The present work was submitted to
the Research Group
Software Construction

of the Faculty of Mathematics,
Computer Science, and
Natural Sciences

Bachelor Thesis

Implementation of a Metamodel for Viewpoint-based Security Pattern Descriptions

presented by

Michael Zerbe

Aachen, September 22, 2025

Examiner

Prof. Dr. rer. nat. Horst Lichter

Prof. Dr. rer. nat. Bernhard Rumpe

Supervisor

Alex Mattukat, M.Sc.

Acknowledgment

I would like to begin by thanking Prof. Dr. rer. nat. Horst Lichter for the opportunity to pursue this bachelor's thesis at the Chair of Software Construction. I also appreciate that he and Prof. Dr. rer. nat. Bernhard Rumpe agreed to examine my work.

I am deeply grateful to my supervisor, Alex Mattukat, M.Sc., whose guidance combined expertise, lightness, and humor. Working with him and discussing ideas was a pleasure, and his constructive feedback strengthened this thesis.

I am profoundly thankful to my family for their unwavering support throughout my bachelor's studies and in bringing this thesis to completion.

My heartfelt thanks go to my beloved girlfriend, whose constant encouragement brightened my days.

A special thanks goes to Moritz Wehr, Diyar Tetik, and Lukas Wang, whose camaraderie and humor made the journey through my studies all the more enjoyable.

Finally, sincere thanks to everyone who supported me along the way.

Michael Zerbe

Abstract

The security of software systems is a critical requirement in today's digital environment. Security patterns are a well-established means of supporting architects and developers in designing and developing secure software systems. However, their application remains limited. The reasons for this include suboptimal abstractions in existing pattern catalogs and limited modeling approaches for modeling security-relevant information in its entirety in the pattern description. To close this gap, as part of the SCAM research project by the Research Group Software Construction at RWTH Aachen University, a security design pattern description metamodel (SDPDM) was developed. As part of the project's future work, this thesis provides a reference implementation of the SDPDM, which allows practitioners to collect, create, and review security patterns. We operationalize the SDPDM by implementing it in a graph database. Following an incremental process, we implement each viewpoint and iteratively refine the implementation until all elements and modeling rules are faithfully represented. We demonstrate correctness by fully instantiating the Single-Factor Password-based Authentication security pattern, yielding 286 nodes and 744 relationships that conform to the SDPDM and exercise the implementation. The instance serves as an example of how to use and work with the implementation. By implementing the SDPDM, this thesis provides practitioners in the secure design of software systems with an accessible means to use the SDPDM to collect, create, and review security patterns. Furthermore, this work establishes a structured foundation for future research on security patterns.

Contents

List of Figures

List of Source Codes

1. Introduction

Nowadays, the security landscape is vast and complex, a reality that is reflected in the staggering cost of security breaches, which is estimated to reach 10.5 trillion dollars worldwide by 2025 [Cyb22]. To overcome this problem, research presented the idea of "Security-by-Design", aiming that security is a fundamental requirement throughout the process of software development, instead of only considering security after finishing the software product development. This process is also known as "Shift-left". *Security Patterns (SPs)* are an established means for Security-by-Design, as they present reusable solutions for security requirements on a conceptual level without concrete implementation details. However, the application of *SPs* remains limited, because they are often too abstract and not easily applicable in real-world contexts [SLL25]. The Improvement of *SPs* and their usability is recognized as a state-of-the-art goal, as also emphasized by OWASP in their Top Ten category A04:2021 Insecure Design [OWA21]. The SCAM research project of the SWC group is trying to achieve Security-by-Design, contributing by providing secure modeling approaches and making security models more accessible to system architects [Sof23]. To address the problem that *SPs* are often too abstract and difficult to apply in real-world contexts, Lammers proposed in his Master's thesis "Conception of a Security Design Pattern Catalog for Constraint-based Recommender Systems" [Lam24], conducted within the SCAM research project, a division of *SPs* into usage and knowledge aspects. For the usage aspect, he introduced the *Security Design Pattern Description Metamodel (SDPDM)*, which aims to improve the structure, completeness, and usability of *SPs*. It also enables the possibility of providing a tool that can collect, create, and review *SPs* in a consistent model. Based on this foundation, the contributions of this thesis are presented in the following section.

1.1. Contribution

This thesis will be part of the SCAM research project and contributes by implementing the *Security Design Pattern Description Metamodel (SDPDM)* [Lam24], as presented by Lammers. The *SDPDM* uses *viewpoints* to describe the necessary aspects of a security feature to enable its correct implementation. *Viewpoints* define the elements and modeling rules of *views*, which describe the different aspects of the *SP*. To comprehensively cover all important elements of an *SP*, the *SDPDM* defines four *viewpoints*: *Conceptual, Data, Behavioral,* and *Structural Viewpoint*. Each of these captures different aspects of a security feature in a simplified manner, making it easier to select an appropriate technique for a given system, thus aligning with the goals of the SCAM research project. Furthermore, it establishes relations between the *viewpoints*, illustrating detailed usage scenarios, such as how a *Role*, which represents the responsibilities to implement the rules and required conditions

to resolve the problem of an *SP*, behaves and the data it uses during that behavior [Lam24]. We will implement these *viewpoints* and their relations according to the descriptions provided by Lammers. In addition, we will instantiate an example *SP* of Lammers' *SP* catalog from his MT report [Lam24], enabling the usage of our implementation and future scientific work. Due to time restrictions, we will not evaluate the *SDPDM* and do not provide a profound conformance check on the *SDPDM* implementation and data instance of the *SP*.

