ARGOS: Ontology Design Patterns for Governing Dynamic Data Operations in LLM-Powered Applications

Nipun D. Pathirage[1,*], Oshani Seneviratne[1] and Deborah L. McGuinness[1]

[1]*Rensselaer Polytechnic Institute, Troy, NY, USA*

Abstract

Traditional access control systems assume a deterministic, finite set of predefined data operations (actions), tightly coupling access rights to these actions. Retrieval-Augmented Generation (RAG) systems, however, connect large language models (LLMs) to enterprise databases, enabling dynamically generated, context-specific actions for which defining individual policies is both impractical and conceptually flawed. We introduce the ARGOS (Action Realignment using Graph-based Ontological Semantics), an ontology design pattern, which semantically represents actions, aligns them with structured data schemas, and governs access based on user-granted information operation boundaries. ARGOS employs a formal semantic policy framework with an extensible rule set to detect violations at the granularity of individual data operations and provide fine-grained, actionable feedback, thus moving beyond the conventional binary grant/reject paradigm toward context-aware, explainable enforcement.

Keywords

Access Control, Data Operation Control, Action Semantics, Fine-Grained Policy Enforcement, Ontology-Based Access Control (OBAC), Usage Control

1. Introduction

The proliferation of Large Language Models (LLMs) into mainstream applications highlights their transformative, often seemingly beyond human-level, capabilities. Real world applications must contend with complex and often messy databases, coupled with diverse and evolving information needs, where data privacy and security remain paramount. These environments are rarely static, as schemas evolve, data quality varies, and the queries required to satisfy operational goals often span multiple heterogeneous sources. Retrieval-Augmented Generation (RAG) systems, which enhance LLM responses by incorporating domain-specific information via an information retrieval step, are particularly affected, because they must not only navigate this complexity but also do so dynamically, generating context-specific queries on demand. This amplifies the challenge of enforcing fine-grained, semantically coherent access policies that can safeguard sensitive information while still enabling the rich, context-aware responses expected in modern AI applications. This retrieval stage grants the RAG agent direct access to databases containing potentially millions of valuable records, necessitating a provably correct mechanism to control precisely what action each agent can perform on data.

The field of access control has long addressed the challenge of controlling user access to applications. Application-layer systems (e.g., Mandatory Access Control (MAC) [1], Discretionary Access Control (DAC) [2], Role-based Access Control (RBAC) [3], Attributed-based Access Control (ABAC) [4], Ontology-based Access Control (OBAC) [5]) primarily manage access based on **finite predefined set of user actions**. Conversely, Database Management Systems (DBMS) offer robust object-level control via their Data Control Language (DCL) [6]. However, this purely object-centric approach complicates the enforcement of unified semantic policies. A single logical rule spanning multiple tables and columns requires a set of disparate DCL statements that are difficult to manage and prone to inconsistencies.

WOP at ISWC'25: 16th Workshop on Ontology Design and Patterns at the International Semantic Web Conference, Nara, Japan
*Corresponding author.
pathin@rpi.edu (N. D. Pathirage); senevo@rpi.edu (O. Seneviratne); dlm@cs.rpi.edu (D. L. McGuinness)
https://nipdep.github.io/portfolio/personal (N. D. Pathirage); https://oshani.info (O. Seneviratne); https://www.cs.rpi.edu/~dlm/ (D. L. McGuinness)
0000-0002-9313-0925 (N. D. Pathirage); 0000-0001-8518-917X (O. Seneviratne); 0000-0001-7037-4567 (D. L. McGuinness)

This fragmentation is particularly problematic for AI agent systems, which require coherent knowledge boundaries for secure operation, i.e., clearly defined semantic limits on what information the agent can access, process, or infer.

For traditional software, this dual-model approach works because applications operate over a well-defined set of actions corresponding to fixed software features. RAG-based applications, in contrast, dynamically generate actions based on user queries at runtime. This flexibility invalidates the long-standing assumption that an "action" is a predefined access request, since the space of possible operations is effectively unbounded. Defining a separate access policy for each generated action is not only impractical, it is conceptually flawed. The problem calls for decoupling access control from static action definitions and instead governing access by the semantics of actions in relation to user-defined information boundaries.

This paper introduces ARGOS, short for **A**ction **R**ealignment using **G**raph-based **O**ntological **S**emantics, which is an ontology design pattern for representing and governing access to dynamically generated actions in LLM-powered systems. ARGOS unifies the semantic representation of structured queries with the underlying data schema, enabling access policies to be expressed in terms of the meaning and scope of an action rather than its surface form. This design pattern incorporates a semantic rule set that detects violations at the level of individual data operations and returns fine-grained, actionable feedback, moving beyond the binary grant/reject model.

2. ARGOS Ontology Design Pattern

The conceptual foundation of the ARGOS ontology design pattern is a two-stage operational model that deliberately decouples static system configuration from dynamic runtime analysis. The initial stage involves establishing a formal "knowledge boundary" by applying the pattern to model both the target data source's schema and the granular access policies that govern an agent's operational rights. This pre-configured semantic environment provides the ground truth for the second stage: dynamic evaluation. At runtime, the ontology is used to lift an incoming query's syntactic structure, such as its Abstract Syntax Tree (AST), into the same semantic space as the knowledge boundary. By asserting semantic attributes onto this query representation, a reasoner can then infer the precise intent of each data operation and evaluate it against the established policies, enabling a nuanced, point-by-point validation of access rights.

