ReLOG: A Unified Framework for Relationship-Based Access Control over Graph Databases

Stanley Clark, Nikolay Yakovets, George Fletcher, and Nicola Zannone(✉)

Eindhoven University of Technology, 5600 MB Eindhoven, The Netherlands
{s.clark,hush,g.h.l.fletcher,n.zannone}@tue.nl

Abstract. Relationship-Based Access Control (ReBAC) is a paradigm to specify access constraints in terms of interpersonal relationships. To express these graph-like constraints, a variety of ReBAC models with varying features and ad-hoc implementations have been proposed. In this work, we investigate the theoretical feasibility of realising ReBAC systems using off-the-shelf graph database technology and propose a unified framework through which we characterise and compare existing ReBAC models. To this end, we formalise a ReBAC specific query language, ReLOG, an extension to regular graph queries over property graphs. We show that existing ReBAC models are instantiations of queries over property graphs, laying a foundation for the design of ReBAC mechanisms based on graph database technology.

1 Introduction

Relationship-Based Access Control (ReBAC) [14] has been proposed to support the specification of policies that naturally express relationships between a requester and a resource. For example, in a healthcare scenario, a typical notion used to specify policies is 'treating physician', where a doctor (the requester) wanting access to a patient's file (the resource) must either be the family doctor or a referred specialist of the patient. This constraint can be more naturally expressed by encoding the relationships between the resource and requester rather than through ad-hoc attributes as typically done in Attribute-Based Access Control (ABAC).

In general, ReBAC policies encode graph-like access conditions and are evaluated by determining whether the specified conditions occur in the graph encoding the current state of the system [15]. Therefore, ReBAC policies can naturally be viewed as graph queries over graph databases (DBs), an increasingly popular technology leveraging known complexity results, optimisation techniques and understandable performance for the efficient and scalable querying and analysis of graph data [4]. Graph databases thus have the potential to serve as a foundation for the design general-purpose ReBAC mechanisms.

However, to date it still remains unclear to what extent graph DB technology can be applied to evaluate ReBAC policies. A lack of consensus on the precise

Published by Springer Nature Switzerland AG 2022
S. Sural and H. Lu (Eds.): DBSec 2022, LNCS 13383, pp. 303–315, 2022.
https://doi.org/10.1007/978-3-031-10684-2_17

definition of ReBAC [14] has led to the proposal of a variety of domain-specific, rather than general-purpose, ReBAC models and underlying ad-hoc implementations. These models range from specifying simple conjunctions of relationships over social networks [2,8] to variants of first-order logic [3,16] operating over graphs specified in contexts such as medical records [12]. This diversity of models and policy languages has led to significant difficulties in comparing the expressiveness of ReBAC proposals and assessing their relation to graph query languages, with subtle but important differences easily overlooked.

To address these concerns, this work presents a *unified framework* for understanding and comparing existing proposals for ReBAC in the context of graph databases. We first develop an understanding of which types of ReBAC policies should be supported. Given a classification of such policy types, we then study how existing ReBAC models fit into this space. Our unified framework provides the essential ReBAC characteristics which we use as a basis for the definition and formalisation of an abstract query language suitable for modelling and theoretically reasoning about ReBAC policies. In particular, we show how to encode the different policy types in an abstract Datalog-style property-graph query language (with some proposed extensions) to help compare their expressiveness. This language offers formal well-defined semantics and complexity results over the property-graph model (PGM) [4], a leading industry standard for modelling graph data collections. This abstract query language poses a baseline for future implementations of ReBAC using widely-used concrete query languages such as Cypher and SPARQL.

2 Background

As the baseline for our framework, we employ the property-graph model and regular property-graph queries, the de facto graph data model and abstract query language. Hereafter, we provide an overview and refer to [4] for their formal definitions.