Furthermore, this thesis will also contribute to the SecuRe Approach, which aims to simplify decision making when choosing an *SP* by providing design recommendations based on security requirements [SLL25]. Our contribution lies in providing a database that contains an initial *SP* and that allows the collection, creation, and review of *SPs*. This supports the conclusion of SecuRe that a foundation for the reuse of security design knowledge is essential [SLL25], since our goal is to offer this database for scientific work, allowing further use and extension of the achievements of this thesis.

1.2. Research Question

Given the contributions we want to achieve, namely the implementation of the *SDPDM* to enable the collection, creation, and review of *SPs*, we want to examine in this thesis whether a graph database is able to host the implementation of the *SDPDM*. Accordingly, this thesis focuses on the following Research Question:

RQ: How can the *SDPDM* be implemented in a graph database while supporting its fundamental concepts and preserving its semantics?

With "supporting its fundamental concepts," we mean the *viewpoint* structure of the *SDPDM* that *SPs* are strictly divided into different aspects and relationships between the *views* enable capturing security-relevant information. By "preserving its semantics," we refer to the correct and, therefore, intended implementation of the elements, relationships, and constraints the *SDPDM* presents. The study aims to determine feasibility and to provide key limitations and workarounds.

1.3. Thesis structure

The structure of this thesis is as follows:

- In chapter 2, we present the relevant related work for this thesis and explain how this thesis extends or contributes to that work.
- In chapter 3, we introduce definitions and terminology that are important in the context of this thesis. We will first clarify the concept of *Security Pattern (SP)*, then explain what the *Security Design Pattern Description Metamodel (SDPDM)* from Lammers' MT report [Lam24] is, describe its structure and introduce the definitions relevant to this thesis.

- In chapter 4, we present the technical requirements for the graph database, which were extracted from Lammers' MT report [Lam24]. Then we explain the process of implementing the *SDPDM* and how to instantiate *SPs* in the database.
- In chapter 5, we explain why Neo4j was chosen as the graph database for implementing the *SDPDM*. We introduce Neo4j and show a feasibility analysis of Neo4j, with respect to the technical requirements we assessed in chapter 4.
- In chapter 6, we present the concrete implementation of the *SDPDM* in a Neo4j GraphQL schema. We first introduce the technologies used, introduce which *SP* was selected as an example *SP*, and then show the implementation of the *SDPDM* schema. We also present the implementation of a design choice we made and introduce the Query Builder, whose purpose is to improve the usability of the implementation.
- In chapter 7, we demonstrate the Neo4j GraphQL schema of the *SDPDM* from chapter 6, by showing certain parts of the instantiated *PBA SP* in the Neo4j property graph.
- In chapter 8, we validate the *SDPDM* schema by validating the instantiated *PBA SP*. We follow the validation approach introduced in chapter 4, by concretizing the idea of the approach and performing it.
- In chapter 9, we critically evaluate the implementation of the *SDPDM*. In addition, strengths, limitations, and potential improvements of the approach are discussed.
- In chapter 10, we summarize the methodology and results of this thesis and highlight the directions for future research and development.

2. Related Work

In this chapter, we present the relevant related work for this thesis and explain how this thesis extends or contributes to that work.

2.1. The Security Design Pattern Description Metamodel (SDPDM)

The foundation of this thesis is the *SDPDM* proposed by Lammers. In his MT report, the metamodel is formally described, and we will use this description to implement the metamodel in Neo4j, a graph database management system. The *SDPDM* is a viewpoint-based approach for *SP*, which seeks to describe *SP* in greater detail than previous approaches [Lam24]. *Viewpoints* are defining elements and modeling rules of *views*, which are representing specific aspects of an *architecture*, such as the static and dynamic structure [Lam24]. The viewpoint-based approach for an *SP* separates the *SDPDM* from previous approaches, as *views* are well known in software architecture and therefore easier to understand for architects and because the *views* separate the security-relevant information into smaller pieces which improves understandability and usability [Lam24]. Lammers defined four *viewpoints*, which describe an *SP* with its four corresponding *views*: *Conceptual*, *Data*, *Behavioral*, and *Structural Viewpoint*. The *SDPDM* and its *viewpoints* will be explained in more detail in chapter 3.

Our contribution to Lammers MT report [Lam24] is concerned with the implementation of all these *viewpoints* by a technical solution. We will extend the application of this metamodel by implementing it, enabling both the practical use of the metamodel and the ability to leverage its benefits. Furthermore, this implementation will facilitate the collection, creation, and review of *SPs*.

2.2. Sehbaoui's Bachelor Thesis

This thesis will collaborate with the thesis of Sehbaoui, who is developing a tool to graphically visualize the *viewpoints* of *SPs* based on the *SDPDM*. Its goal is to bridge the gap between the theoretical concept of the *SDPDM* and the practical application, providing a GUI that helps software architects without deep security expertise to understand and apply *SPs*. Further details can be found in his thesis [Seh25].

