SN Computer Science (2021) 2:351
https://doi.org/10.1007/s42979-021-00684-8

ORIGINAL RESEARCH

Extended Authorization Policy for Graph-Structured Data

Aya Mohamed[1,2] · **Dagmar Auer**[1,2] · **Daniel Hofer**[1,2] · **Josef Küng**[1,2]

Received: 22 March 2021 / Accepted: 6 May 2021 / Published online: 22 June 2021

Abstract
The high increase in the use of graph databases also for business- and privacy-critical applications demands for a sophisticated, flexible, fine-grained authorization and access control (AC) approach. Attribute-based access control (ABAC) supports a fine-grained definition of authorization rules and policies. Attributes can be associated with the subject, the requested resource and action, but also the environment. Thus, this is a promising starting point. However, specific characteristics of graph-structured data, such as attributes on vertices and edges along a path from a given subject to the resource to be accessed, are not yet considered. The well-established eXtensible Access Control Markup Language (XACML), which defines a declarative language for fine-grained, attribute-based authorization policies, is the basis for our proposed approach—XACML for Graph-structured data (XACML4G). The additional path-specific constraints, described in graph patterns, demand for specialized processing of the rules and policies as well as adapted enforcement and decision-making in the access control process. To demonstrate XACML4G and its enforcement process, we present a scenario from the university domain. Due to the project's environment, the prototype is built with the multi-model database ArangoDB. Finally, compliance of XACML4G with quality standards for access control systems administration and enforcement is assessed. The results are promising and further studies concerning performance and use in practice are planned.

Keywords Authorization policy · Access control · ABAC · XACML · Graph database · ArangoDB

Introduction

The amount of data in IT systems is still growing exponentially. Besides the amount, the value of data is increasing as well [1]. Enterprises, public services, public and private organizations as well as individuals are highly interested not to risk this value—more than that lost or stolen data could be used for harmful and damaging activities. Consequently, data must be protected and access to data must be controlled. This control has to be as close as possible to the data itself, with no bypassing options.

More and more data are stored in graph databases today. Because of their natural and direct support of connected data objects (e.g., identity and access management, social networks, and recommendation systems) and the increasing expectations in knowledge graphs, graph databases are considered to have the potential to replace the existing relational market by 2030 [2].

As graph databases are continuously entering business- and privacy-critical application domains, flexible authorization and fine-grained *access control (AC)* are increasingly necessary. For now, established graph database systems such as *Neo4j* [3] or multi-model database systems such as *Microsoft Azure Cosmos DB* [4] and *ArangoDB* [5] provide *role-based access control (RBAC)*, which is not sufficient for our demands to apply fine-grained constraints on vertices and edges.

This article is part of the topical collection "Future Data and Security Engineering 2020" guest edited by Tran Khanh Dang.

✉ Aya Mohamed
aya.mohamed@jku.at

Dagmar Auer
dagmar.auer@jku.at

Daniel Hofer
daniel.hofer@jku.at

Josef Küng
josef.kueng@jku.at

[1] Institute for Application-Oriented Knowledge Processing (FAW), Johannes Kepler University Linz (JKU), Linz, Austria

[2] LIT Secure and Correct Systems Lab, Linz Institute of Technology (LIT), Johannes Kepler University Linz (JKU), Linz, Austria

By now, authorization policies and access control mechanisms that support graph characteristics such as patterns on access paths are still missing. Therefore, access control in graph databases needs to be enhanced.

This work provides a solution for access control in graph databases from policy specification to enforcement of the proposed authorization policy language in graph databases based on the *eXtensible Access Control Markup Language (XACML)*. This solution is demonstrated in a proof of concept prototype and analyzed based on quality assessment measures introduced by the National Institute of Standards and Technology (NIST). Specifically, the contributions of our work are the following:

1. The *eXtensible Access Control Markup Language for Graph-structured data (XACML4G)*, a policy language based on the XACML structure (described in the JSON format) for expressing authorization policies for graph-structured data. XACML4G allows to describe patterns in terms of constraints on vertices and edges.
2. An initial implementation to enforce XACML4G in the data layer.
3. A proof of concept prototype in the university domain, i.e., professors, students and courses.
4. An assessment of XACML4G administration and enforcement using the National Institute of Standards and Technology (NIST) standard quality metrics for access control systems. The assessed properties are classified according to their importance (critical, optional, and supplementary) for our use case.

The rest of the paper is structured as follows. The next section provides details concerning access control models, especially attribute-based access control and the well-established policy language XACML and its architecture. They are the basis for the authorization and access control approach for graph databases developed within our research presented in the following section. The proposed policy definition language, XACML4G, and the enforcement process are demonstrated by a proof of concept prototype in the university domain in the next section. An assessment for XACML4G is presented in the following section with respect to the administration and enforcement quality metrics of access control systems from NIST. The paper concludes with a summary and an outlook on future work in the last section.

Related Work

The focus of our work is on highly flexible, fine-grained authorization policies for graph-structured data and their enforcement in a graph database. As authorization policies are an established means to allow for flexibility and separation of concerns, policies are one of the driving forces for including the following approaches into this discussion. An authorization policy defines the regulations, which determine whether a requested access can be granted or not. Policies are applied in the access decision-making during the enforcement process.

Regarding graph databases, we do not go into details here. For the basic concept, it is sufficient to consider a graph as a set of vertices which can be related in pairs to each other by edges. Both vertices and edges are distinct entities with attributes. When traversing a graph, a certain path is paced [6].

In the following, we concentrate on potentially suitable authorization models such as *attribute-based access control (ABAC)* and *relation-based access control (ReBAC)* before discussing the *eXtensible Access Control Markup Language (XACML)* that provides a policy definition language, along with the architecture and processing model for handling access requests.

Authorization Models

Due to the focus on graph-structured data, the flexible ABAC model and the ReBAC model, which specializes in relationships, are studied in more detail.

Attribute-Based Access Control (ABAC)

The attribute-based access control is a flexible authorization model, based not only on arbitrary attributes of the subject (i.e., the user) and the requested resource, but also on action attributes and environmental conditions (e.g., time, device, location). It allows for fine-grained definition of access rights as rules. Access is granted or denied by evaluating the attributes against these authorization rules. No relationship between subject and resource is needed. The rules are typically defined in policies. Thus, ABAC is often characterized as policy-based. Defining and managing the policies independently from the application, i.e., externalizing authorization, makes it much easier to coordinate access rights with dynamically evolving IT systems and authorization scenarios [7, 8].

Even though ABAC is more complex than many other authorization models, it is considered to be the most robust, scalable and flexible one in practice. Compared with the widely spread RBAC model [9], it not only avoids a huge number of difficult to manage roles, but especially problems due to overlapping roles with contradicting access right definitions. For example, if students are also teaching assistants, they are allowed to edit student's grades, but are not allowed to edit their own ones. With RBAC, these students cannot be denied to edit their own grades. However, ABAC supports such ownership scenarios [10].

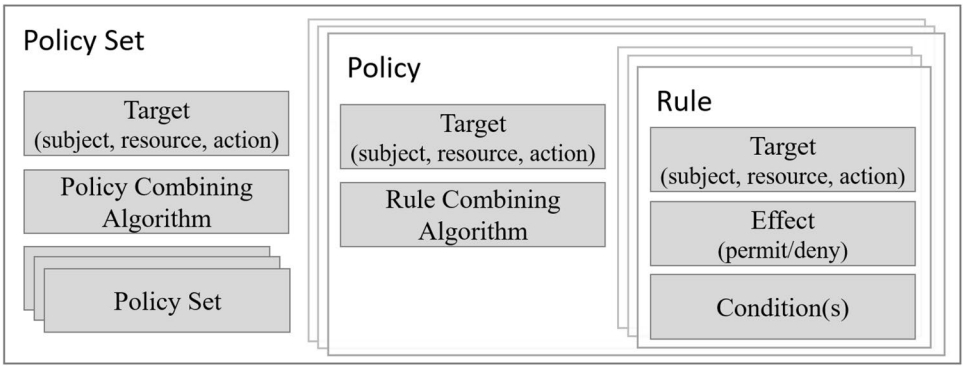

Fig. 1 XACML policy structure

With ABAC, relationships between the subject, resource, action, and environment can be flexibly defined. Therefore, ABAC is well suited for the flexible, fine-grained definition of access rights required in this work. However, it does not support path patterns from the subject to the resource, which is relevant for the access decision for some of our authorization scenarios (see "Demonstration Case").