Property-Graph Model. Let $\mathcal{O}$ be a set of objects, $\mathcal{L}$ be a finite set of labels, $\mathcal{K}$ be a set of property keys, and $\mathcal{N}$ be a set of values; all pairwise disjoint. A property graph comprises a set of vertexes ($\mathcal{V} \subseteq \mathcal{O}$) and a set of edges ($\mathcal{E} \subseteq \mathcal{O}$) encoding relationships ($\eta : \mathcal{E} \to \mathcal{V} \times \mathcal{V}$) between them. Each vertex and edge can have multiple labels ($\lambda : \mathcal{V} \cup \mathcal{E} \to \mathcal{P}(\mathcal{L})$), also called *types*, and properties ($\nu : (\mathcal{V} \cup \mathcal{E}) \times \mathcal{K} \to \mathcal{N}$). The restriction of the types of nodes and the types of edges between them forms the *graph schema*.

As an example, consider the graph in Fig. 1 showing a social network modelling users posting on forums. The graph can be defined in terms of vertexes $(1, \ldots, 7)$, each with the label `person` (e.g., `Anna`) or `forum` (e.g., `Forum 1`). `person` vertexes have *name* and optionally *date of birth* properties, while `forum` vertexes have a *name* property. The graph also encodes labelled edges between vertexes: `knows` represents bi-directional relationships between users, `likes` shows which user likes a forum, `hasMember` represents the members of a forum, and `hasModerator` shows which user is the moderator of a forum.

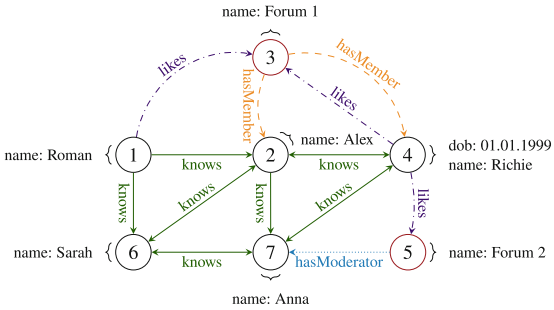


Fig. 1. Social network with nodes of type `person` (in black) and of type `forum` (in red) (Color figure online)

Regular Property-Graph Logic. Queries over the PGM are commonly specified using *Regular Property-Graph Logic* (RPGLog), a variant of non-recursive Datalog extended with transitive closure (through Kleene star operator, p^*) specialised to query graph properties [4]. An RPGLog query, or *program*, is a set of *rules* written over the graph schema, where each rule has a *head*, at least one *body* predicate and any number of *constraint* predicates. Each body predicate specifies the topology of vertexes and edges over which constraint predicates place restrictions based on property values or vertex equality. Queries over the topology are known as *basic graph patterns* (BGPs).

As an example consider the query to check the existence of a path between two people who (indirectly) know each other, with one of those people liking a forum. An RPGLog program for this query consists of a single rule, namely:

$$\texttt{result}() \leftarrow \texttt{knows}^*(x, y), \texttt{person}(x), \texttt{person}(y), \texttt{likes}(x, z), \texttt{Forum}(z)$$

3 Relationship-Based Access Control

A ReBAC model consists of three core elements [14]: *(i)* the set of information that specifies the access rights for each subject with respect to an object, or the *protection state* [12]; *(ii)* the entities involved in the authorisation decision; and *(iii)* a way to specify policies. Authorisation decisions are made by matching policies against the protection state in the context of an *access request* consisting of a requester and a resource.

ReBAC models typically propose a *policy language* over the protection state. Policies are written over the *schema* of the protection state. Based on the policy language, we can classify ReBAC models into three main classes: *simple user-oriented models*, *arbitrary graph query models* and *provenance-based models*. We study one representative model from each of the presented ReBAC model classes to derive a representative view of the entire spectrum of proposed ReBAC models.

Simple User-Oriented Models. Models in this class operate over graphs with a limited form of path conditions, have a strict syntax and often enforce requirements such as maximum path length [2,7,9]. They are well suited to regulate permissions in social networks where the protection state is a social graph representing user-to-user relationships and policies are specified in terms of simple relationships such as *friend* and *friend-of-a-friend*.