Sehbaoui's work is closely related to this thesis. While our contribution lies in implementing the *SDPDM* in a graph database and providing the corresponding implemented metamodel and instantiation of an example *SP*, Sehbaoui builds on this foundation by using

the metamodel implementation to define the necessary metadata for SVG files generated by his tool. He will also use the instantiated *SP* example to visualize it with his tool. Together, both theses contribute to the SCAM research project by operationalizing the *SDPDM* and making *SPs* more accessible to practitioners.

2.3. Ontologies and Conceptual Metamodels on Graph Databases

Our implementation lies in the field of using a graph database to implement ontologies. In this section, we review work that shares the same intersection of ontologies and graph databases. We cover the storing and querying of domain ontologies on Neo4j and translating conceptual models, such as UML/OCL, into property graph implementations.

Gong et al. efficiently stored and retrieved an oilfield domain ontology expressed in RDF/OWL (Resource Description Framework/Web Ontology Language) in Neo4j by defining explicit mapping rules from RDF/OWL ontology files to a Neo4j labeled property graph representation. They realized the conversion from RDF to Neo4j with the Java Jena API. To achieve a satisfactory result, Gong et al. proposed a two-tier index architecture, including object and triad indexing. They also proposed a retrieval method based on these indices to match different query patterns, including relational degree retrieval. In their evaluation, the approach reduces 13.04% storage compared to relational database methods and yields more than 30 times faster retrieval [Gon+18].

Spasov, Lazarova, and Petrova-Antonova state that the data for research on Alzheimer's disease (AD) are heterogeneous in naming, granularity, and format. Therefore, they motivate the construction of an Alzheimer's disease knowledge graph to aid in analysis. They built a Neo4j-based AD knowledge graph based on the AD-DPC domain ontology (Alzheimer's disease Ontology for Diagnosis and Preclinical Classification), which they proposed in earlier work. This medical domain ontology incorporates the knowledge of medical experts in a way that is understandable for non-experts [LPK23]. They populated the knowledge graph with ADNI (Alzheimer's Disease Neuroimaging Initiative) data. The resulting knowledge graph includes 2,996 diagnoses, 154,953 psychometric, 24,102 blood, 12,471 CSF (Cerebrospinal Fluid), and 14,703 brain imaging findings. The nodes were annotated with 259,260 labels and 673,325 relations based on the AD-DPC ontology. Spasov, Lazarova, and Petrova-Antonova conclude that ontologies provide an effective semantic modeling basis for graph databases with straightforward querying and visualization [SLP24].

Daniel, Sunyé, and Cabot note that, while mappings from conceptual schemas to relational databases are well-studied, only a few solutions target NoSQL databases and even fewer focus on graph databases. They further state that this holds particularly for mapping business rules and constraints. Therefore, they provide a systematic path from UML/OCL (Unified Modeling Language/Object Constraint Language) conceptual schemas to property graph implementations. They propose UMLtoGraphDB, a systematic pipeline from UML/OCL to property graph implementations. It consists of three main elements. Class2GraphDB which maps UML class diagrams to an abstract property graph metamodel, OCL2Gremlin, which translates OCL constraints and queries from the conceptual schema into Gremlin traversals,

and Graph2Code, which generates Java middleware using the Blueprints API to access and update the target graph database and enforce the modeled rules. Daniel, Sunyé, and Cabot provide tool support and report that the UML/OCL in code transformation runs in a few seconds on several examples. A detailed scalability study is stated as future work [DSC16].

In contrast to these works, which map and query domain ontologies in Neo4j and provide a general UML/OCL to property graph pipeline, this thesis implements a security-specific viewpoint-based metamodel (*SDPDM*) on a graph database and focuses on making *SPs* collectible, creatable, reviewable for practitioners in the secure design of software systems and consumable for visualization tools. In this thesis, we operationalize a security domain ontology rather than a medical domain ontology or UML mapping.

3. Theoretical Foundation

In this chapter, we introduce definitions and terminology that are important in the context of this thesis. We will first clarify the concept of *Security Pattern (SP)*, then explain what the *Security Design Pattern Description Metamodel (SDPDM)* from Lammers' MT report [Lam24] is, describe its structure and introduce the definitions relevant to this thesis.