Relation-Based Access Control (ReBAC)

To deal with access rights in social networks, which can be naturally described by graph-structured data, the relation-based access control model (ReBAC) has been developed [11, 12]. It focuses on interpersonal relationships between users, where permissions are modeled as relations between subject and object classes. However, ReBAC does not consider fine-grained access rights unlike ABAC.

Cheng et al. [13] published a policy specification in 2014 that integrates ABAC with ReBAC as an enhancement to their user-to-user relationship-based access control model (UURAC) [14], which uses regular expressions to describe relations and their characteristics (e.g., type, depth, and trust value), and the subsequent extensions for user-to-resource and resource-to-resource relationships in [15]. However, this model can cause privacy issues resulting from end-users specified policies [16].

As conditions on the path between subject and resource are also missing with the ReBAC extensions, we take the very flexible ABAC model as the basis for our further work.

eXtensible Access Control Markup Language (XACML)

While XACML stands for eXtensible Access Control Markup Language, it is not only a language, but also an architecture and processing model of how to evaluate access requests. XACML is an established OASIS[1] standard and widely used for ABAC. XACML has a strong and active community that continuously works on enhancements since the first version was approved in 2003.

Policy Language

It defines an XML-based declarative access control policy language with focus on fine-grained, attribute-based access control policies. It is hierarchically structured (see Fig. 1) with three main levels: rule, policy, and policy set [7, 17].

- **Rule**: is the basic building block. Each rule holds a target, an effect, and one or more conditions. All rule elements are optional except for the rule effect. The effect determines whether access to an object is granted or denied.
- **Policy**: is structured according to a *Composite Pattern* [18] and contains one to multiple rules which are combined according to a predefined or user-defined rule-combining algorithm [19].
- **Policy Set**: is a collection of policies and policy sets (i.e., the composite element). The policies and policy sets are combined according to the defined policy-combining algorithm.

The target specifies the component's subject (who), action (what), and resource (which) by giving attributes and literals to compare with. The definition of the target is optional with all three components. It is a kind of filter to specify the relevant target. If the target is not defined in the rule, the one of the surrounding policy will be used, the same is true for the policy and the policy set. Defining no target at all indicates that the rule is relevant for all access requests.

Since policies within the same policy set and rules inside a policy can return different decisions, the overall result is determined by combining the single decisions with a

[1] Organization for the Advancement of Structured Information Standards, www.oasis-open.org.

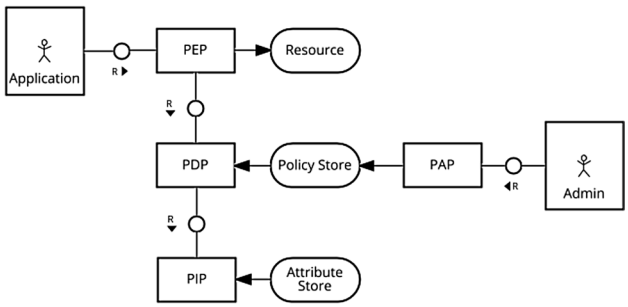

Fig. 2 XACML architecture

combining algorithm [20]. The following list provides some basic combining algorithms which are mostly implemented on policy level (rule-combining) as well as on policy set level (policy-combining) [21]:

- **Permit/Deny overrides:** with *permit overrides* the decision is permit if any of the rules or policies returns permit. The same is true for *deny overrides*, the safest combining algorithm.
- **First applicable:** the first decision taken is the overall decision.
- **Only one applicable:** is only valid for combining policies and policy sets, but not for rules. A valid output is only achieved if exactly one of the children either returns the decision *Permit* or *Deny*.
- **Ordered permit/deny overrides:** like the Permit/Deny overrides algorithm, but takes the order of rules, policies, and policy sets into account.
- **Permit unless deny/Deny unless permit:** guarantees that either *Permit* or *Deny* decision is returned. *Not Applicable* and *Indeterminate* are excluded.

Authorization requirements are specified in authorization policies using the policy definition language of XACML. These policies are used with the XACML architecture to determine the result of the access control request. This allows for a high level of flexibility as they are not implemented within the source code.

Architecture

The XACML architecture supports separation of concerns, so that authorization policies are managed independently of their application in access control. It consists of several functional components (see FMC[2] diagram in Fig. 2), so-called points [7, 22]:

- **Policy Administration Point (PAP)**: is the component to create, modify, and distribute policies.
- **Policy Decision Point (PDP)**: decides about access by evaluating and issuing authorization decisions. The outcome is one of the four decisions: *Permit*, *Deny*, *Indeterminate* (if it cannot be decided, e.g., in case of error or missing values), and *Not Applicable* (if the request is not supported by the policy).
- **Policy Information Point (PIP)**: is an intermediate point between the data source, to which attribute requests are sent, and the PDP, to which the information is passed.
- **Policy Enforcement Point (PEP)**: is responsible for enforcing the authorization decisions, when an application requests access to a protected resource.

An established way to enforce XACML polices with relational databases is query rewriting. Query rewriting extends the user query with information from the authorization policy. This approach is used for example with the Virtual Private Database (VPD) mechanism introduced by Oracle [23] and the Truman model [24]. Besides the inconsistencies between user expectations and system output (e.g., unexpected incorrect query results instead of rejected user request) also decidability issues [25] pose practical problems.

For now, no applications of XACML for graph-structured data have been reported. Some research is available on using graph databases to manage XACML policies using policy graphs [26–28], but not for graph-structured data sources. The well-established XACML policy description language and its architecture are still a promising viable starting point for our approach. Therefore, XACML is the basis for the proposed extensions to be discussed in the next section.

XACML4G

The proposed idea presented in this paper is to extend authorization policies so that they can support graph-specific access control as well. The standard BPMN 2.0[3] diagram visualized by Fig. 3 is representing the enforcement process for XACML4G. The XACML architecture and process model have been used as a reference basis. Policy processing is independent of the users' access requests and database queries. Policy enforcement approaches are often based on an intermediate layer between the user and the database at the application level where the policy preprocessing takes place and the user query is rewritten to embed the access rights requirements. In our solution, the system

[2] http://www.fmc-modeling.org/notation_reference.

[3] https://www.omg.org/spec/BPMN/2.0/PDF. http://www.bpmb.de/images/BPMN2_0_Poster_EN.pdf.

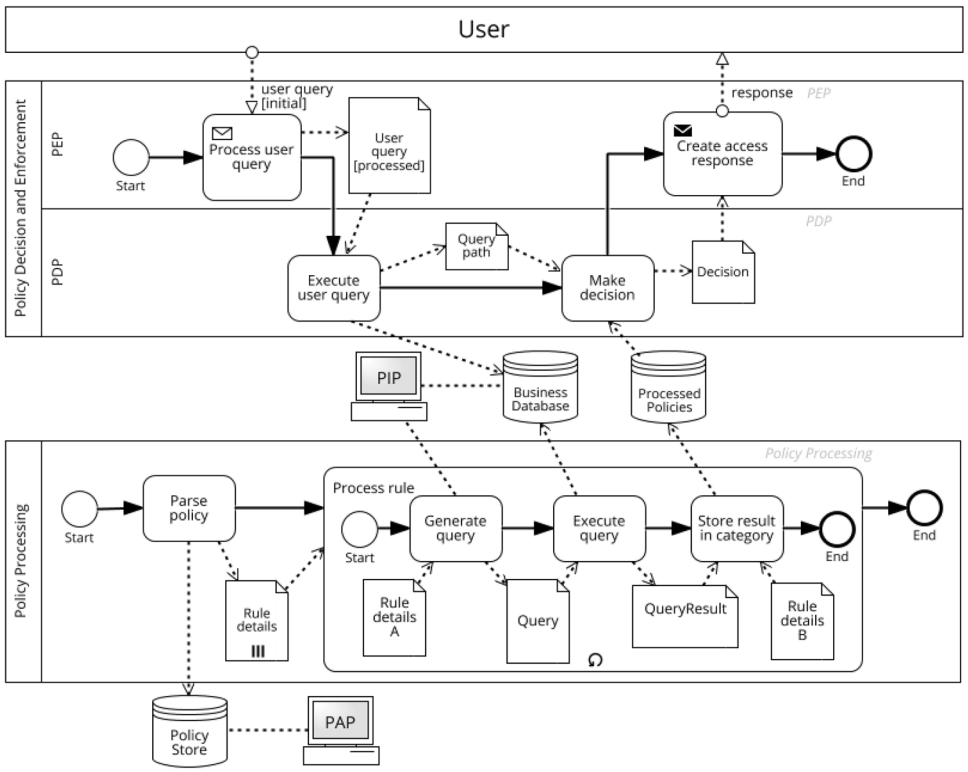

Fig. 3 Policy enforcement process

is implemented in the database layer for better performance and security. The upcoming subsections start with a motivation and definition of the problem and then address each of the components in detail.