2.1. Ontology

The ARGOS ontology design pattern provides a formal structure for modeling and enforcing fine-grained, semantic access control. Its common structure is based on the following core components:

- A **Schema Component** that provides a structural representation of a target data source.
- A **Policy Component** that defines the rules governing how an `Agent` may interact with entities defined in the Schema Component.
- A **Query Structure Component** that creates a semantic graph representation of a syntactic data operation query.

These components are designed to work in concert, forming a comprehensive framework for runtime evaluation. The compositional model operates by creating a "triangle of validation" between the static and dynamic elements. The **Policy Component** is statically linked to the **Schema Component** via the `controlAccessTo` property, pre-defining which data objects are subject to specific rules for a given `Agent`. At runtime, the **Query Structure Component** is also linked to the **Schema Component** as its `ReferenceNodes` are mapped to the specific schema entities being accessed. A Pallet reasoner [7] then completes the triangle by evaluating whether the semantic action requested by the `Agent` within the query structure is permissible, according to the policies associated with the referenced schema entities. This design ensures that any dynamic operation is judged against a static, verifiable set of rules grounded in the data's structure.

2.1.1. Schema Component

Our model is designed to represent the database schema for the purpose of governing access, rather than modeling the data instances contained within it. The model is composed of three primary classes—`Database`, `Table`, and `Column`—which allow for the decomposition of the relational schema into a graph of interconnected individuals. Notably, the model deliberately omits a `Row` class. This is because rows in a relational database are not accessed as distinct entities but are instead identified through value-based conditions applied to columns, such as in a `WHERE` clause (e.g., `user_id = 'user_1'`). To accurately represent relational integrity, foreign key columns are modeled as single `Column` individuals that link their respective `Table` individuals through dedicated object properties such as `primaryKey` or `foreignKey`. Conversely, columns that share the same name across different tables but are not linked by a foreign key relationship are modeled as distinct individuals to prevent semantic ambiguity. These design decisions are foundational to our approach, providing a precise yet manageable representation of the schema that directly supports our primary goal of defining and enforcing fine-grained semantic access policies.

2.1.2. Policy Component

Our policy protocol moves beyond traditional object-level control (on tables, columns, and rows) to enable control over the *semantic access* to those objects. This means we consider not only the high-level operation being performed (`hasAction`) but also the specific data usage within different scopes of that action. To achieve this, our protocol, inspired by frameworks like XACML [8], provides granular control at multiple levels: by object (`GrantLevel`), by operation (`hasAction`), and, most critically, by sub-operational scope (`hasActionScope`). The cornerstone of this approach is the `ActionScope` class, which allows us to control user actions down to an operational scope granularity. For instance, for `Read` actions, we define two crucial scopes: `ViewActionScope` and `ProcessActionScope`. This allows us to distinguish between two fundamental operations within a single SQL query: the **view** operation (e.g., in a `SELECT` clause), which directly exposes data, and the **process** operation (e.g., in a `WHERE` clause), which uses data in background computations. This operational distinction, combined with other attributes like `GrantType` (`Permitted`/`Prohibited`) and `Condition` for dynamic `RowLevel` control, forms a highly expressive policy system. While administrators can define both `Permitted` and `Prohibited` policies for ease of use, the system automatically converts `Permitted` grants into an equivalent set of `Prohibited` policies, as the reasoner operates exclusively on prohibitions. The four main attributes in our policy definition protocol, through their various configurations, are sufficient to cover all possible database access scenarios corresponding to the four primary SQL query types, as shown in Figure 2.

2.1.3. Query Structure Component

The purpose of this domain is to lift a syntactic SQL AST into a semantic, graph-based representation upon which the reasoner can operate. Here we model a customized version of the query's AST, focusing only on components relevant to access control rather than a more complete, verbose taxonomy. The transformation begins by asserting high-level semantic attributes, such as mapping a `SelectStatement` to a `ReadAction` via the `representsAction` predicate and tagging the query with the executing `Agent` using the `executedBy` predicate. Next, the ontology models the operational scope of each `Clause` using the `representsActionScope` predicate. This scope is then propagated down to individual `Reference` nodes as an `effectiveActionScope`, which precisely identifies the context in which data is being used. The most critical function of this domain, however, is to map the query's `ColumnRef` and `TableRef` nodes to their corresponding `Column` and `Table` entities in the Schema Domain. This explicit mapping between the dynamic query and the static schema creates the essential bridge across the three domains, enabling the reasoner to perform its validation.

2.2. Pattern Entities

The ARGOS pattern is formally defined by a set of core classes and properties, which are detailed below.

2.2.1. Schema Entities

The `Schema` class is the abstract root for modeling the structure of the data source. It is not intended to model data instances, but rather the containers and fields that hold data. This allows the pattern to define access control based on the semantics of the data's structure (e.g., prohibiting access to any column designated as personally identifiable information), a key distinction from value-based access control models [9].