3.1. Security Pattern

The *Security Pattern (SP)* was introduced by Yoder and Barcalow in 1998 [YB]. Since then many research efforts have been made in the field of *SPs*, such as non-exhaustive *SP* catalogs from Schumacher et al., Fernandez-Buglioni, or van den Berghe, Yskout, and Joosen. In its core, an *SP* is a description of a recurring security problem in a given context and provides a solution to this security problem, for example, in the form of UML diagrams [Sch+06] [Fer13]. However, Heyman et al. found that even 12 years after the first *SP* catalog was published by Schumacher et al. their adoption in practice still remains limited. The reasons found in the referenced paper are, for example, that some *SPs* are too abstract, give too low-level mechanism descriptions, or that *SP* solutions often describe what should be the outcome instead of presenting the strategy to achieve the outcome [Hey+07]. To address this problem and improve the adoption of *SPs* in practice, Sabau, Lammers, and Lichter introduced a novel concept-model for *SPs*. To define the concept-model for *SPs*, we first need to define what a *security control* is. A *security control* addresses *security requirements*, which specify a *security condition* that a system should meet, following the security-by-design paradigm. The *security control* reduces the vulnerability of a software system by being an action, procedure, technique, or other measure [NIS06]. An example of this is authentication. To group the various options that exist for each *security controls* realization the SecuRe approach used the concept of *SP*. *SP* is a reusable solution for a *security control* on the conceptual level without concrete implementation details. We only use the term *Security Pattern (SP)* in this thesis and not *Security Design Pattern (SDP)*, as we refer to the SecuRe approach, which says that SDPs are just special SPs. In the context of authentication (AuthN) an *SP* represents a conceptual solution of AuthN, for example, *password-based* or *passkey-based* AuthN [SLL25].

3.2. Security Design Pattern Description Metamodel (SDPDM)

Looking at the definition of *SP* we can see the need for a structured way to work with this concept, and that is where Lammers' MT report [Lam24] introduced the *SDPDM*, with the

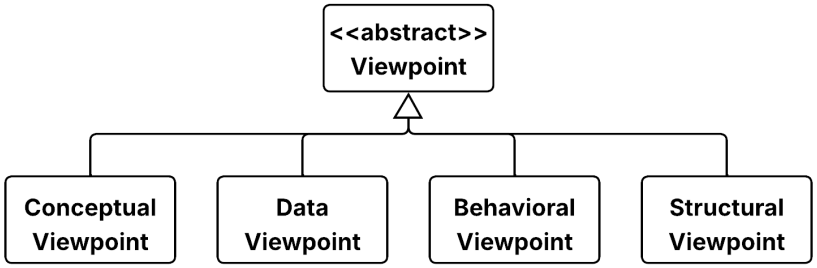

Figure 3.1.: The *SDPDM's viewpoints* that define how to represent the Solution and Example of an *SP* [Lam24].

goal of addressing the lack of a structured and accessible tool for reviewing security features [Lam24]. With the *SDPDM* we can work in a structured way with the concept of a *SP*. To begin with, Lammers defines *views* and *viewpoints* as follows.

Definition 3.1 (Architecture)**.** The *Architecture* of a system comprises its static and dynamic structures. The static structure includes structural elements and their organization, while the dynamic structure includes runtime elements and their interactions. An *Architecture* also has externally visible behavior that defines the system's provided functionality and its quality properties [Lam24] [RW11].

Definition 3.2 (View)**.** A *View* represents specific aspects of the static or dynamic structure of an *architecture* relevant to one or more stakeholders [Lam24] [RW11].

Relevant to note here is that a *view* only helps to understand an *architecture* by explaining a specific aspect and limiting the information to what is relevant to achieve that. Describing a whole *architecture* in a single model would make it unmanageable due to complexity [Lam24].

Definition 3.3 (Viewpoint)**.** A *Viewpoint* defines the elements and modeling rules for constructing a specific type of *view* [Lam24] [RW11].

This is very important in the context of the *SDPDM* as it defines *viewpoints* to have structured and consistent *views* which can easily be understood, if a stakeholder is familiar with the *viewpoint* [Lam24] [RW11].

3.2.1. SDPDM Viewpoint Structure

Lammers identified four viewpoints to achieve focusing on specific aspects at different levels of abstraction. Figure 3.1 taken from Lammers' MT report [Lam24] shows the identified *viewpoints*. Figure 3.2 taken from Lammers' MT report [Lam24], recreated as the *SP* version, shows the main elements and the inter-view relationships between the *viewpoints* [Lam24].

Before we describe each *viewpoint*, note that Lammers defined the *SDPDM* on *SDP*. As already mentioned, the SecuRe approach states that a *SDP* is only a special *SP* [SLL25]. In addition, there exists a not yet published version of the *SDPDM* that uses only *SP* to define the *SDPDM*. Therefore, we also define the metamodel for *SP*, not *SDP*. Due to defining it on *SP* we need to recreate the Figure 3.2, to stay consistent in this thesis with fonts and diagram styles, every figure taken from Lammers is recreated. We are using the *SDPDM* in this thesis for its implementation, therefore we need to introduce the *SDPDM* in this thesis, but we do not cover all the details. For details of the *SDPDM* please read Lammers' MT report [Lam24].

Conceptual Viewpoint

Definition 3.4 (Conceptual Viewpoint)**.** The *Conceptual Viewpoint* specifies how *Policies* and *Roles* are represented. *Policies* define the rules and conditions required to resolve the problems of an *SP*. *Roles* define the responsibilities required to implement the *Policies* [Lam24].

The *Conceptual View* serves as the foundation for other *views* as it provides a high-level understanding of the *SP* solution [Lam24]. Lammers' MT report [Lam24] does not cover an explanation which *Unified Modeling Language (UML)* diagram the *Conceptual View* follows. But in regard to the *view*, we have *Roles* that maintain relationships to each other as well as to *Policies*. The *Conceptual View* can be associated with a modified version of the UML class diagram, in which *Roles* are modeled as Classes without attributes or methods. The relationships between the *Roles* are represented as associations between the corresponding *Role* Classes. *Policies* and the relationship between *Roles* and *Policies* can be represented as further information about the UML class diagram, for example, notes.