Policy Language Limitations

Most of the current policy languages are limited to describing subjects, objects, and access rights. Relations between entities could be represented using joining conditions by mapping primary and foreign keys of the tables in relational databases. However, when it comes to specifying policies for graph databases, it will be necessary to formulate conditions on edges and vertices that are neither subject nor resource vertices. They belong to the path between them. An example of such a policy to be applied to a graph data model is given below. It demonstrates how the policy components are extracted and describes the relations between them. Thus, policy languages need to be extended to detect paths with certain patterns. Since XACML is the most commonly used language for specifying fine-grained security policies, the policy format in this work is an extension for XACML structure. For easier formulation and parsing, it is implemented in JSON instead of XML. Our approach overcomes the limitations of XACML to adapt with graph-structured data by adding an extra feature to specify the patterns, how subject and object vertices are related to each other, and conditions on the attributes of the vertices as well as edges along the path (see Example 1).

Example 1

"Professors are allowed to read data of students enrolled in their courses"

- **Subject:** professor
- **Object:** student
- **Action:** read
- **Additional constraints that could not be mapped in XACML:** To express "their courses" a pattern like "professor → course → student" is needed, which indicates that the professor and student entities are connected through a vertex of type course. Otherwise, undesired results could be returned, because of various paths, with respect to length or content, between the given subject and resource. Thus, there is a significant demand for adding path patterns in policies for graph-structured data.

Extended Policy Format

Based on the current limitations to express policies for graph-structured data, the JSON formatted authorization policy definition language XACML4G is introduced. XACML4G is based on XACML. Thus, the XACML4G policy language is structured like XACML with an additional

element to specify graph requirements such as conditions on vertices along with the in-between relations.

This pattern property is inspired by the syntax of Cypher[4] for matching patterns of nodes and relationships in the graph. Note that being well familiar with the graph model and its architecture is a requirement for policy writing. Listing 1 depicts the template used for defining the authorization policies in this work to be enforced in any graph database. The variable *op* in the subject, resource, and condition represents the relational operator for the condition statement. Each policy is composed of an identifier, rule combining algorithm, and rules list. Rules are the fundamental elements

```
{
  "policy-combining-algorithm":"first-applicable,permit/deny-
      override,etc",
  "policy":[
  {
    "id":"policy_id",
    "rule-combining-algorithm":"first-applicable,permit/deny-
        override,etc",
    "rule":[
    {
      "id":"rule_id",
      "target":{

        "subject":"entity.attribute op value",
        "resource":"entity.attribute op value",
        "action":"read/write"
      },
      "pattern":"(var1:entity{'or/and':{key:value,..}})-[var2:
          entity{key:value}]->(var3:entity{key:[value;..]})",
      "condition":"var1.attribute op var2.attribute"/
          "{'or/and':[var1.attribute op var2.attribute,..]}",
      "effect":"permit/deny"
    }]
  }]
}
```

Listing 1. Policy in JSON format

defined with a design similar to XACML rules implementing the concept of *effect*, *target*, and *condition* as explained in the following rule components list:

- **Rule ID:** unique identifier for each rule.
- **Target:** action type, subject and resource vertices are specified along with their conditions.
- **Pattern (optional):** to specify nodes, their connections, and characteristics on the attributes level. If no pattern is specified, the result is the shortest path between the subject and object vertices. Long patterns can be split and joined in the condition. The arrows in the path of the described pattern represent the direction of the edges whether inbound or outbound. If no arrow direction is indicated, all hops are considered.
- **Condition (optional):** to join vertices/edges upon certain attributes.
- **Effect:** represents the rule decision whether to grant access to the resource through the returned paths satisfying the described patterns or not.

According to the policy syntax in Listing 1, the cases for describing a node/an edge in the pattern along with its condition(s) are identified as follows:

- **Empty node/edge using empty brackets**
  e.g., ( ) → for nodes, [ ] → for edges
- **A defined entity with empty or without conditions**
  e.g., (dataObjects), (dataObjects{ })
- **A single condition with one value**
  e.g., (dataObjects{typeCode:'X'})
- **Multiple conditions joined with and/or (both operators could exist together in one entity)**
  e.g., (dataObjects{'or':{typeCode:'X',_key:'Y'},'and':{created:'Z'}})
- **A condition with multiple values separated by semicolons**
  e.g., (dataObjects{typeCode:['X';'Y']})

[4] Cypher is the declarative query language used to work with graphs in Neo4j (https://neo4j.com/docs/cypher-manual/current/introduction/).

Policy Processing

After establishing the authorization policies and converting them from text to JSON in the previous step, the procedure of policy processing is carried out in three stages: policy parsing, query generation and execution as well as classification. These components along with the decision maker belong to the PDP. First, the list of policies has to be parsed to get the parameters and filter conditions. The extracted data will serve as input (*Rule details A* in Fig. 3) to the database query which is dynamically generated to get the paths matching the designated patterns. The query execution component communicates with the database for retrieving further attributes related to subject, action, resource and environment of the request. Listing 2 shows an example of the query template in *ArangoDB Query Language (AQL)* with variables representing the inputs extracted from the policy file. The statements that correspond to the selected database query language syntax are generated accordingly.

```
FOR x in ${subjectCollection}
${subjectQueryFilter}
FOR v, e, p IN ${depth}..${depth}
ANY x GRAPH ${graphName}
${graphQueryFilters}
RETURN p.vertices[*]._key
```

Listing 2. AQL Query Template

The variable *subjectCollection* is equivalent to entity in the policy. *subjectQueryFilter* represents the subject conditions. *depth* is the length of each path in the pattern such that it is calculated from the count of the edges representing the number of hops. Finally, *graphQueryFilters* is a variable consisting of several AQL filter statements generated from the rule pattern conditions for each vertex/edge in the path including the resource vertex.

Meanwhile extracting the query requirements, the parser also saves the name of the combining algorithms that will be used to resolve conflicts not only between rules of the same policy, but also on the policies level. For each policy, the rule combining algorithm is obtained from the policy file and both are represented as key-value pair. The same is true for the policy combining algorithm, but in this case *"general"* is the key. All of the policies are identified by their id and stored as subset of the object as illustrated below.

```
{
    "general":"policy-combining-algorithm",
    "policyID":"rule-combining-algorithm"
}
```

Based on the rule effect (*Rule details B* in Fig. 3), the paths resulting from the query execution module are assigned to one of the categories, either permit or deny. In this process, the output paths satisfying the specified description in the policy are saved in a way that will also help in the next step, decision-making, as stated in the data structure below. This JSON snippet is composed of the category as a key and the value is a list of objects where each of them has a key consisting of a concatenated string of an integer index along with the id of the policy and rule to which the value, an array of the resulting paths, is associated. Each path is a list of vertices.

Table 1 Access decision for categories permutations

Access decision	Category	
	Permit	Deny
Permit	✓	×
Deny	×	✓
Not Applicable[a]	×	×
Indeterminate[b]	✓	✓

[a] Nothing is specified in the policy to decide about the access and the PEP will handle this situation

[b] Conflicting rules of the same or different policies. Rule/Policy combining algorithm (see "eXtensible Access Control Markup Language (XACML)") resolves the conflict(s) to determine the final decision

```
{
    "permit":[{"index:policyID.ruleID":[path,..]},..],
    "deny":[{"index:policyID.ruleID":[path,..]},..]
}
```

The role of the variable index in the access decision procedure will be explained in the upcoming subsection.

Decision-Making

This is the last phase in the enforcement process where the final decision is taken for the response to the user requesting access. For each access control request, the query path from the principal to the requested resource is checked against the categorized output of the policy patterns. The decision is taken according to Table 1. For instance, if the query result exists in the category *Permit* and not in *Deny*, this means that access is granted. Hence, the access decision response is *Permit*. In case of the response *Indeterminate*, conflict resolution is required.