2.2.2. Policy Entities

This set of entities is used to construct expressive, multi-faceted rules:

- `Agent`: Represents the actor, either a person or an automated entity, for whom policies are defined and whose actions are evaluated.
- `Policy`: The central class that associates an `Agent` with a set of rules. It is the primary object for defining a specific permission or prohibition.
- `ActionType`: A high-level descriptor of the data operation being performed. The base pattern proposes two types: *read* and *modify*.
- `ActionScope`: The pattern's key innovation for enabling granular control. It describes the specific context of data usage within a single `ActionType`. For instance, a *read* action is distinguished into a **view scope** (for direct data exposure) and a **process scope** (for using data in background computations like filtering or aggregation).
- `GrantType`: A simple qualifier indicating whether the policy is a permission or a prohibition. To simplify reasoning, the pattern operates on a prohibition-only basis, where permissions are converted into a set of prohibitions.
- `GrantLevel`: Defines the granularity at which a policy applies, such as at the level of a data container (e.g., a table) or a specific data field (e.g., a column).

2.2.3. Query Structure Entities

These entities are used to create a semantic representation of an incoming query:

- `SQLNode`: A generic superclass for any node within the Abstract Syntax Tree (AST) of a given query.
- `ActionNode`: A subtype of `SQLNode` that represents the primary action of the query (e.g., a `SelectStatement`).
- `ActionScopeNode`: A subtype of `SQLNode` that represents the clauses defining specific operational scopes (e.g., a `SelectClause` or `WhereClause`).
- `ReferenceNode`: The critical subtype that represents a reference to a specific data entity in the schema. These nodes are ultimately mapped back to instances in the **Schema Component**.
- `SQLNodeType`: A class for representing auxiliary attributes of query terms that carry operational significance within a specific action scope.
- `SQLNodeStatus`: An attribute asserted on a `ReferenceNode` to flag whether its use within its inferred scope constitutes a policy violation.

2.3. Potential Variations

The ARGOS pattern is a foundational framework that can be adapted. As with other ontology design patterns, such as the OWL+PROV [10] pattern, which has a "flat" representation variant, ARGOS can be modified to suit different implementation needs.

- **Flattened Policy Representation:** For domains where granular explanation is less critical, the policy structure could be "flattened." Instead of building a policy from distinct `ActionType`, `ActionScope`, and `GrantLevel` entities, these could be combined into fewer, more complex OWL class expressions or SWRL rules. This trades some modularity and reusability for potentially more concise rule definitions.
- **Alternative Query Representations:** While the pattern is described using an AST-based model (`SQLNode`), it is not strictly tied to this representation. The **Query Structure Component** could be adapted to model other query formalisms, such as a query execution plan or a more abstract logical query graph, so long as the representation can be semantically grounded and linked back to the **Schema Component**.
- **Extended Action Scopes:** The initial pattern proposes *view* and *process* as the two primary scopes for read actions. This is not an exhaustive list. For different data models or use cases, new scopes could be introduced. For example, a *metadata scope* could be defined for operations that only query the schema information itself, or an *aggregation scope* could be explicitly modeled for operations involving functions like `SUM` or `AVG`.

2.3.1. Enforcement Logic Pattern

The ARGOS ontology design pattern includes a set of enforcement rules that tie the Schema, Policy, and Query Structure components together to detect prohibited data operations. This core runtime process evaluates the query's semantic graph against the set of `Prohibited` policies applicable to the `Agent`. The rules are designed to handle the granularity introduced by the pattern's core concepts, particularly the `ActionScope` class. This allows for nuanced control by distinguishing between the different ways data can be used within a single high-level action, enabling policies to permit an action in one context while prohibiting it in another.

The fundamental logic of the rule set can be formalized as a first-order logic implication. A `ReferenceNode` within a query is marked as a violation if there exists a prohibitive policy that applies to the agent, the referenced data entity, and the specific action and scope context in which the entity is being used. This principle can be expressed as:

$$\forall r \in \text{RefNode}, \exists p \in \text{Policy} c(p, r) \rightarrow \text{hasStatus}(r, \text{Violated})$$

Where $c(p, r)$ is the conjunction of conditions that must hold for a policy p to apply to a reference node r. This conjunction asserts that: (1) the policy and the query share the same executing `Agent`; (2) the policy's `GrantType` is `Prohibited`; (3) the policy's target (via `controlAccessTo`) is the same `Schema` entity that the reference node r maps to; and (4) the policy's specified `ActionType` and `ActionScope` match the effective action and scope inferred for the reference node r.

This logical structure is not a fixed implementation but a reusable pattern for enforcement. It can be extended and specialized for different use cases by instantiating the general rule template. For example, to support a new query language, one would create specific rules for its unique node types (e.g., a `PathExpressionRef` for a graph query language) that instantiate the variables in the formal logic above. Similarly, if a user extends the pattern with custom `ActionScopes`, they would define a corresponding new violation detection rule that follows the same logical pattern. This makes the enforcement logic itself a modular and extensible component of the overall design pattern, guaranteeing that a secure operational boundary can be enforced across diverse applications.