A representative ReBAC model in this class is user-to-user ReBAC (UURAC) [9], which defines a set of users connected by *multiple types* of *directed user-to-user* relationships. A UURAC policy, or graph rule, specifies the dis/conjunction of a series of paths between two users. We call this notion *node identity* where the policy is evaluated with respect to both the social graph and a function which provides the value of certain nodes, e.g. the requester or resource owner. Each path is a regular expression allowing quantification over single relationship types and associated with a mandatory maximum path length, together called a path specification. Each path specification can be negated such that the specified path should not exist within the maximum path length.

Example 1. Consider a policy allowing access to a forum with a moderator if the requester *(i)* is not (indirectly) known by the moderator in 2 hops, but *(ii)* is connected to the moderator. In UURAC, this policy can be expressed as follows:

$$(req, (\neg(\mathfrak{knows}^{-1}+, 2) \wedge (\mathfrak{T}, 1))$$

The start node *req* defines that the path is evaluated between the requester and the resource owner *owner* (the forum moderator). In terms of the graph in Fig. 1, access is granted when $req = \text{Roman}$, $owner = \text{Anna}$ and denied otherwise.

This policy is a typical example of a user-defined social network policy regulating access to content. It is *non-monotonic* [12], where adding relationships to the graph might cause policy evaluation to switch from permit to deny. In this case, if the requester would become connected to the forum moderator then their access would be revoked. The policy is *relational* [12], evaluating paths between the requester and resource owner, greatly reducing the search space and thereby guaranteeing tractability. The policy also shows that, using UURAC, it is only possible to specify paths which have a *restricted depth* due to the mandatory maximum path length. Moreover, due to the use of the so-called *no-repeated-node* semantics where a node cannot be visited twice, only paths which do not contain cycles can be found (this limitation is further discussed in Sect. 5). Importantly, the policy also specifies a path matching *any relationship*. Thus, UURAC policies can ask for arbitrary connections regardless of the current protection state which allows the specification of policies requiring that some indirect connection exists between users.

Arbitrary Graph Query Models. Models in this class support, to a large extent, arbitrary first-order logic over arbitrary labelled (property) graphs [5,11,12,16,17]. This allows the specification of policies encoding, e.g., *common friends*.

A representative model in this class is Extended Hybrid Logic (EHL) [16], which is composed of Hybrid Logic (HL) [5] extended with path conditions [11].

The hybrid logic model defining the protection state can be seen as a graph in which every node is assigned to a unique identifier. Accordingly, it is trivial to observe that the graph in Fig. 1 can be encoded as an EHL model where the nominals are the identifiers assigned to each node (i.e., $1, \ldots, 7$) such that we can refer to the node with the name Roman by the nominal 1. By adding non-user entities to the model, EHL allows specifying conditions between the requester and the resource itself instead of the resource owner as in UURAC.

Example 2. Consider a policy expressing that access to a forum is granted if the requester *(i)* knows the forum moderator not named Bob and *(ii)* knows two users also known by the forum moderator. This policy can be expressed in EHL as follows:

$$@_{\mathfrak{req}}\langle \mathfrak{knows}\rangle{\downarrow}\mathfrak{v}_1(\langle \mathfrak{hasModerator}\rangle^{-1}\mathfrak{res} \wedge \neg \mathfrak{Bob} \wedge$$
$$@_{\mathfrak{req}}\langle \mathfrak{knows}\rangle{\downarrow}\mathfrak{v}_2(\langle \mathfrak{knows}\rangle^{-1}\mathfrak{v}_1 \ \wedge @_{\mathfrak{req}}\langle \mathfrak{knows}\rangle(\langle \mathfrak{knows}\rangle^{-1}\mathfrak{v}_1 \wedge \neg \mathfrak{v}_2)))$$

which defines a regular expressions over the set of relationship labels. The $\downarrow\mathfrak{v}$ symbol 'stores' the current node for later reference with $@_{\mathfrak{v}}$ which 'jumps' to the node represented by the variable $\mathfrak{v}$, similar to $@_{\mathfrak{n}}$, which jumps to the nominal $\mathfrak{n}$. Note that path conjunction is only required to support transitive closure over arbitrary paths. Additionally, only inverse relationship labels are considered since every inverse path can be transformed into a path consisting of only inverse relationship labels [11]. In the context of Fig. 1, this policy grants access when $req = \text{Alex}, res = \text{Forum 2}$ and denies access otherwise.