The entire decision-making process is depicted in Fig. 4. The procedure starts with one start event, but has several end events to reflect the different possible decision-making situations: (1) without any conflict, (2) with conflicts resolved within the policies and no overall conflict resolution and finally the end point which requires (3) full conflict

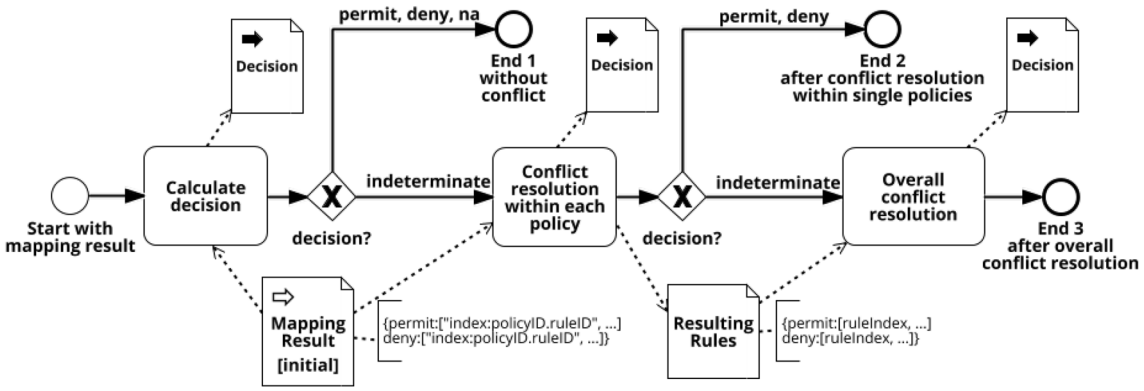

Fig. 4 Decision-making process

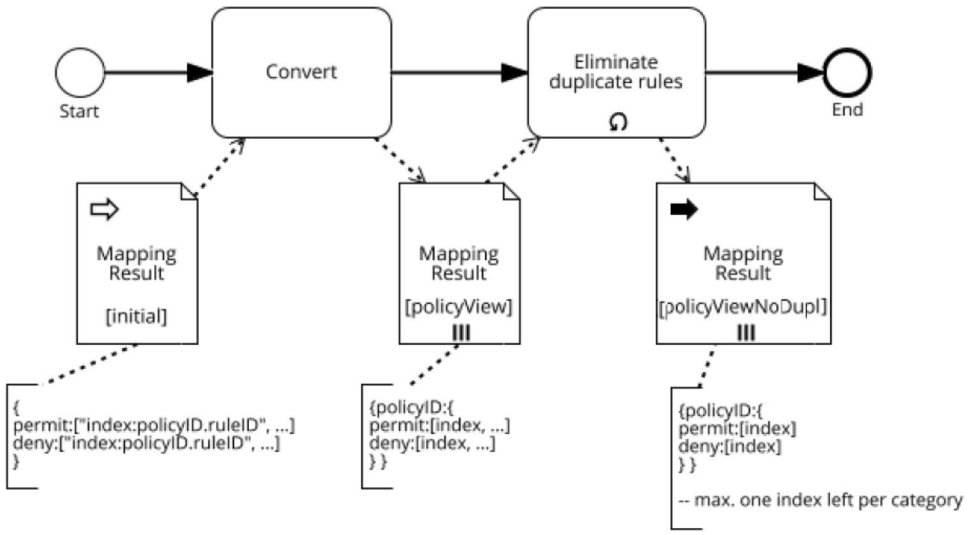

Fig. 5 Subprocess: convert and eliminate duplicates

resolution, i.e., not only within the policies but also by integrating all policies. The upcoming subsection discusses the conflict resolution on both levels in details.

To calculate the decision (as defined in Table 1), the output from checking the user query path against the specified policies needs to be processed (see Fig. 5). Thus, the mapping result (`Mapping Result [initial]`) is converted to a policy-focused view (MappingResult [policyView]), i.e., the policies are the keys instead of the categories. This conversion is crucial to differentiate between the levels of conflict, as conflicts can occur between rules within one policy, or rules across several policies, and also might result in automatic conflict resolution at one of these levels. The initial mapping result (MappingResult [initial]) is represented by a collection of categories with a list of some compound variables indicating the rule and the policy containing this input (i.e., the query path) combined with an index representing the rule order. This compound variable is used earlier in the result of the Permit/Deny Classifier as a key in the categorized list of collections (`index:policyID.ruleID`). This data structure is converted to a collection of policies having rule indices as the value (`MappingResult [policyView]`). This data structure can contain duplicates, which need to be removed before conflict resolution takes place. For each policy, duplicates are handled per category by selecting the rule with the minimum index regardless of the combining algorithm. This is to guarantee that the right access permission will be returned in case of having a first-applicable as a combining algorithm for rule and/or policy.

Until now, the JSON-formatted authorization policies are parsed to retrieve the subject, resource, action and environment conditions which are then passed to the query generator module. Upon retrieving the results which are basically paths satisfying the described patterns in the policy, they are classified and stored in the repository *processed policies* (see Fig. 3) to end up the policy processing phase. The decision

maker component checks the output of a user query against the existing paths (i.e., processed policies), which causes conflicts if the respective rules have different effects.

Conflict Resolution

A conflict occurs when the criteria of more than one rule with different effects are satisfied. For achieving a determined decision, the conflicts, regardless of their level, have to be resolved using the combining algorithm. Although the resolution method is the same for any conflict level, the conflict for each level is investigated and resolved individually according to the outcome of the check point, i.e. decision. Algorithm 1 summarizes the conflict resolution sequence as part of the decision-making process.

Algorithm 1: Decision-making Algorithm

Input: i_initial := {category:[V_1, V_2,..V_n]}, combining_algorithms = {general: policy-combining-algorithm, policyId$_j$: rule-combining-algorithm$_j$}
// where category=Permit|Deny, V="index: policyId.ruleId"
Output: decision
1: res := {permit:[ ], deny:[ ]} // res={category:[index$_1$,index$_2$,..index$_n$]}
2: decision := EvaluateDecision() // check for conflicts
3: **if** decision = indeterminate **then**
4: i_converted := ConvertInput(i_initial) /* i_converted={policyId$_j$: {category: [I_1, I_2,..I_n]}} where category=Permit|Deny, I="index" */
5: **for** i_dup in i_converted **do**
// i_reduced={policyId:{category:[index]}}
6: i_reduced := EliminateDuplicates(i_dup)
7: decision := EvaluateDecision(i_reduced)
8: **if** decision = indeterminate **then** // rule conflicts
9: i_resolved := ResolveConflict(i_reduced,combining_algorithms[policyId])
10: **end if**
11: res[i_reduced.category].add[i_reduced.category.index]
12: decision := EvaluateDecision(res)
13: **if** decision = indeterminate **then** // policy conflicts
14: decision := ResolveConflict(res,combining_algorithms[general])
15: **end if**
16: **end for**
17: **end if**
18: **return** *decision*

The first conflict resolution attempt (lines 7–10) takes place as a result of an intermediate decision made according to the strategy discussed in Table 1. Then, the winning rule index is saved in the category of the returned decision (line 11). If the evaluation decision is indeterminate, then the conflict is resolved after the completion of the post-decision-making subprocess (see Fig. 5) which is responsible for converting the output of decision-making and eliminating duplicated rules (if any) having identical effects within the same policy as indicated by Algorithm 1 in lines 4 and 6, respectively. This is performed in two steps as exemplified in Fig. 4 starting with the rules within policies followed by rules across different policies if a conflict still exists.

The conflict resolution function input parameters are the combining algorithm along with a group of collections having the category and the winning rule index after eliminating duplicates as key and value respectively. The combining algorithm is retrieved from the previously stored data structure using either the policy identifier, or *general* as a key depending on the level of conflict. Conflicts are resolved by returning *permit* in case of *permit-overrides*, or *deny-unless-permit* and *deny* for the opposite combining algorithms, i.e. *deny-overrides* and *permit-unless-deny*. For the *first-applicable* combining algorithm, the decision of the minimum rule index will be returned. Otherwise, the default access decision, *indeterminate*, will be returned.

The first conflict resolution attempt (lines 7–10) takes place as a result of an intermediate decision made according to the strategy discussed in Table 1. Then, the winning rule index is saved in the category of the returned decision (line 11). If the evaluation decision is indeterminate, then the conflict resolution process is repeated, but on the level of rules across policies (lines 12–15).