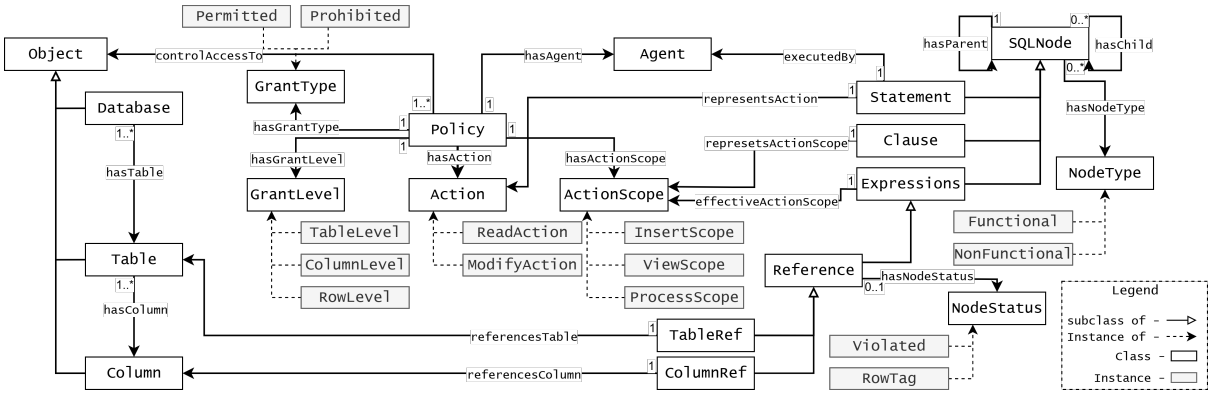


Figure 1: Overview of the instantiated ARGOS ontology for the RDB/SQL use case, showing the interconnected Schema, Policy, and SQL AST components.

ARGOS Ontology Pattern Summary

Intent: Govern dynamically generated data operations in LLM-powered systems through semantic action–schema alignment.
Context: Dynamic action spaces (e.g., LLM-generated queries) where predefined action lists are infeasible.
Problem: How to enforce fine-grained, context-aware policies without relying on fixed action enumerations.
Solution: The three-world model (Schema, Policy, Query) combined with *ActionScope* and formal enforcement logic to semantically map and validate operations.
Example: RDB/SQL instantiation illustrating granular read/process scopes and semantic policy enforcement over dynamically generated SQL queries.
Consequences: Reusability, adaptability, interpretability, and modularity. Supports adaptation to multiple query languages and policy granularities.
Known Uses: Current SQL-based use case (see Section 3); applicable to GraphQL [11], SPARQL [12], and NoSQL scenarios where query decomposition and semantic scope control are required.

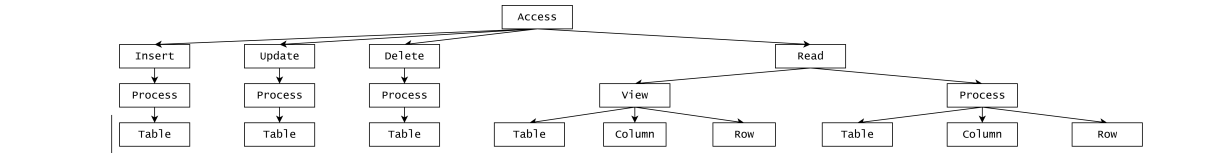
3. Use Case: Controlling Relational Database Operations via SQL

We demonstrate the application of the ARGOS ontology design pattern in a critical enterprise scenario: securing access to a relational database (RDB) queried via Structured Query Language (SQL). This domain is particularly challenging due to the intricate relationships created by data normalization and the sheer volume of enterprise data residing in RDBs, making it a critical target for secure integration. The goal is to enforce granular access policies over dynamic, unpredictable SQL queries, such as those generated by modern AI agents. For example, in a query like `SELECT name, SUM(salary) FROM employees WHERE dept_id > 10;`, a security framework must understand that ‘name‘ and ‘salary‘ are being used for viewing, while ‘dept_id‘ is used for filtering, and apply different policy rules to each context.

3.1. Ontology Design for RDB and SQL

To address this challenge, we instantiate the ARGOS design pattern to create a concrete ontology tailored for the RDB/SQL domain. This process involves specializing the abstract components of the pattern as follows:

- **Schema Component Instantiation:** The pattern’s abstract `Schema` component is implemented



Action	ModifyAction	ModifyAction	ModifyAction	ReadAction	ReadAction	ReadAction	ReadAction	ReadAction	ReadAction
ActionScope	InsertActionScope	UpdateActionScope	DeleteActionScope	ViewActionScope	ViewActionScope	ViewActionScope	ProcessActionScope	ProcessActionScope	ProcessActionScope
GrantLevel	TableLevel	TableLevel	TableLevel	TableLevel	ColumnLevel	RowLevel	TableLevel	ColumnLevel	RowLevel
GrantType	Prohibited	Prohibited	Prohibited	Prohibited	Prohibited	Conditional	Prohibited	Prohibited	Conditional

Figure 2: ARGOS policy definition protocol, showing how its four main attributes cover all action scenarios arising from the four primary SQL query types.

using concrete classes such as `Database`, `Table`, and `Column`. These classes are used to build a semantic graph representing the target RDB's schema, including its relational integrity constraints like primary and foreign keys.