This example shows that EHL supports several characteristics which makes it naturally suited for expressing policies in a variety of complex scenarios. Indeed, EHL allows distinguishing intermediate nodes in the path, which we refer to as *node binding* capabilities, and is thus able to express desirable policy types such as *k-clique* and *k-common friends* [9]. The lack of a restriction on maximum relationship depth along with the inclusion of transitive closure means that EHL is also able to express *non-finitary* policies [12] such as 'allow access if the requester and owner of the resource know each other'. EHL is also able to express a variety of *owner-checkable* policies [12].

Provenance-Based Models. Models in this class specify access control policies on a provenance graph, which relates artefacts to depict their provenance through the system [3,6,8]. The paths encoding these policies can be seen as ReBAC policies as they define access requirements in terms of nodes and relationships to traverse in the protection state.

A representative model in this class is provenance-based access control (PBAC) [3], which provides an ABAC language encoding provenance-based access constraints as path conditions evaluated through user-defined functions [13]. Path conditions are built on the Open Provenance Model. The provenance graph can be seen as an acyclic edge-labelled property graph where role labels are edge properties and the three supported relationship types (i.e., *used*, *wasGeneratedBy*, and *wasControlledBy*) are the edge labels. PBAC allows to

combine these relationships using Boolean connectors and existential and universal quantifiers. On evaluation, quantified variables are replaced with concrete node values.

Example 3. Consider the policy representing that only the owner of a resource can access it. Using PBAC, this policy can be specified as follows:

$$\exists p : wasGeneratedBy^{\circ}(res, p) \wedge wasControlledBy^{owner}(p, req)$$

where the domain of p is a set of process variables specified during policy evaluation.

An immediate observation is that the domain of quantified variables should be ascertained. Thus, each variable may have an *arbitrary variable domain*, with a mapping of variables to the values they can take, specified during policy evaluation. This characteristic allows expressing policies that, for instance, deny access to a particular concrete set of users. In addition, the inclusion of universal quantification, although not increasing the expressiveness of the language, allows specifying *all-of-type* policies with greater ease than in EHL. All-of-type policies are policies such as 'allow access if the user knows all members of a group', specifying a condition which should hold for every entity.

Discussion. Table 1 presents a classification of ReBAC models according to the types of expressible policies and the data that can be used in their evaluation. These characteristics, extracted from both the literature [1,2,9,12] and our study, represent the constraints that ReBAC models should be able to express. Notably, we identify three characteristics which are not explicitly defined in prior work: a desirable policy type *all-of-type*; a solidified notion of node identity, allowing to dynamically assign (sets of) node values at runtime; and the ability to encode the relationship depth for transitive closures of arbitrary relationships.

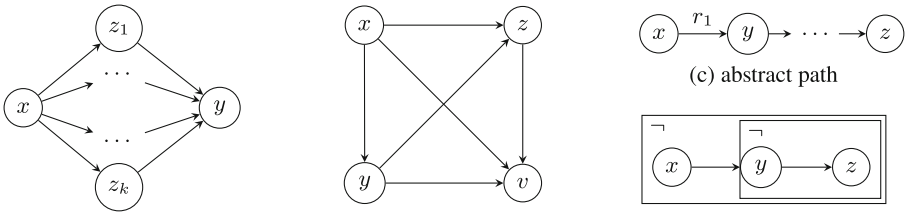
Figure 2 visualises four of the policy types that can be considered as policy templates [2] which can be extended with other features such as negation and more expressive paths. Policies encoding k-common connectors (Fig. 2a) specify that the requester and resource should have some number of common distinct connections. Those encoding k-cliques (Fig. 2b) specify that each node should be connected to every other node in the policy. Policies encoding *abstract path* (Fig. 2c) check if there is some remote pair of users n arbitrary hops away which satisfy relationship r_1 within a certain 'distance' from the user the policy is assigned to. Policies encoding *all-of-type* (Fig. 2d) specify universal quantification. We explicitly consider several characteristics pertaining to the model's data use to help exemplify that, while ReBAC policy languages make distinctions between these characteristics, graph query languages operating over the PGM make no such assumptions. A further important data characteristic is *any relationship* which ensures that in case new relationship types are added, policies will retain the same semantics compared to enumerating all relationship types in the current protection state.