Demonstration Case

To demonstrate the applicability of the proposed concept, we present a case study. A policy for a scenario is formulated and applied.

We start with an overview of the chosen graph database and the framework used in the implementation. The description of the selected scenario along with a proper

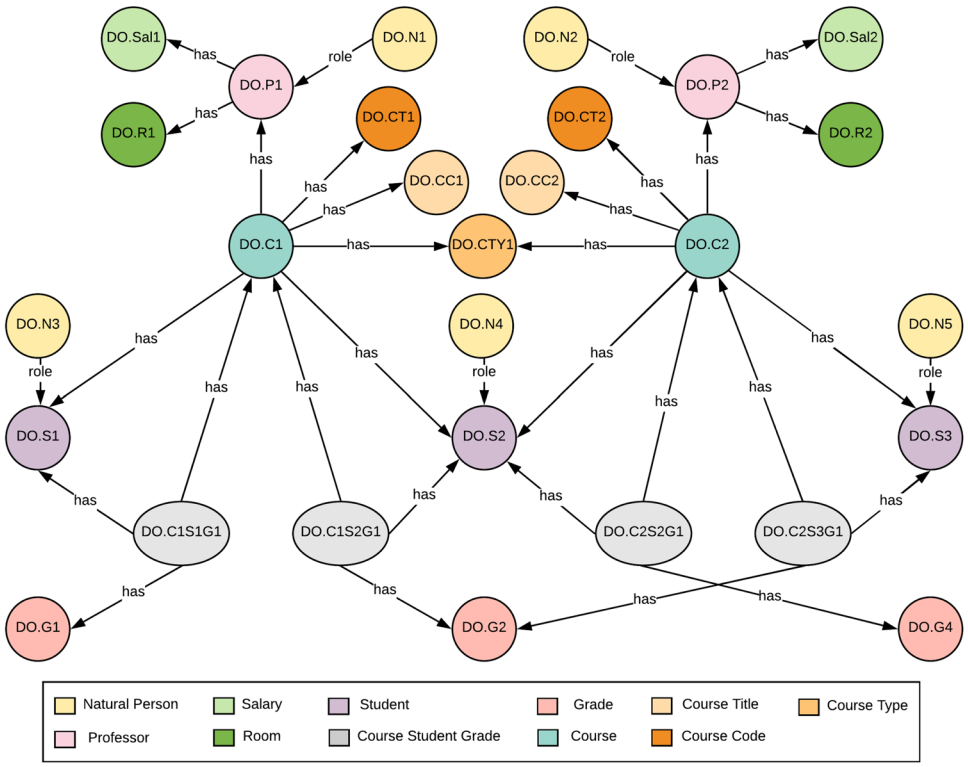

Fig. 6 Uni scenario database model

visualization for our database model follows. Finally, the results for each step are presented, from the policy description, its preprocessing, to determin the access decision for an example request. Also conflicts and special cases are considered to demonstrate the capabilities of XACML4G.

ArangoDB and Foxx Microservices

ArangoDB is the selected database management system for the prototype discussed in this paper because of several reasons. It will fit the storing requirements for any application since it supports more than one data model. Moreover, it has an application framework named Foxx that is directly communicating with the database. Finally, dealing with graphs in AQL, ArangoDB's query language, is not straight forward like in Neo4j, the leading GraphDB according to DB-Engines Ranking of Graph DBMS in March 2021.[5] Thus, if some concept proves to be viable with ArangoDB, it will be feasible with other graph databases as well.

ArangoDB is a non-relational open source multi-model database management system written in C++. It supports documents, key/value, and graphs. These schema-free NoSQL models have one database core and one declarative query language for retrieving and modifying data. Its query language AQL supports complex graph traversals, but no data definition operations including creating and dropping databases, collections, and indexes.

Foxx services are embedded inside ArangoDB and can be executed as an application directly within the database with native access to in-memory data. Furthermore, it makes ArangoDB extensible because custom HTTP endpoints are added to ArangoDB's REST API using JavaScript [29].

Model and Scenario

A university scenario is chosen for demonstrating and testing the proposed approach. The database is composed of *professors* teaching *courses* that are attended by *students* who are graded for their course(s), i.e., get *grades*. The graph constructed for this use case is illustrated in Fig. 6. As we only use one kind of vertices, so-called *dataObjects (DO)*, all instance names in this scenario start with the same abbreviation (DO) followed by the first letter(s) of the associated type and a number. The type (*typeCode*) of the instances is indicated by different colors (see legend in Fig. 6).

Initial Case

Recalling the policy example defined for this scenario (Example 1 in "Policy Language Limitations"), it can now be expressed using XACML4G. The text and the corresponding policy syntax in JSON are demonstrated in the following Example 2 bearing in mind that conditions cannot only be added on vertices but also on edges.

[5] https://db-engines.com/en/ranking/graph+dbms.

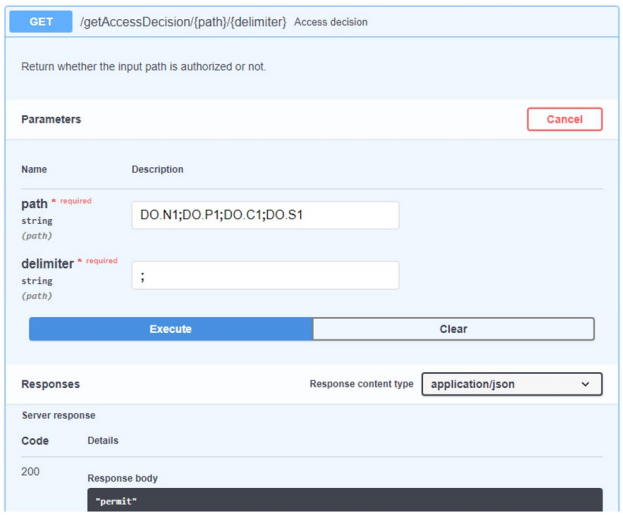

Fig. 7 Test case 1 result: permit

Example 2

"Professors are allowed to view students in their courses"

```
{
  "policy-combining-algorithm":"deny-overrides",
  "policy":[
  {
    "id":"naturalPerson_to_students",
    "rule-combining-algorithm":"first-applicable",
    "rule":[
    {
      "id":"allow_professors_students_theirCourses",
      "target":{
        "subject":"dataObjects.typeCode=='NaturalPerson'",
        "resource":"dataObjects.typeCode=='Student'",
        "action":"read"
      },
      "pattern":"(subject)-[]->(dataObjects{typeCode:
        'Professor'})<-[]-(dataObjects{typeCode:'Course'})
        -[]->(resource)",
      "effect":"permit"
    }]
  }]
}
```

According to the role of the processing procedure, the policy file is parsed and the generated query (Listing 3) is executed.

```
FOR x in dataObjects
    FILTER x.typeCode=="NaturalPerson"
    FOR v, e, p IN 3..3 ANY x GRAPH 'InstanceGraph'
        FILTER p.vertices[1].typeCode=="Professor"
        FILTER p.vertices[2].typeCode=="Course"
        FILTER v in dataObjects and v.typeCode=="Student"
    RETURN p.vertices[*]._key
```

Listing 3. The Generated Query Respective to the Policy Example

The result of this query is classified according to the rule effect and stored along with the rule and policy combining algorithms.

Two test cases exemplify the path of a user query for investigating the prototype and getting an access decision as result. The input is represented as a sequence of node keys separated by a delimiter (e.g., ";"). The decisions returned for each input are depicted in the screenshots taken from the Foxx services interface given the path and its node separator as parameters.

Case 1 DO.N1; DO.P1; DO.C1; DO.S1

Since *DO.S1* is one of the students attending the course *DO.C1* that is lectured by person *DO.N1* who has the role of professor *DO.P1*, access should be allowed (see Fig. 7) for any user query returning this path, because of the *Permit* rule effect. A *Deny* decision is returned for the same path if stated in the rule effect of the policy. If legitimate users are trying to access a resource in a non-recognized pattern, the response will be *Not Applicable (NA)*. Thus, the authorization of principal, resource, and pattern is mandatory for obtaining access.