- **Policy Component Instantiation:** The abstract `Policy` component is used to define access rights specific to RDB operations. Here, the generic `ActionScope` is instantiated with two crucial individuals for SQL `Read` actions: `ViewActionScope`, which corresponds to data being exposed in a `SELECT` clause, and `ProcessActionScope`, which corresponds to data being used in background computations within clauses like `WHERE` or `GROUP BY`.
- **Query Structure Component Instantiation:** The pattern's generic `SQLNode` is specialized to model a SQL query's Abstract Syntax Tree (AST). This involves creating concrete subclasses such as `SelectStatement`, `UpdateStatement`, `TableRef`, and `ColumnRef` to represent the specific syntactic elements of a SQL query.

Figure 1 provides an overview of this instantiated ontology, illustrating how the three components are interconnected to model the RDB schema, SQL-specific policies, and the query AST.

3.2. Instantiated Rule Set

The abstract `Enforcement Logic Pattern` is instantiated as a concrete set of SWRL [13] rules designed to operate on the SQL-specific ontology. These rules first perform semantic attribute assertion to lift the syntactic SQL AST into a semantic graph and then use this graph to detect policy violations. The following seven core runtime rules are designed to enforce the full spectrum of policies on SQL queries.

- **R1 (Modify Action Violation):** Detects prohibited 'Modify' actions on a table.

```
Policy(?p) ^ hasAction(?p, ModifyAction) ^ hasActionScope(?p, ?scope) ^ hasGrantType(?p,
    ↪ Prohibited) ^ hasGrantLevel(?p, TableLevel) ^ hasAgent(?p, ?a) ^
    ↪ controlAccessTo(?p, ?t) ^ Statement(?stmt) ^ executedBy(?stmt, ?a)^
    ↪ representsAction(?stmt, ModifyAction) ^ hasChildNode(?stmt, ?ref) ^ TableRef(?ref)
    ↪ ^ referencesToTable(?ref, ?t) ^ immediateChildNode(?stmt, ?cls) ^
    ↪ representsScope(?cls, ?scope) -> hasStatus(?ref, Violated) ^ relatedPolicy(?ref,
    ↪ ?p)
```

- **R2 (Table-Level Read Action, Process Violation):** Detects any reference to a table prohibited for processing.

```
Policy(?p) ^ hasAction(?p, ReadAction) ^ hasAgent(?p, ?a) ^ hasGrantType(?p, Prohibited)
    ↪ ^ hasGrantLevel(?p, TableLevel) ^ hasAgent(?p, ?a) ^ controlAccessTo(?p, ?t) ^
    ↪ Statement(?stmt) ^ executedBy(?stmt, ?a) ^ representsAction(?stmt, ReadAction) ^
    ↪ hasChildNode(?stmt, ?ref) ^ TableRef(?ref) ^ referencesToTable(?ref, ?t) ->
    ↪ hasStatus(?ref, Violated) ^ relatedPolicy(?ref, ?p)
```

- **R3 (Table-Level Read Action, View Violation):** Detects a reference to a view-prohibited table within a `ViewActionScope`.

```
Policy(?p) ^ hasAction(?p, ReadAction) ^ hasActionScope(?p, ViewActionScope) ^
    ↪ hasGrantType(?p, Prohibited) ^ hasGrantLevel(?p, TableLevel) ^ hasAgent(?p, ?a) ^
    ↪ controlAccessTo(?p, ?t) ^ Statement(?stmt) ^ executedBy(?stmt, ?a) ^
    ↪ representsAction(?stmt, ReadAction) ^ hasChildNode(?stmt, ?ref) ^ TableRef(?ref) ^
    ↪ hasEffectiveScope(?ref, ViewActionScope) ^ referencesToTable(?ref, ?t) ->
    ↪ hasStatus(?ref, Violated) ^ relatedPolicy(?ref, ?p)
```

- **R4 (Column-Level Read Action, Process Violation):** Detects any reference to a column prohibited for processing.

```
Policy(?p) ^ hasAction(?p, ReadAction) ^ hasGrantType(?p, Prohibited) ^ hasGrantLevel(?p,
    ↪ ColumnLevel) ^ hasActionScope(?p, ProcessActionScope) ^ hasAgent(?p, ?a) ^
    ↪ controlAccessTo(?p, ?c) ^ Statement(?stmt) ^ executedBy(?stmt, ?a) ^
    ↪ representsAction(?stmt, ReadAction) ^ hasChildNode(?stmt, ?ref) ^ ColumnRef(?ref)
    ↪ ^ referencesToColumn(?ref, ?c) -> hasStatus(?ref, Violated) ^ relatedPolicy(?ref,
    ↪ ?p)
```