Data Viewpoint

Definition 3.5 (Data Viewpoint)**.** The *Data Viewpoint* defines how to model the *Data Elements* involved in an *SP*, including their structure, properties, and security considerations [Lam24].

The *Data View* of an *SP* specifies which *Data Elements* are involved in the *SP* Solution, how these *Data Elements* relate to each other, and assigns security-relevant information to the *Data Elements*. The impact that disclosure of a *Data Element* would have on the security of a user or system is part of that information. Improper data design can result in an insecure system, because of that the *Data View* is critical for the secure implementation of an SP [Lam24] [DJS19].

Data Group serves as a container for similar data. *Data Fields* are a specific piece of data with all their defined properties. *Data Groups* and *Data Fields* follow the *Unified Modeling Language (UML)* class diagrams, where *Data Groups* correspond to classes and *Data Fields* correspond to attributes [Lam24].

Behavioral Viewpoint

Definition 3.6 (Behavioral Viewpoint)**.** The *Behavioral Viewpoint* specifies how to model the detailed interactions between *Roles* and how to represent *Events*. An *Event* represents a specific situation that occurs during these interactions and allows important security considerations to be defined [Lam24].

The *Behavioral View* of an *SP* models the behaviors required to securely implement the *Policies*. The modeled behavior typically involves sending and receiving data using the *Data Elements* modeled in the *Data View*. Events can highlight and describe important situations within the modeled behavior, representing both exceptional (e.g., "Password Incorrect") and informational (e.g., "Successful Authentication") situations. Behavior can be modeled by using *Data Elements* and *Events* in an understandable way, while still containing the security-relevant information defined within these elements [Lam24].

A *Behavioral Model (BM)* represents a specific aspect of an *SP*, detailing how *Roles* interact to implement *Policies*. The *Behavioral Modeling Element (BME)* represents interactions, conditions, or flows that illustrate the dynamic behavior of *Roles* and are the fundamental component of the *BM*. We can model *BM* in any type of behavioral diagram, such as UML sequence and activity diagrams. The *BMEs* include all possible model elements that represent interactions, conditions, or flows, such as messages in sequence diagrams or activity and decision nodes in activity diagrams [Lam24].

Structural Viewpoint

Definition 3.7 (Structural Viewpoint)**.** The *Structural Viewpoint* specifies how to represent the *Roles* within an architecture. It takes into account both the static structure, i.e., the specific elements and their arrangement, and the dynamic structure, i.e., the runtime elements and their interactions [Lam24] [RW11].

The *Structural View* of an *SP* provides practical *Examples* that facilitate the implementation of the *SP* by giving architects a well-documented starting point. The *Structural Viewpoint* defines how to represent *Examples* that show the implementation of the *SP* Solution within an architecture. These *Examples* serve as starting points, allowing architects to adopt specific components and discuss the underlying design choices [Lam24].

The *Architectural Model (AM)* represents the static and dynamic structure of specific aspects of the *SP* solution. The *Architectural Modeling Element (AME)* is the fundamental component of the *AM*. There is no restriction from the *Structural Viewpoint* how *Examples* are represented in terms of *AMs* and *AMEs*. An *AM* can be modeled using UML diagrams, such as component diagrams to represent the static structure and sequence or activity diagrams to represent the dynamic structure [Lam24].

3.3. Inter-view Relationships

Inter-view relationships are a fundamental improvement for the structure of *SPs*. It provides security-relevant information between each *view* of an *SP*, such as how a *Role* behaves and

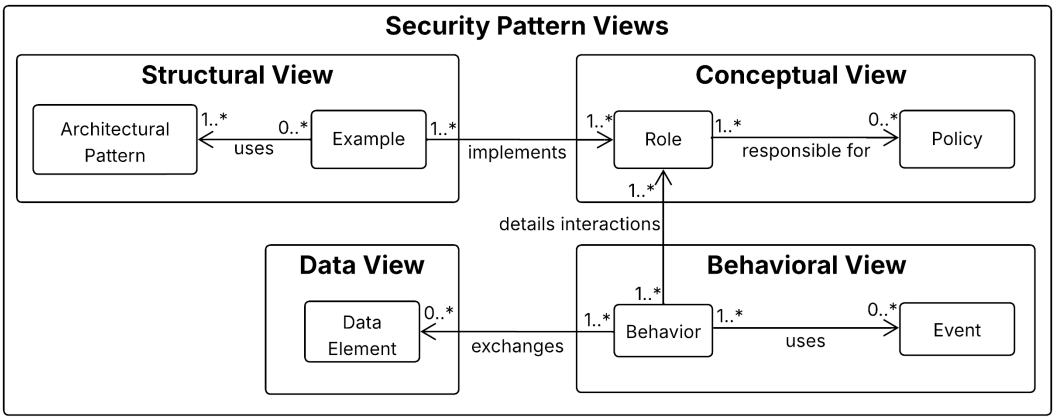

Figure 3.2.: Overview of an *SP* that conforms to the *SDPDM Viewpoints*. Each view highlights its main elements and their relationships [Lam24].