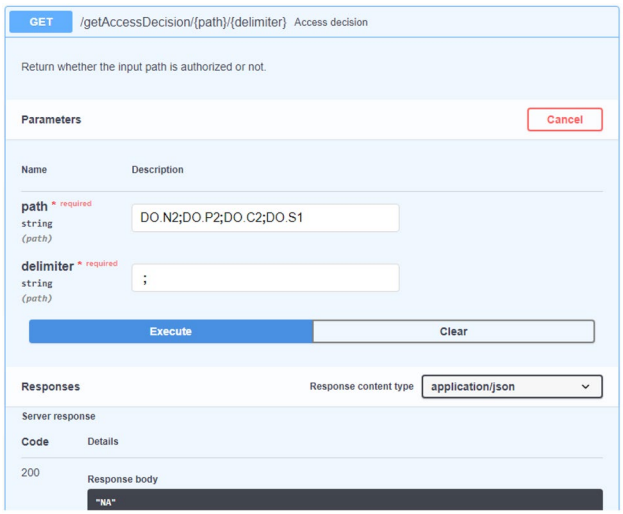

Fig. 8 Test case 2 result: NA

Case 2 DO.N2; DO.P2; DO.C2; DO.S1

Although this input path satisfies the policy pattern constraints defined for this scenario, the *NA* decision is returned as per the output in Fig. 8 because there is no such path. According to the data model in Fig. 6, there is no valid connection between the student *DO.S1* and the course *DO.C2*.

Extended Case

In the previous subsection, we presented all procedures from declaring authorization rules along with specifying path constraints to enforcing the policy and getting access decisions for a given test path. This example demonstrates the whole process, however, there are more sophisticated scenarios that actually highlight the strengths of our concept and illustrate how the special cases are handled, e.g., instances with complex path constraints (even extended beyond the resource) as well as conflicts.

Complex Constraints

Based on our graph model from the university scenario in Fig. 6 and the policy in Example 2,, the professors authorized to access the students enrolled to their courses does not imply being allowed to access the information (e.g., grade) related to this student. For instance, professor *DO.P1* is only allowed to access the grade *DO.C1S2G1* of student *DO.S2* for the course *DO.C1* and not the grade *DO.C2S2G1* of the same student for the other course *DO.C2*. The same is valid for professor *DO.P2* with student *DO.S2*. In fact, there is neither an authorization rule that can differentiate between these two paths nor a database query that can return only one of them. This is because the two paths have the same pattern in terms of node and edge characteristics as well as constraints. For a traditional policy having the professor as a subject and the student as a resource, this is an unsolved problem.

In XACML4G, it is possible to specify the pattern from the subject to the resource and extend the constraints even beyond. This is useful in our example to assure that the grade is related to the student, but also to the course of the associated professor, i.e., the subject. The rule including pattern

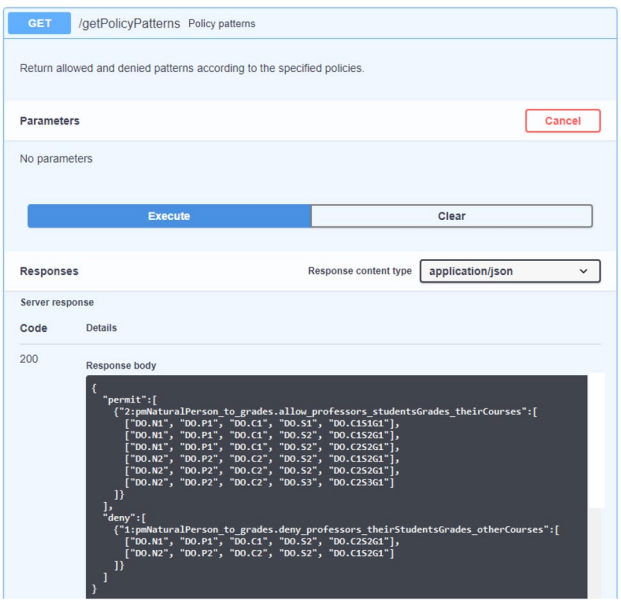

Fig. 9 Policy resulting paths

constraints beyond the resource and the joining condition relating entities via variables is depicted in Example 3.

Example 3

```
"Professors are allowed to access the grades of
 students in their courses"
```

```
{
  "id":"allow_professors_studentsGrades_theirCourses",
  "target":{
    "subject":"dataObjects.typeCode=='NaturalPerson'",
    "resource":"dataObjects.typeCode=='CourseStudentGrade'",
    "action":"read"
  },
  "pattern":"(subject)-[]->(dataObjects{typeCode:'Professor'})
    <-[]-(c1:dataObjects{typeCode:'Course'})-[]->
    (dataObjects{typeCode:'Student'})<-[]-(resource)
    -[]->(c2:dataObjects{typeCode:'Course'})",
  "condition":"c1._key==c2._key",
  "effect":"permit"
}
```

Conflict Case

Following the example described in the previous subsection, we allow professors to access a student's grade for their courses and prevent the paths for grades of this student to other courses, which are not related to the professor (i.e., the subject). This restriction can be defined with two rules. One allows all paths from the subject to the resource matching the specified pattern while the other denies specific ones using extended constraints and joining conditions. The resulting paths are grouped according to the rule effect (see Fig. 9). The rule-combining algorithm should prioritize the deny decision (e.g., *deny-overrides* or *permit-unless-deny*) to solve the occurring conflicts.

This demonstration case shows the feasibility of the proposed concept. Fine-grained attribute-based policies can be enforced for access requirements with different complexities and conflicts. Due to the size of the current test dataset, no reliable performance measures can be provided. Hence, more testing especially with more complex application scenarios and much more data are necessary. By working on the test case, additional requirements for the policy description have been identified, e.g., the idea to reuse graph patterns (iterative or recursive) and efficient handling of optional sub-patterns.

Assessment

The *National Institute of Standards and Technology (NIST)* proposed a set of quality metrics of AC systems [30] in 2006, which were used by [31] in 2012 to define respective evaluation properties. These metrics are also used to evaluate concepts, not only systems [32]. Properties need to be prioritized according to an organization's use cases and operational needs [31]. The proposed priority categories are *critical*, *optional*, and *supplemental*.

The properties in [31] are grouped into four categories: *administration*, *enforcement*, *performance*, and *support*. As XACML4G is currently a concept with a prototypical implementation in ArangoDB and not a product, we only considered administration and enforcement properties. For each of the following metrics, the assigned priority and an overview are given before providing the qualitative assessment statement.

Administration

The upcoming described properties are for evaluating access control system's administration with respect to cost, efficiency, and performance.

Auditing *(Supplemental)*

Does the AC system log granted/denied access requests, system failures and provide organization-specific log data management? Features such as providing the error source in case of system failure when processing access decisions, logging denied user requests, and tracking access with respect to the granted capabilities should be available.

– No. Auditing is irrelevant when evaluating a concept since it is product-specific and not AC model-specific. Thus, it is not considered with XACML4G right now as we provide a prototype implementation of the concept.

Privileges/Capabilities Discovery *(Supplemental)*

Can capabilities, system states, objects (or object groups) and environment variables of a given subject (or subject group) be discovered from assigned privileges/constraints?

Furthermore, the discovery of the subjects (or subject groups), system states, and environment variables for a given capability/object (or object group) is considered.

– No. Privileges and capabilities cannot be discovered in the current XACML standard, thus also not for XACML4G. A supplemental service could be considered in future work, based on the dynamic mapping of rule constraints to query filters for a given subject and object.

Ease of Privilege Assignments *(Optional)*

Considers the number of steps required for assigning, changing, and removing privileges as well as subject capabilities/object entries along with their groups and relations into the system. The more steps are needed to perform these tasks, the higher is the error rate—due to human or system mistakes.

– Yes. In XACML4G, policies are described in *JSON* syntax and can be immediately processed in our prototype, which is implemented as a built-in service running on top of *ArangoDB*, while writing and changing XACML policies is a challenging task. Moreover, the policy is described in the XML syntax and needs to be published in the infrastructure. Thus, the policy needs be explicitly re-published in the case of modifications.

Syntactic and Semantic Support for Specifying AC Rules *(Critical)*

Can rules be specified using logical expressions and/or by some programming language?

– Yes. XACML defines its own declarative AC policy language supporting syntactic as well as semantic grammar required for specifying AC policies. Although, this language uses the boolean logic relations (e.g., AND, OR, <, =, and >) for describing complex policy constraints, it is only applicable for XACML AC systems. XACML4G additionally provides a syntax to group all the policies and their rules in one file besides expressing complex path patterns along with their constraints and conditions, which could be joined and compared using the logic operators.