- **R5 (Column-Level Read Action, View Violation):** Detects a reference to a view-prohibited column within a `ViewActionScope`.

```
Policy(?p) ^ hasAction(?p, ReadAction) ^ hasActionScope(?p, ViewActionScope) ^
    ↪ hasGrantType(?p, Prohibited) ^ hasGrantLevel(?p, ColumnLevel) ^ hasAgent(?p, ?a) ^
    ↪  controlAccessTo(?p, ?c) ^ Statement(?stmt) ^ executedBy(?stmt, ?a) ^
    ↪ representsAction(?stmt, ReadAction) ^ hasChildNode(?stmt, ?ref) ^ ColumnRef(?ref)
    ↪ ^ ofClauseNode(?ref, ?cls) ^ representsScope(?cls, ViewActionScope) ^
    ↪ hasEffectiveScope(?ref, ViewActionScope) ^ referencesToColumn(?ref, ?c) ^
    ↪ immediateParentNode(?ref, ?pn) ^ hasNodeType(?pn, NonFunctional) ->
    ↪ hasStatus(?ref, Violated) ^ relatedPolicy(?ref, ?p)
```

- **R6 (Row-Level Read Action, Process Tagging):** Tags a table reference for subsequent condition enforcement if it is referenced anywhere in a query and has a row-level process restriction.

```
Policy(?p) ^ hasAction(?p, ReadAction) ^ hasActionScope(?p, ViewActionScope) ^
    ↪ hasGrantType(?p, Conditional) ^ hasGrantLevel(?p, RowLevel) ^ hasAgent(?p, ?a) ^
    ↪ controlAccessTo(?p, ?t) ^ Statement(?stmt) ^ executedBy(?stmt, ?a) ^
    ↪ representsAction(?stmt, ReadAction) ^ hasChildNode(?stmt, ?ref) ^ TableRef(?ref) ^
    ↪ referencesToTable(?ref, ?t) -> hasStatus(?ref, RowTag) ^ relatedPolicy(?ref, ?p)
```

- **R7 (Row-Level Read Action, View Tagging):** Tags a table reference for condition enforcement if it appears in a `ViewActionScope` and has a row-level view restriction.

```
Policy(?p) ^ hasAction(?p, ReadAction) ^ hasActionScope(?p, ViewActionScope) ^
    ↪ hasGrantType(?p, Conditional) ^ hasGrantLevel(?p, RowLevel) ^ hasAgent(?p, ?a) ^
    ↪ controlAccessTo(?p, ?t) ^ Statement(?stmt) ^ executedBy(?stmt, ?a) ^
    ↪ representsAction(?stmt, ReadAction) ^ hasChildNode(?stmt, ?ref) ^ TableRef(?ref) ^
    ↪ hasEffectiveScope(?ref, ViewActionScope) ^ referencesToTable(?ref, ?t) ->
    ↪ hasStatus(?ref, RowTag) ^ relatedPolicy(?ref, ?p)
```

4. Evaluation

The ARGOS ontology design pattern was evaluated based on key model quality criteria introduced in Thorn's criterion [14], including correctness, reusability, changeability, formalness, and usability. The evaluation is grounded in the successful application of the pattern to the RDB/SQL use case, which provides a complex, real-world scenario for assessing its capabilities.

4.1. Correctness

Correctness is the correspondence between the model and the artifacts it represents. We evaluated the correctness of the ARGOS pattern by analyzing the representational adequacy of its instantiation in the RDB/SQL use case. To do this, we formulated 25 competency questions (CQs) designed to test whether the resulting ontology could accurately model the database schema, define granular policies, and reason over incoming SQL queries. These CQs, which cover all three components of the pattern, are enumerated below:

- **Schema Modeling (CQ 1-5):** Questions on the ability to represent tables, columns, and key relationships.
- **Policy Definition (CQ 6-17):** Questions on defining agent-specific policies with varied action types, scopes, and granularities.
- **Query Analysis & Reasoning (CQ 18-25):** Questions on modeling the query AST and identifying policy violations.

The instantiated ontology was able to be used to successfully answer all 25 CQs, demonstrating that the ARGOS pattern provides a correct and sufficiently expressive foundation for modeling the domain and enforcing access control.

4.2. Reusability

Reusability is the ability to reuse parts of the pattern when developing other models. The ARGOS pattern exhibits high reusability by design. Its core components are abstract: the `Schema Component` can model any structured data source, the `Query Structure Component` can be adapted to any query language with a parsable syntax (e.g., SPARQL, GraphQL), and the `Policy Component` is inherently generic. The successful and detailed application of this abstract pattern to the specific and complex RDB/SQL use case serves as strong evidence of its reusability in other domains.

4.3. Changeability

Changeability is the ability to evolve the model while maintaining its integrity. The modular design of the ARGOS pattern supports this. For example, a new policy can be added simply by creating a new `Policy` individual and linking it to the relevant entities, without affecting existing policies. More significantly, the pattern can be extended with new concepts. If a use case required a new type of operational context, such as a distinct `AggregationScope`, it could be added as a new individual to the `ActionScope` class, and a corresponding enforcement rule could be added to the reasoner. This could be done without altering the existing rules for other scopes, demonstrating the pattern's capacity to evolve.