Table 1. Supported ReBAC characteristics by ReBAC policy specification languages

	Characteristic	UURAC	EHL	PBAC	RPGLog
Policy Types	Common connectors [9]	-	✓	-	✓
	Clique [9]	-	✓	-	✓
	Abstract path [2]	✓	-	-	-
	Non-finitary [12]	-	✓	✓	✓
	Non-monotonic [12]	✓	✓	✓	-
	Owner-checkable [12]	-	✓	✓	-
	Relational [12]	✓	✓	✓	-
	All-of-type	-	✓	✓	-
Data Use	Restricted relationship depth [9]	✓	✓	✓	✓
	Exact relationship depth	-	✓	✓	✓
	Any relationship	✓	-	-	-
	Arbitrary variable domain	-	-	✓	-
	Node identity	✓	✓	✓	-
	Node & relationship properties [1]	-	✓	-	✓
	Object-to-Object relationships [9]	-	✓	✓	✓
	Directed relationships [9]	✓	✓	✓	✓
	Multiple relationship types [9]	✓	✓	✓	✓

The middle three columns of Table 1 give an overview of these constraints. An immediate observation is that no one ReBAC model is strictly more expressive than another. UURAC is the only language able to express *any relationship* and in turn the *abstract path* policy type. Owner-checkable policies are not expressible since policies in this model are evaluated with respect to the requester and resource owner. On the other hand, EHL is the only language which can express policies requiring to distinguish nodes, i.e. common connectors and cliques, and fully supports arbitrary properties of nodes and relationships, with PBAC only supporting a limited set of properties. While both EHL and PBAC support universal quantifiers to specify both non-finitary and non-monotonic policies, PBAC is the only language that supports an at runtime defined domain for each variable.

RPGLog, despite being a general-purpose graph query language, cannot express all desired ReBAC characteristics as shown in the last column of Table 1. Therefore, in the next sections we introduce an abstract query language which extends RPGLog to encompass the desired features and formally classify ReBAC languages in terms of these features.



(a) k-common connectors (b) k-clique (c) abstract path (d) all-of-type

Fig. 2. The main policy types supported by ReBAC

4 ReLOG: A Graph Query Language for ReBAC

In this section, we formalise ReBAC-Log (*ReLOG*), which extends RPGLog (cf. Section 2) to fully support all identified ReBAC features, thus providing a suitable language through which to compare ReBAC models (cf. Sect. 5). Supporting all identified ReBAC features in Table 1 requires the following general adjustments to RPGLog:

- any body or constraint predicate can be negated, supporting policies with *negation*;
- a query is evaluated in the context of both a graph and a mapping for a set of nodes from the graph to variables in the query, which are called *parameterised queries* and can therefore bind the requester and resource to specific nodes in the graph;
- addition of a dedicated label *any* to specify policies matching any label in the graph;
- rules used as transitive intensional (those that appear in rule's head) predicates (IDB) can only be built from other non-constraint positive body predicates such that rules using transitive closures represent *Regular Path Queries* (RPQs) matching paths conforming to a regular expression between nodes;
- non-transitive IDB predicates can be of arbitrary arity, such that variables can be compared across negated rules, so-called *cross-rule vertex references*; and
- a rule may be a constraint predicate on its own, allowing to specify *non-path* programs which test a node present as part of the parameters to a query in terms of its properties.