which data elements it uses during that behavior. This allows the *SDPDM* to separate aspects of an *SP* and keep boundaries between the *views* to reduce complexity. We now explain the inter-view relationships that can be found in Figure 3.2. The figure shows the main elements of each *view* and overviews an *SP* that conforms to the *SDPDM Viewpoints*. The *Conceptual View* has *Roles* and *Policies*. The *Roles* are being implemented by the *Examples*, which are in the *Structural View*. The *Examples* provide architects with an overview on how to implement the *SP* solution, which conforms to the *Conceptual*, *Data* and *Behavioral View* and uses a specific *Architectural Pattern*, such as the microservice architecture. The detailed interactions of *Roles* are depicted by the *Behavior*, which can cause *Events*, of the *Behavioral View*. The *Behavioral View* also exchanges *Data Elements* from the *Data View* [Lam24].

3.4. Password-based Authentication Security Pattern (PBA SP)

To ground this chapter, we briefly present one *view* of the *PBA SP* from Lammers' MT report. We chose the *Conceptual View* as our example because it provides a single UML class diagram that covers all aspects of the *PBA SP*, including its four *Policies*. This example shows what a concrete *view* of a concrete *SP* looks like in the *SDPDM* and foreshadows our methodology, in which the *PBA SP* serves as the reference SP for the initial implementation of the *SDPDM*. For further details on *PBA SP*, we refer to Lammers' MT report [Lam24].

In *PBA* the authentication of a user is done using a secret password, which is associated in our context with the user's email address. To gain access to the system, the user needs to register by providing an email address and password. The *Registration* is an *Policy* in the case of *PBA SP* in the *SDPDM*. *Login* is a further *Policy*, in which the user can log in to the system, providing the email address and password of the user. The password

can also be reset, when the user forgets it, this is the third *Policy Password Reset*. This procedure involves further implementation, such as a *password reset link* to enable the user to change their password. The fourth *Policy* is the *Email Change*, which allows the user to change the registered email address by providing their password and confirming the access of the new and old email addresses. To visualize *PBA SP* we can find in Figure 3.3 the *Conceptual View* of the *PBA SP*, illustrated as a UML class diagram, taken from Lammers' MT report. As mentioned earlier, the UML classes without attributes and methods are the *Roles*, which define the required responsibilities to implement the four mentioned *Policies*. The associations between the *Roles* represent the *requires* relation, which states that a *Role* depends on another *Role* to fulfill its responsibilities. The relations between *Policy* and *Roles* are depicted by notes, as well as the *Role's* properties. For further detail on the *Conceptual View* and *PBA SP* we refer to Lammers' MT report [Lam24].

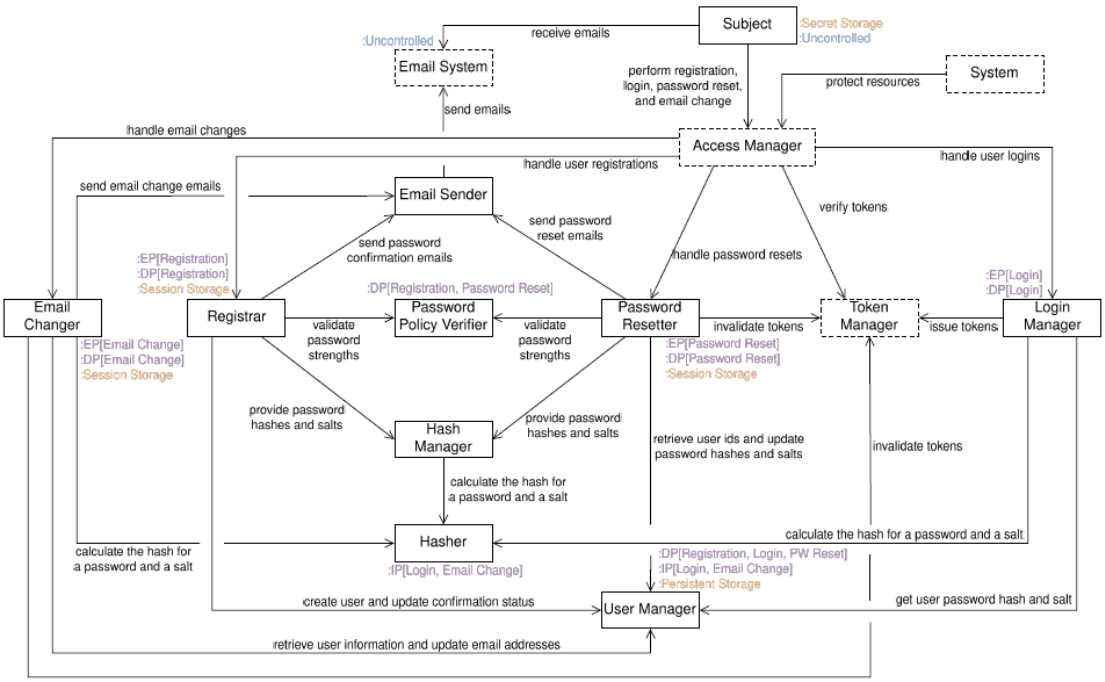

Figure 3.3.: PBA SP Conceptual View Diagram taken from Lammers' MT report [Lam24].