4.4. Formalness

Formalness is the ability to manage the model in a formalized manner. The ARGOS pattern is specified in OWL 2 DL [15], and its enforcement logic is designed to be implemented with standards like SWRL. This grounding in W3C [16] standards ensures that models created with the pattern are machine-interpretable and can be processed by standard OWL reasoners. As described in the pattern definition, the core enforcement logic can be captured in a formal first-order logic implication, providing a precise, unambiguous specification for its implementation.

4.5. Usability

Usability refers to the ease of effectively communicating and aligning the model with users' mental models. While the concept of decomposing query semantics into distinct action scopes may be novel, the three-part structure of the pattern (Schema, Policy, Query) provides a clear and logical separation

of concerns. To further ensure the quality and usability of ontologies produced from the pattern, we validated the instantiated RDB/SQL ontology using the OOPS! (OntOlogy Pitfall Scanner!) tool [17]. The scan confirmed that the ontology is structurally robust and free of critical modeling errors, which ensures a stable and reliable foundation for developers building applications with the pattern.

5. Related Work

The application of ontologies to access control is a mature field of research. We categorize related work into three primary areas: foundational Ontology-Based Access Control (OBAC) [5] models that semanticize traditional paradigms, Usage Control (UCON) [18] models that introduce decision continuity, and modern applications that address fine-grained data control.

A significant body of work establishes the foundation for OBAC by using ontologies to enrich and formalize access control decisions [5]. Early research [19] demonstrated how to model users, profiles, and domain concepts to enable personalized and semantic access control for web services and other resources. This includes foundational models for semantic web services [20], personalizable OBAC [19], and methods for protecting semantic resources at the concept and property level. A parallel effort [21] focused on providing semantic representations of traditional models like Role-Based Access Control (RBAC) [3], showing how to map its primitives into OWL to support interoperability and create richer, context-aware role expressions. While seminal, these approaches were designed for environments with large populations of human users, where access is determined by the roles and attributes of the user. They provide the foundation but do not address the paradigm of a few powerful, non-human agents.

A second thread of research, Usage Control (UCON) [18], introduced a more dynamic model integrating authorizations with ongoing obligations and conditions. Foundational UCON models and their more recent context-aware extensions (CA-UCON) [22] enable policies that require decision continuity and can react to mutable attributes during a data usage session. This work is significant for introducing temporal and conditional logic that goes beyond a simple, one-time "permit" or "deny." However, the UCON framework was not designed to manage agents that can generate a large space of complex, unforeseen actions on demand, which is a core challenge with LLM agents.

More recent work pushes OBAC toward the fine-grained control necessary for modern data ecosystems [23, 24]. This includes applying ontologies to enforce provenance-aware policies for FAIR data [24] and using standardized vocabularies like ODRL [25] for interoperable usage control in decentralized systems. Other recent studies revisit OBAC from a data access perspective to integrate legacy sources [23]. These contemporary works highlight the field's trajectory towards data-centric, fine-grained control. However, our approach addresses a specific gap they do not: the nature of the LLM agent itself. Existing models, as surveyed in the literature, often treat an "action" as a static, atomic category. In contrast, an LLM agent generates operations that are not atomic but are structured, hierarchical, and dynamically composed. The ARGOS pattern contributes a specific mechanism that is lacking in prior work for decomposing these complex operations. It allows policies to bind to the sub-parts of a single query, providing a necessary layer of granular control for the unique challenge of managing a small number of powerful agents with a large, dynamic action space.

6. Conclusion

For future work, we plan to facilitate broader adoption and interoperability by mapping the ARGOS ontology design patterns to standard models in the Generative AI landscape, such as the GENAIO ontology [26]. Currently, ARGOS formalizes the semantic interpretation of data operations but does not model the semantics of the data itself. A significant avenue for extension is to combine our operational semantics with data semantics representation mechanisms like SHACL [27]. This integration would pave the way for fully automated access control systems, where complex policies could potentially be defined and enforced through simple natural language directives.

This paper addressed the critical security gap that emerges when LLM-based RAG agents are granted direct access to databases. We introduced a novel ontology design pattern, ARGOS, that unifies object-level and action-level access control, offering a coherent framework for managing the vast and unpredictable query variations generated by LLMs. Complemented by a formal set of reasoning rules, our framework dynamically evaluates and enforces defined access rights at runtime, ensuring policy compliance for any incoming query. This work thus provides a provable and interpretable foundation for securing the next generation of RDB-backed RAG systems against unauthorized or unintended data access. Looking forward, the proposed ontology has the potential to become a cornerstone for fully semantic access control systems, extending beyond RAG to a broad class of applications requiring deep, context-aware data governance and fine-grained policy enforcement over relational databases.

Declaration on Generative AI

During the preparation of this work, the author(s) used Gemini-2.5 Pro and GPT-4 in order to: Grammar and spelling check. After using these tool(s)/service(s), the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

References

[1] H. Lindqvist, Mandatory access control, Master's thesis in computing science, Umea University, Department of Computing Science, SE-901 87 (2006).