These extensions provide a language which finds a natural balance of expressiveness between the restrictive nature of RPGLog and the excess of pure Datalog. By remaining within the bounds of Datalog, we are able to theoretically reason about the expressiveness and known complexity results of (subsets of) Datalog which can then be directly applied to ReBAC policies [16]. The proposed extensions are supported through the formal ReLOG language syntax which is defined as follows:

Definition 1. A ReLOG rule has the form:

$$head \leftarrow body_1, \ldots, body_n, constraint_1, \ldots constraint_m$$

for some $n \geq 0$, $m \geq 0$ and $n + m > 0$. The *head* of a rule is of the form:

$$\texttt{head ::= } p(x_1, \ldots, x_m) \texttt{ | } result()$$

where $p \in \mathcal{L}$ is a label, $x_1, \ldots, x_m \in V$ are vertex variables for $m \geq 0$ appearing in a constraint or body predicate and *result* is a reserved label not in $\mathcal{L}$. By supporting heads of arbitrary arity, variables in negated predicates can be referenced in outer scopes. Each body predicate $body_i$ for $1 \leq i \leq n$ is of the form:

$$\begin{aligned} \texttt{rpq} &\texttt{ ::= } l(x, y) \texttt{ [AS } e \texttt{] | } l^*(x, y) \\ \texttt{path} &\texttt{ ::= } p(z_1, \ldots, z_j) \texttt{ | } l(x) \\ \texttt{body} &\texttt{ ::= rpq | path | } \neg\texttt{rpq | } \neg\texttt{path} \end{aligned}$$

where $x, y, z_1, \ldots, z_j \in V$ are vertex variables for some $j \geq 0$, $e \in E$ is an edge variable, and $p \in \mathcal{L}$ and $l \in \mathcal{L} \cup \{\texttt{any}\}$ are labels. Compared to RPGLog, body predicates can be negated, non-transitive predicates can have arbitrary arity, and a new reserved label $\texttt{any} \notin \mathcal{L}$ matches arbitrary edges in the graph.

Each predicate $constraint_i$ for $1 \leq i \leq m$ is of the form:

$$\texttt{constraint ::= } x.k \ \theta \ val \texttt{ | } y_1 = y_2 \texttt{ | } y_1 \neq y_2$$

where $x \in V \cup E$ are vertex or edge variables, $k \in \mathcal{K}$, $val \in \mathcal{N}$, $\theta \in \{=, \neq, <, >, \leq, \geq\}$ and $y_1, y_2 \in V$ are vertex variables. Vertex inequality is added, and variables used in a constraint predicate do not necessarily have to appear in the head of the rule, provided a runtime mapping for those variables is specified. A ReLOG query is a finite non-empty non-recursive set of rules such that at least one rule has head `result`().

To give an intuition about the semantics of a ReLOG query and the relevance of the proposed extensions, we present some example policies utilising the additional features of ReLOG compared to RPGLog (for the full formal semantics see [10]).

For instance, consider a ReBAC policy granting access to a resource if the requester is over twenty years old and is somehow connected to a user who knows at most one other user. This can be encoded in ReLOG as a non-monotonic owner-checkable policy:

$$\begin{aligned} &\texttt{result}() \leftarrow \texttt{any}(req, z), \neg\texttt{knowsTwoUsers}(z), \texttt{old}() \\ &\texttt{knowsTwoUsers}(z) \leftarrow \texttt{knows}(x, z), \texttt{knows}^*(z, y) \\ &\texttt{old}() \leftarrow req.\texttt{age} \geq 20 \end{aligned}$$

This ReLOG query uses a number of features not supported by RPGLog. The `any` predicate shows that the requester is *somehow* connected to a user. Negation is used to exclude those connections who know more than one user using a cross-rule vertex reference for the non-binary predicate `knowsTwoUsers` to bind the middle vertex z rather than x and y. Additionally, the last rule demonstrates

Table 2. RPGLog/ReLOG features required to support ReBAC policy languages

	Feature	UURAC	EHL	PBAC
RPGLog	BGP semantics - homomorphic	✓	✓	✓
	Mutual exclusion constraints		✓	
	Path semantics - arbitrary	✓	✓	✓
	Path features - $\{\epsilon, s, \overline{s}, \pi; \pi, \pi \vert \pi\}$	✓	✓	✓
	Path features - RPQ		✓	✓
ReLOG	Path negation - simple	✓	✓	✓
	Path negation - nested		✓	✓
	Parameterised queries - node	✓	✓	✓
	Parameterised queries - sets			✓
	Cross-rule node reference		✓	✓
	Any predicate	✓		
	Non-path program		✓	✓

the ability of ReLOG to encode non-path constraints, and by extension policies, only relying on the properties of nodes. To correctly encode the meaning of this policy, the ReLOG query should be evaluated in the context of a mapping where parameter *req* is mapped to the requester.