4. Methodology

Our goal in this thesis is to implement the *SDPDM* and by that being able to instantiate example *SPs* in a graph database. To achieve this goal, we present in this chapter the methodology we used to implement the *SDPDM* and instantiate example *SPs* in a database. Important for the correct implementation of the *SDPDM* was to analyze Lammers proposed *SDPDM* to extract technical requirements, which our chosen graph database needed to satisfy. Therefore, the implementation of the *SDPDM* and the instantiation of example *SPs* can be separated into two phases. The first phase was the extraction of technical requirements, where we extracted the technical requirements our graph database needed to meet. The second phase was the actual development, where we proposed our method to implement the *SDPDM* and instantiate *SPs* in a graph database.

4.1. Technical Requirements

To achieve a satisfactory technical solution for the implementation of the *SDPDM* in a graph database, we analyzed the *SDPDM* in regard to requirements, which our graph database should be able to satisfy.

In the following, the extracted requirements are presented along with their rationale:

TR1: The *viewpoint hierarchy* of an *SP* must be realizable in the graph database.

Explanation: As the *viewpoint* and *view* concept is fundamental in the *SDPDM*, we need to be able to realize the *viewpoint hierarchy* in our graph database to ensure that we explain specific aspects of an *SP* at different levels of abstraction and to have clear boundaries between different aspects of an *SP* [Lam24].

TR2: The *Inter-view relationships* of an *SP* must be realizable in the graph database.

Explanation: *Inter-view relationships* are crucial as they connect the different aspects of an *SP*. Lammers stated in his MT report that existing patterns lack this, because the *inter-view relationships* are often given informally [Lam24]. Therefore, enabling this in our graph database is critical to achieve an improvement on how descriptive *SPs* are and to correctly implement the *SDPDM*.

TR3: All elements and modeling rules defined for all different *viewpoints* must be realizable in the graph database.

Explanation: A *viewpoint* defines the elements and modeling rules to construct a specific type of *view*, as we can read in the definition in chapter 3 [Lam24]. Therefore,

we need to be able to realize this in our graph database to have the different types of *views* for an *SP* from the *SDPDM* to represent the different aspects of an *SP* and to connect them by their *inter-view relationships*.

TR4: The graph database must be able to realize *SDPDM* conform *SP* instances, which are consumable by visualization tools to operationalize them.

Explanation: Lammers proposed for every *viewpoint* type except the *Conceptual Viewpoint* UML diagram types, which the *views* follow. The specific types can be seen in chapter 3. The graph database must be able to provide *SP* instances that conform to the *SDPDM* and whose information is complete such that visualization tools are able to operationalize the instantiated *SPs*. In the context of the proposed UML diagram types from Lammers it means that the visualization tool is able to generate UML diagrams out of the *SP* instances.

4.2. SDPDM Implementation

After finishing the extraction of technical requirements phase, we continue now with the development of the *SDPDM* implementation and the instantiation of *SPs* in a graph database. In order to break down the complex *SDPDM* and its example into simpler chunks, we first decided to select a *SP* as an example. This *SP* served as a template for implementing the *SDPDM*. For this *SP*, we designed an incremental, iterative approach to develop all *viewpoints* of the *SDPDM* and example *views* of the selected *SP*. The overall process is depicted in Figure 4.1. We denote the "graph dataset" as the persisted as graph stored data, for example, a set of vertices and edges, optionally with associated attributes, which are independent from any specific graph data model.

4.2.1. Security Pattern Selection

To meet the requirements of our technical solution and to persist the *SDPDM* in a graph database, it was necessary to select a *SP* from the Security Pattern catalog provided by Lammers [Lam24]. The selected *SP* needed to satisfy three criteria: It should cover all four *views* of the *SDPDM* to fully exercise the implementation, be simple enough to be instantiated in a reasonable amount of time, and be sufficiently relevant to justify its instantiation.

4.2.2. Incremental Approach

After selecting a suitable *SP*, we proceed with an incremental approach. For this purpose, we define an increment as a selected part or functionality of the *SP*, described through *views*, which can then be instantiated. To begin with a suitable choice, we decided on a simple but representative increment. The first increment has the purpose of initially creating an initial implementation for each *viewpoint* and to initially create the graph dataset for the *views* of the *viewpoints* that should be instantiated there. All subsequent increments use the already existing metamodel implementation and graph dataset. All increments go through the *Development Phase* and *Refinement Phase* for each *viewpoint*.