Policy Management *(Supplemental)*

Measures the ability to manage the AC policy life cycle including activation/deactivation, deployment verification, expiration date, and runtime change [33].

– No. Neither XACML nor XACML4G manage AC systems with respect to these capabilities.

Delegation of Administrative Capabilities *(Supplemental)*

Is it allowed to delegate policy administration, i.e., transfer the privileges from an administrator to other administrators easily and securely?

– No. This property is not supported in the current XACML4G prototype. *XACML Administration and Delegation Profile* [34] is a specification published by *OASIS* for providing administration and delegation features to XACML policies.

Flexibilities of Configuration into Existing Systems *(Supplemental)*

Can the AC mechanism be enforced by different parts of the system, e.g., the operating system, a microkernel, an application, or a client/server communication protocol?

– No. This kind of flexibility is not considered with the XACML architecture [31] and hence, XACML4G. Our prototype is an add-on to the database for high performance and reliability issues.

The Horizontal Scope (Across Platforms and Applications) of Control *(Optional)*

Can the AC system cover only a single host or also multiple hosts in a network or even virtual communities?

– Yes. One of the major advantages of XACML AC systems is the externalization of authorization such that policies are defined and managed independently from the application(s). XACML4G is mainly targeting graph data, local or distributed across server machines, managed by single AC system.

The Vertical Scope (Between Application, DBMS, and OS) of Control *(Optional)*

Deals with the scope of data control coverage on the level of applications, files, database (records, fields, and networks).

– No. As XACML4G is aimed at authorization policies for graph-structured data specifying path constraints. The current prototype implementation is limited to a single database, *ArangoDB* (see "Model and Scenario"). However, this property is partly supported by XACML since it is primarily an ABAC system which has an intermediate level of vertical scope.

Enforcement

The following metrics concern the efficiency of rendering AC decisions. They basically evaluate the AC system's policy enforcement techniques [31].

Policy Combination, Composition, and Constraint *(Critical)*

Represents the capability to combine authorization rules of different policies as well as policy models.

– Yes. In the underlying XACML, the model/rules combination, composition, and constraint methods are implemented in the PAP. Regarding XACML4G, combining contents of different policies as well as path patterns specified in the rules are considered.

Bypass *(Supplemental)*

Can policy rules be bypassed for critical access decisions and is this tolerable? This supplementary service must be used for emergency only and the risk should be tolerable in this case.

– No. The XACML and XACML4G frameworks are not designed to allow bypassing. However, critical situations can be handled with the rules by specifying environmental conditions.

Least Privilege Principle Support *(Optional)*

Can the system specify the minimum access rights required for performing a task and enforce the least privilege principle with respect to the level of granularity, flexibility, scope, and different groupings of the controlled objects?

– Yes. The ABAC model enforces an intermediate level of the least privilege principle as access rights and constraints can be specified using attributes [32]. Thus, XACML4G supports this property the same way as for XACML.

Separation of Duty (SoD) *(Critical)*

Ensures that access is only granted to subjects that are duty-related to the objects to limit power and avoid conflict of interests. The level of supporting SoD varies. Static, dynamic, and historical are the three basic types of SoD. Static and dynamic SoD differ in such that the former always applies and the latter is session-based. In the historical SoD, previous accesses determine the future authorization. To determine this metric, the number of supported SoD types as well as the steps needed to separate subjects and objects can be counted.

– Yes. XACML and hence, XACML4G is based on ABAC which has high levels of SoD, because it specifies access rights in terms of attributes and assigns them to certain principles [32]. In XACML4G, this property is critical in the context of the university scenario (see "Demonstration Case") to differentiate between conflicting roles and activate only one at a time (e.g., a person with the roles professor and student).

Safety (Confinements and Constraints) *(Critical)*

Are safety checks in permissions (e.g., via constraints) considered to prevent leaking of permissions? This property can be measured by the number of supported types of safety constraints or operational steps needed to build a certain safety constraint.

– Yes. The ABAC model enforces safety at high levels [32]. XACML is flexible and trustable in expressing and applying constraints on subjects and resources. XACML4G can additionally describe path pattern constraints.

Conflict Resolution or Prevention *(Critical)*

Can conflicting rules, deadlocks be prevented or resolved with rules stemming from the same but also from different policies?

– Yes. XACML provides conflict resolution by specifying the rule and policy combining algorithms. XACML4G additionally implements an algorithm for resolving conflicting paths (see Section 3.5).

Operational/Situational Awareness *(Optional)*

Specification and enforcement of access rules should also consider operational/situational factors. Thus, these factors (e.g., some environment variables) need to be considered in decision making.

– Yes. Similar to XACML, environmental conditions and attributes can be expressed in XACML4G policies.

Granularity of Control *(Critical)*

Can different granularity levels of objects such as data fields and endpoint system components (e.g., servers, workstations, routers, databases, or even cross domain systems) be supported by the system?

Table 2 XACML4G assessment overview

Metrics	Priority	XACML4G
Administration		
Auditing	Supplemental	×
Privileges/capabilities discovery	Supplemental	×
Ease of privilege assignments	Optional	✓
Syntactic and semantic support for specifying AC rules	Critical	✓
Policy management	Supplemental	×
Delegation of administrative capabilities	Supplemental	×
Flexibilities of configuration into existing system	Supplemental	×
The horizontal scope of control	Optional	✓
The vertical scope of control	Optional	×
Enforcement		
Policy combination, composition, and constraint	Critical	✓
Bypass	Supplemental	×
Least privilege principle support	Optional	✓
Separation of Duty (SoD)	Critical	✓
Safety (confinements and constraints)	Critical	✓
Conflict resolution or prevention	Critical	✓
Operational/situational awareness	Optional	✓
Granularity of control	Critical	✓
Expression (policy/model) properties	Critical	✓
Adaptable to AC policies implementation and evolution	Critical	✓

– Yes. XACML is selected as a starting point for our approach since fine-grained access control is supported and the application of its policies is not restricted. XACML4G is even more fine-grained in protecting graph-structured data, because it allows to define constraints on paths with respect to attributes on edges and vertices.

Expression (Policy/Model) Properties *(Critical)*

Does the AC system support a standard, language for expressing authorization rules, formal AC mechanisms, and rule/policy combination?

– Yes. XACML4G is built upon the ABAC model, provides a syntax for specifying AC rules, and implements the XACML architecture, one of the standards for defining and evaluating access policies/requests.

Adaptable to the Implementation and Evolution of Access Control Policies *(Critical)*

Evaluates the AC system with respect to its adaptability for policy changes and its capability of dynamically interposing AC rules according to the system states. Adaptability not only concerns changes in rules, but also in the applied AC mechanism (e.g., RBAC, Discretionary Access Control (DAC) [35, 36], Mandatory Access Control (MAC) [37], and Chinese Wall). Furthermore, the capability to dynamically interpose rules with respect to the entire current system state including its history is relevant.

– Yes. XACML4G, like XACML, is based on ABAC which is very flexible and can also implement other policy models like RBAC, DAC, and MAC.

Assessment Summary

To sum up, NIST's evaluation metrics for access control systems have been taken as a reference to assess XACML4G. Out of the 28 standard criteria, only the 19 criteria from the two groups administration and enforcement have been assessed. As we are not evaluating a product but a concept and its prototype implementation, the metrics for performance and support are not relevant. All applied metrics have been categorized to *critical*, *optional*, or *supplemental* according to our case study requirements.

One of the reasons for selecting XACML as the basis for our approach is being complied with all critical metrics. Since XACML4G is based on XACML's architecture, process model and AC policy structure, all the critical properties are met. In addition, most of the optional properties are satisfied. The supplemental metrics are rather related to the system than to the concept, but some of them can be taken into account in the future work. Table 2 summarizes the assessment results.