As another example, consider a scenario where multiple users collaborate in moderating forums. We can define a policy granting access to a forum if the requester is a forum moderator and knows all moderators who (indirectly) collaborate with the moderator of the requested forum via some other forums. This policy can be encoded in ReLOG as:

$$\texttt{result}() \leftarrow \texttt{person}(req), \texttt{hasModerator}(x, req), \neg\texttt{notKnown}()$$
$$\texttt{notKnown}() \leftarrow \texttt{connectedMods}^*(x, y), \texttt{hasModerator}(res, y), \neg\texttt{knows}(req, x)$$
$$\texttt{connectedMods}(x, y) \leftarrow \texttt{hasModerator}(z, x), \texttt{hasModerator}(z, y), x \neq y$$

with a mapping from the requester and resource specified in the access request to the *req* and *res* vertexes respectively. To compute `result`(), nested negation is used to encode universal quantification over *all* moderators. The unary predicate `person`(*req*) refers to the type of vertex as opposed to an IDB predicate, e.g. `knowsTwoUsers`(z) in the previous example. To compute `notKnown`(), we take the transitive closure of `connectedMods`(x, y) which encodes an RPQ $\texttt{hasModerator}^{-1}/\texttt{hasModerator}$. The transitive closure of predicates representing such RPQs in ReLOG follows *arbitrary path semantics*, meaning that there are no restrictions on repetition of nodes or edges in the matched path. Node inequality $x \neq y$ is used to impose that the rule only returns distinct moderators. This condition is necessary since ReLOG relies on *homomorphic* semantics, where two variables can be instantiated to refer to the same node. To encode that

moderators are connected *via some other forums*, in the last rule z is mapped to a set of forums determined at runtime.

5 Application of the ReLOG Framework to ReBAC Models

We investigated how the three classes of ReBAC policy languages discussed in Sect. 3 along with the underlying data model can be mapped to ReLOG. This mapping allows us to establish the features of graph query languages needed to encode the characteristics of those languages. For the lack of space, we refer to [10] for the mapping and examples and only present the key findings of our analysis. Table 2 presents the smallest subset of graph query language features needed to support each studied ReBAC model along with which of RPGLog and ReLOG supports it. This table naturally captures the essence of mapping, showing that ReLOG can express all constraints expressible by UURAC, EHL and PBAC.

ReLOG demonstrates the generality of graph query languages in encoding the characteristics of ReBAC (cf. Table 1) into distinguishing features. For instance, ReLOG uses *homomorphic* query evaluation semantics consistent with most ReBAC models (models using no-repeated-node semantics primarily do so to ensure termination of their policy evaluation algorithms [9,11]). By being able to specify BGPs and (in)equalities between nodes (so-called *mutual exclusion constraints*) policies specifying common connectors and cliques can be encoded. By supporting RPQs, ReLOG can specify non-finitary policies through path transitive closure as well as paths with restricted and exact relationship depth. Using *(nested) path negation* one can encode both non-monotonic and all-of-type policies. ReLOG's support for *parameterised queries* allows to express owner-checkable and relational policies by being able to bind variables to (sets of) values (similar to parameterised policies [2]). The inclusion of an 'any' predicate label allows to specify UURAC policies and rules can consist solely of constraint predicates which allows to write EHL policies specifying constraints based on the properties of a single vertex only.

Table 2 shows that no one policy language is contained in the other. Some example policies from non-overlapping fragments of the policy languages are:

- $PBAC \setminus EHL \cup UURAC$: Two users know a user in a runtime assigned set of users;
- $EHL \setminus PBAC \cup UURAC$: Two users have two different friends in common;
- $UURAC \setminus PBAC \cup EHL$: Two users are connected by any relationship within 2 hops;
- $EHL \cap PBAC \setminus UURAC$: The user knows all users belonging to a particular group

where $UURAC$, EHL and $PBAC$ denote the set of queries expressible in UURAC, EHL and PBAC respectively.