[2] D. D. Downs, J. R. Rub, K. C. Kung, C. S. Jordan, Issues in discretionary access control, in: 1985 IEEE symposium on security and privacy, IEEE, 1985, pp. 208–208.

[3] C. Ramaswamy, h. R. S, Role-based access control features in commercial database management systems (1998) 503–511.

[4] A. Padia, T. Finin, A. Joshi, Attribute-based fine grained access control for triple stores (2015).

[5] G. Xiao, D. Calvanese, R. Kontchakov, D. Lembo, A. Poggi, R. Rosati, M. Zakharyaschev, Ontology-based data access: A survey, in: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, International Joint Conferences on Artificial Intelligence Organization, Stockholm, Sweden, 2018, p. 5511–5519. URL: https://www.ijcai.org/proceedings/2018/777. doi:10.24963/ijcai.2018/777.

[6] R. Sandhu, Relational database access controls, Handbook of information security management 95 (1994) 145–160.

[7] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, Y. Katz, Pellet: A practical owl-dl reasoner, Journal of Web Semantics 5 (2007) 51–53. URL: https://www.sciencedirect.com/science/article/pii/S1570826807000169. doi:https://doi.org/10.1016/j.websem.2007.03.004, software Engineering and the Semantic Web.

[8] O. Standard, extensible access control markup language (xacml) version 3.0, A:(22 January 2013). URl: http://docs. oasis-open. org/xacml/3.0/xacml-3.0-core-spec-os-en. html (2013).

[9] J.-W. Byun, N. Li, Purpose based access control for privacy protection in relational database systems, The VLDB Journal 17 (2008) 603–619. doi:10.1007/s00778-006-0023-0.

[10] T. Lebo, S. Sahoo, D. McGuinness, PROV-O: The PROV Ontology, Recommendation, W3C, 2013. URL: https://www.w3.org/TR/prov-o/.

[11] A. Quiña Mera, P. Fernandez, J. M. García, A. Ruiz-Cortés, Graphql: A systematic mapping study, ACM Comput. Surv. 55 (2023). URL: https://doi.org/10.1145/3561818. doi:10.1145/3561818.

[12] S. Harris, A. Seaborne, SPARQL 1.1 Query Language, Recommendation, W3C, 2013. URL: https://www.w3.org/TR/sparql11-query/.

[13] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, M. Dean, SWRL: A Semantic Web Rule Language Combining OWL and RuleML, in: W3C Member Submission, 2004. URL: https://www.w3.org/submissions/SWRL/.

[14] C. Thörn, On the quality of feature models, Linkopings Universitet (Sweden), 2010.

[15] W. O. W. Group, OWL 2 Web Ontology Language Document Overview (Second Edition), Recommendation, W3C, 2012. URL: https://www.w3.org/TR/owl2-overview/.
[16] World Wide Web Consortium, World wide web consortium (w3c), https://www.w3.org/, 2025. Accessed: 2025-09-06.
[17] M. Poveda-Villalón, A. Gómez-Pérez, M. C. Suárez-Figueroa, OOPS! (OntOlogy Pitfall Scanner!): An On-line Tool for Ontology Evaluation, International Journal on Semantic Web and Information Systems (IJSWIS) 10 (2014) 7–34.
[18] J. Park, R. Sandhu, The uconabc usage control model, ACM transactions on information and system security (TISSEC) 7 (2004) 128–174.
[19] Ö. Can, O. Bursa, M. Ünalır, Personalizable ontology based access control, Gazi University Journal of Science 23 (2010) 465–474.
[20] S. Javanmardi, M. Amini, R. Jalili, An access control model for protecting semantic web resources, in: Web policy workshop, 2006.
[21] D. Wu, X. Chen, J. Lin, M. Zhu, Ontology-based rbac specification for interoperation in distributed environment, in: Asian Semantic Web Conference, Springer, 2006, pp. 179–190.
[22] A. Almutairi, F. Siewe, Ca-ucon: a context-aware usage control model, in: Proceedings of the 5th ACM International Workshop on Context-Awareness for Self-Managing Systems, 2011, pp. 38–43.
[23] J. Chen, L. Schinke, X. Gong, M. Hoppen, J. Roßmann, Interoperable access and usage control of self-sovereign digital twins using odrl and i4. 0 language., in: IoTBDS, 2024, pp. 75–85.
[24] C. Brewster, B. Nouwt, S. Raaijmakers, J. Verhoosel, Ontology-based access control for fair data, Data Intelligence 2 (2020) 66–77.
[25] R. García, R. Gil, I. Gallego, J. Delgado, Formalising odrl semantics using web ontologies, in: Proc. 2nd Intl. ODRL Workshop, 2005, pp. 1–10.
[26] R. M. Vsevolodovna, Generative ai ontology (genai), NCBO BioPortal, 2024. URL: https://bioportal.bioontology.org/ontologies/GENAI, version 1.0. Accessed: September 3, 2025.
[27] H. Knublauch, D. Kontokostas, Shapes Constraint Language (SHACL), Recommendation, W3C, 2017. URL: https://www.w3.org/TR/shacl/.

7. Online Resources

ARGOS Ontology Pattern Website: https://tetherless-world.github.io/argos