, pages 191–238. Springer, 1984.
- [25] Syed Zain R Rizvi, Philip WL Fong, Jason Crampton, and James Sellwood. Relationship-based access control for an open-source medical records system. In *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies*, pages 113–124. ACM, 2015.
- [26] Jeffrey D Ullman and Allen Van Gelder. Parallel complexity of logical query programs. *Algorithmica*, 3(1-4):5–42, 1988.
- [27] Jacopo Urbani, Ceriel Jacobs, and Markus Krötzsch. Column-oriented datalog materialization for large knowledge graphs. In *Thirtyeth AAAI Conference on Artificial Intelligence*, pages 258–264, 2016.
- [28] Moshe Y Vardi. On the complexity of bounded-variable queries. In *Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 266–276. ACM, 1995.