6 Conclusion

In this work, we studied the feasibility of modelling ReBAC in an abstract graph query language suitable for use in off-the-shelf graph databases. To this end, we devised a unified framework for comparing ReBAC models and graph query languages. We showed how ReBAC concepts map to more general graph concepts, thereby directing a shift in research to focus on ReBAC using graph DB engines. In the future, we plan to use our framework as a tool to assess the suitability of several concrete graph query languages for evaluating ReBAC policies, e.g. Cypher and SPARQL. We also plan to study how we can apply ReBAC to support fine-grained access control in graph databases.

References

1. Ahmed, T., Sandhu, R.S., Park, J.: Classifying and comparing attribute-based and relationship-based access control. In: CODASPY, pp. 59–70. ACM (2017)
2. Aktoudianakis, E., Crampton, J., Schneider, S.A., Treharne, H., Waller, A.: Policy templates for relationship-based access control. In: PST, pp. 221–228. IEEE (2013)
3. Bertolissi, C., den Hartog, J., Zannone, N.: Using provenance for secure data fusion in cooperative systems. In: SACMAT, pp. 185–194. ACM (2019)
4. Bonifati, A., Fletcher, G.H.L., Voigt, H., Yakovets, N.: Querying Graphs. Synthesis Lectures on Data Management, Morgan & Claypool Publishers, San Rafael (2018)
5. Bruns, G., Fong, P., Siahaan, I., Huth, M.: Relationship-based access control: its expression and enforcement through hybrid logic. In: CODASPY, pp. 117–124. ACM (2012)
6. Cadenhead, T., Khadilkar, V., Kantarcioglu, M., Thuraisingham, B.: A language for provenance access control. In: CODASPY, pp. 133–144. ACM (2011)
7. Carminati, B., Ferrari, E., Perego, A.: Enforcing access control in Web-based social networks. ACM Trans. Inf. Syst. Secur. **13**(1), 6:1–6:38 (2009)
8. Cheng, Y., Bijon, K., Sandhu, R.: Extended ReBAC administrative models with cascading revocation and provenance support. In: SACMAT, pp. 161–170. ACM (2016)
9. Cheng, Y., Park, J., Sandhu, R.S.: An access control model for online social networks using user-to-user relationships. IEEE Trans. Dependable Secur. Comput. **13**(4), 424–436 (2016)
10. Clark, S., Yakovets, N., Fletcher, G., Zannone, N.: A Unified Framework for Relationship-Based Access Control over Graph Databases (2022). https://gitlab.tue.nl/stanrogo/relog-framework/-/blob/main/Unified_ReBAC_Framework_Full.pdf
11. Crampton, J., Sellwood, J.: Path conditions and principal matching: a new approach to access control. In: SACMAT, pp. 187–198. ACM (2014)
12. Fong, P.W.L., Siahaan, I.S.R.: Relationship-based access control policies and their policy languages. In: SACMAT, pp. 51–60. ACM (2011)
13. Kaluvuri, S.P., Egner, A.I., den Hartog, J., Zannone, N.: SAFAX - an extensible authorization service for cloud environments. Frontiers ICT **2**, 9 (2015)
14. Lobo, J.: Relationship-based access control: more than a social network access control model. Wiley Interdiscip. Rev. Data Mining Knowl. Discov. **9**(2), e1282 (2019)

15. Paci, F., Squicciarini, A.C., Zannone, N.: Survey on access control for community-centered collaborative systems. ACM Comput. Surv. **51**(1), 6:1–6:38 (2018)
16. Pasarella, E., Lobo, J.: A datalog framework for modeling relationship-based access control policies. In: SACMAT, pp. 91–102. ACM (2017)
17. Rizvi, S.Z.R., Fong, P.W.L.: Efficient authorization of graph-database queries in an attribute-supporting ReBAC model. ACM Trans. Priv. Secur. **23**(4), 18:1–18:33 (2020)