Conclusion

In this paper, we proposed the authorization policy language XACML4G, based on XACML, with an extension to describe graph patterns. These graph patterns are used to define constraints on vertices and edges and further considering their attribute values. Furthermore, we showed the prototype implementation of the definition and enforcement of XACML4G policies in ArangoDB, a multi-model database. The demonstration case within the university domain shows different scenarios with varying complexity. The authorization policy is specified in XACML4G using the JSON format. The policy needs to be processed and the result is stored in the repository for processed policies. Processing means parsing the policy file, generating and executing the query for each rule to retrieve the query path(s), and clustering the resulting path(s) by the rule effect—permit or deny. Decision-making (including conflict resolution) takes place based upon the result path of the user query and the processed policy. Lastly, we present an assessment of XACML4G taking NIST's quality metrics for evaluating access control systems as a reference. These evaluation metrics are grouped into the four categories (i.e., *Administration*, *Enforcement*, *Performance*, and *Support*) and classified according to their priority (i.e., *Critical*, *Optional*, and *Supplemental*) for a given case study. We only considered the categories related to access control policy administration and enforcement due to their relevance to our scope.

The specific characteristics of graph-structured data, taking the path in the graph connecting nodes into account, can be considered in fine-grained, attribute-based authorization policies by introducing path patterns into the policy. With the demonstration case, we show that this extended policy format can be successfully enforced in a graph database. Our approach allows for more sophisticated attribute-based authorization policies than role-based access control, which is the choice in existing graph database tools (e.g., Neo4j [3, 38]). Specific authorization scenarios as presented in the demonstration can be implemented easily. The assessment of XACML4G with regard to the guidelines for access control system evaluation metrics by NIST [31] shows the all critical metrics for our use case are satisfied.

The results of the demonstration case show that the approach is promising. The use of XACML4G in graph databases is feasible for the demonstration scenarios. Still more research is needed to test this approach on a larger scale with more complex policies, data models and data sets. Furthermore, additional requirements concerning the policy definition have been identified when working on the demonstration case such as the reuse of graph patterns and allowing for optional patterns. In future work, we will deal with performance testing as well as the requirements concerning the advanced features for the policy description language and the resulting policy enforcement.

Acknowledgements The research reported in this paper has been partly supported by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria. The work was also funded within the FFG BRIDGE project KnoP-2D (Grant No. 871299).

Funding Open access funding provided by Johannes Kepler University Linz.

Declaration

Conflict of Interest The authors declare that they have no conflict of interest.

References

1. Reinsel D, Gantz J, Rydning J. Data age 2025: The digitization of the world—from edge to core. 2018; https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf.
2. Graph databases go mainstream. 2019; https://www.forbes.com/sites/cognitiveworld/2019/07/18/graph-databases-go-mainstream/#79c0f5d5179d. Accessed in 03.2021.
3. Fine-grained access control. https://neo4j.com/docs/operations-manual/current/authentication-authorization/access-control/index.html. Accessed in 03.2021.
4. Azure role-based access control in azure cosmos db. 2020; https://docs.microsoft.com/en-us/azure/cosmos-db/role-based-access-control. Accessed in 03.2021.
5. Access control in arangodb oasis. https://www.arangodb.com/docs/stable/oasis/access-control.html. Accessed in 03.2021.
6. Fletcher G, Hidders J, Larriba-Pey JL. Graph data management—fundamental issues and recent developments. Springer International Publishing AG, Gewerbestrasse 11, 6330 Cham, Switzerland 2018, ISBN 978-3-319-96192-7.
7. Hu VC, Ferraiolo DF, Chandramouli R, Kuhn DR. Attribute-Based Access Control. London: Artech House; 2018.
8. Mohan A. Design and implementation of an attribute-based authorization management system. Ph.D. thesis, Georgia Institute of Technology 2011.
9. Sandhu RS. Role-based access control. Adv Comput. 1998;46:237–86 (**Elsevier**).
10. Axiomatics: What is attribute-based access control? White Paper 2016, https://ma.axiomatics.com/acton/ct/10529/s-02c9-1707/Bct/l-0586/l-0586:3307/ct7_0/1?sid=TV2%3AmFBxh9FWI.

11. Giunchiglia F, Zhang R, Crispo B. Relbac: Relation based access control. In: 2008 Fourth International Conference on Semantics, Knowledge and Grid. 2008; pp. 3–11. IEEE.
12. Fong PW. Relationship-based access control: protection model and policy language. In: Proceedings of the first ACM conference on Data and application security and privacy. 2011; pp. 191–202.
13. Cheng Y, Park J, Sandhu R. Attribute-aware relationship-based access control for online social networks. In: IFIP Annual Conference on Data and Applications Security and Privacy. 2014; pp. 292–306. Springer.
14. Cheng Y, Park J, Sandhu R. A user-to-user relationship-based access control model for online social networks. In: IFIP Annual Conference on Data and Applications Security and Privacy. 2012; pp. 8–24. Springer.
15. Cheng Y, Park J, Sandhu R. Relationship-based access control for online social networks: beyond user-to-user relationships. In: 2012 International Conference on Privacy, Security, Risk and Trust and 2012 International Confernece on Social Computing. 2012; pp. 646–655. IEEE.
16. Servos D, Osborn SL. Current research and open problems in attribute-based access control. ACM Comput Surv (CSUR). 2017;49(4):1–45.
17. A brief introduction to XACML. https://www.oasis-open.org/committees/download.php/2713/Brief_Introduction_to_XACML.html. Accessed in 03.2021.
18. Gamma E, Helm R, Johnson R, Vlissides J, Patterns D. Elements of reusable object-oriented software. Design patterns. Massachusetts: Addison-Wesley Publishing Company; 1995.
19. Delessy N, Fernandez EB, Sorgente T. Patterns for the extensible access control markup language. In: Proceedings of the 12th Pattern Languages of Programs Conference (PLoP2005). 2005; pp. 7–10
20. eXtensible Access Control Markup Language (XACML) Version 3.0—OASIS standard. 2013;http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html. Accessed in 03.2021.
21. Brossard D. Understanding XACML combining algorithms. 2014; https://www.axiomatics.com/blog/understanding-xacml-combining-algorithms/. Accessed in 03.2021.
22. Hu VC, Ferraiolo D, Kuhn R, Friedman AR, Lang AJ, Cogdell MM, Schnitzer A, Sandlin K, Miller R, Scarfone K, et al. Guide to attribute based access control (abac) definition and considerations. NIST Special Public. 2019. https://doi.org/10.6028/NIST.SP.800-162
23. Browder K, Davidson MA. The virtual private database in oracle9ir2. Oracle Tech White Paper Oracle Corp. 2002;500:280.
24. Rizvi S, Mendelzon A, Sudarshan S, Roy P. Extending query rewriting techniques for fine-grained access control. In: Proceedings of the 2004 ACM SIGMOD international conference on Management of data. 2004; pp. 551–562.
25. Bertino E, Sandhu R. Database security-concepts, approaches, and challenges. IEEE Trans Dependable Secure Comput. 2005;2(1):2–19.
26. Ahmadi H, Small D. Graph model implementation of attribute-based access control policies. 2019; arXiv preprint arXiv:1909.09904.
27. Jin Y, Kaja K. Xacml implementation based on graph database. Proc 34th Int Conf. 2019;58:65–74.
28. Diez FP, Vasu AC, Touceda DS, Cámara JMS. Modeling xacml security policies using graph databases. IT Professional. 2017;19(6):52–7.
29. Foxx Microservices. https://www.arangodb.com/docs/stable/foxx.html. Accessed in 03.2021.
30. Hu VC, Ferraiolo D, Kuhn DR. Assessment of access control systems. Princeton: Citeseer; 2006.
31. Hu VC, Scarfone K. Guidelines for Access Control System Evaluation Metrics. National Institute of Standards and Technology, Gaithersburg, MD. https://doi.org/10.6028/NIST.IR.7874.
32. Karatas G, Akbulut A. Survey on access control mechanisms in cloud computing. J Cyber Secur Mobil 2018; pp. 1–36.
33. Sokol AW. A report on the privilege (access) management workshop 2010.
34. XACML v3.0 Administration and Delegation Profile Version 1.0. 2014; http://docs.oasis-open.org/xacml/3.0/xacml-3.0-administration-v1-spec-en.html. Accessed in 03.2021.
35. Graham GS, Denning PJ. Protection: principles and practice. In: Proceedings of the May 16–18, 1972, spring joint computer conference. 1971; pp. 417–429.
36. (US), N.C.S.C.: A guide to understanding discretionary access control in trusted systems, vol. 3. National Computer Security Center 1987.
37. Benantar M. Mandatory-access-control model. Access control systems: security, identity management and trust models 2006; pp. 129–146.
38. Role-based access control in neo4j. 2017; https://neo4j.com/blog/role-based-access-control-neo4j-enterprise/. Accessed in 03.2021.