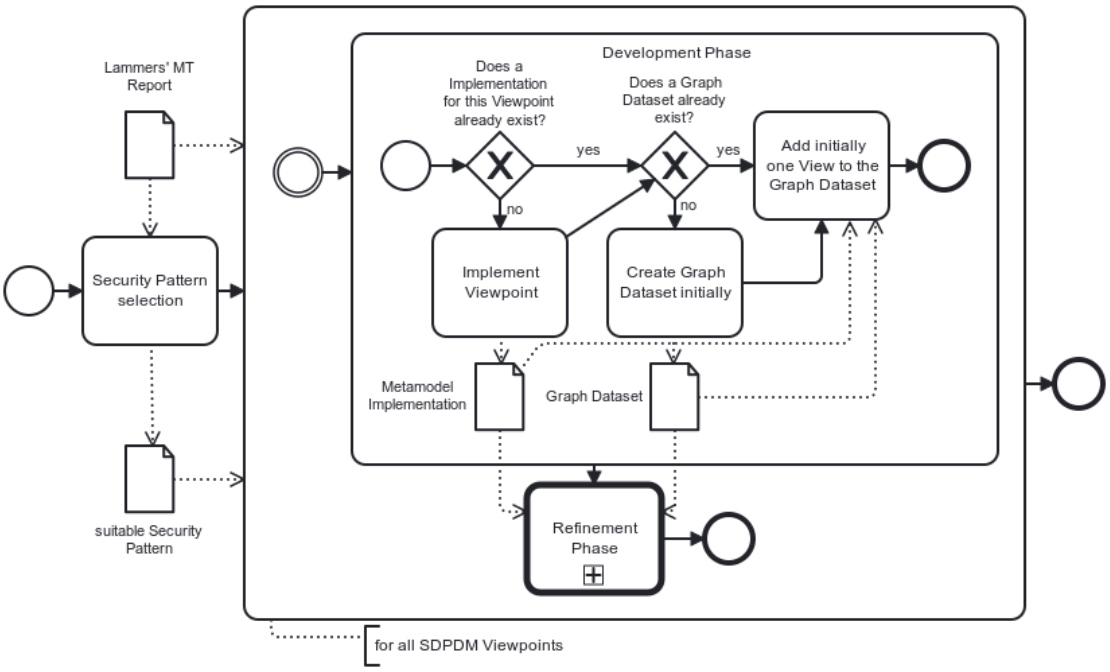

Figure 4.1.: Structure of the Overall Process in a BPMN diagram.

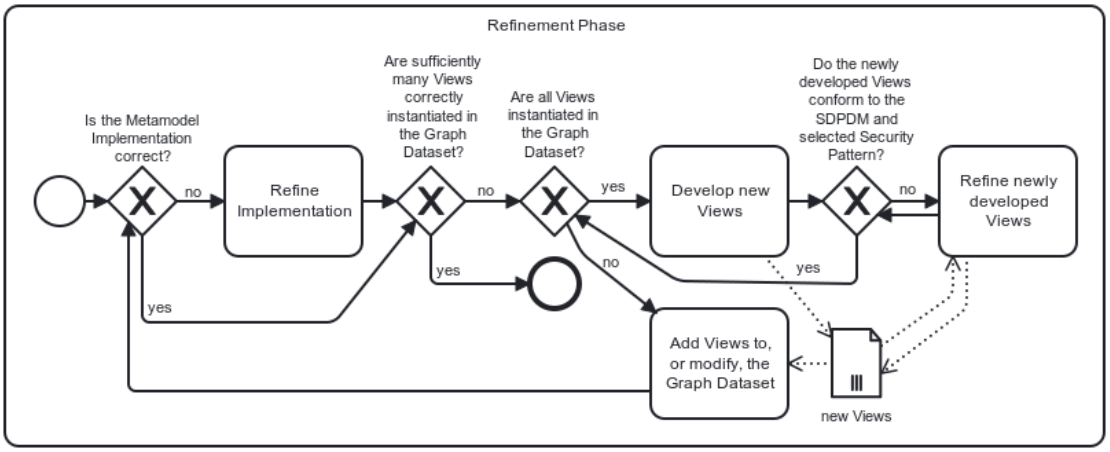

Figure 4.2.: Refinement Phase of the Overall Process in a BPMN diagram.

Development Phase

The goal of this phase is to create an initial metamodel implementation and graph dataset and, if done already, choosing an initial *view* of the current *viewpoint* to prepare the first iteration of the *Refinement Phase*. We first check whether an implementation of the metamodel already exists for this *viewpoint* or not. If it exists then we need to check whether a graph dataset already exists. If so, we can initially add one *view* of the *viewpoint* of the selected *SP increment* to the graph database and continue with the *Refinement Phase*, as described in Figure 4.2. If an implementation for the *viewpoint* does not exist, we need to implement the *viewpoint*. After that we check whether there exists a graph dataset already, if not, we need to create one initially, which can then be used by every *view* of every *viewpoint* and after creating the graph dataset we initially add one *view* of the *viewpoint* from the selected *SP increment*. We continue with the *Refinement Phase*, as described in Figure 4.2.

Refinement Phase

The goal of the *Refinement Phase* is to add all remaining *views* of the *viewpoints* to the graph dataset and to test the implementation of the metamodel and, if necessary, to refine it. With the initial metamodel implementation and the graph dataset, we continue with the *Refinement Phase*, which follows a decision-oriented process, which can be summarized in following pseudocode:

```
1  WHILE true DO
2
3    ASK: "Does the current metamodel implementation allow
         correct implementation of the current viewpoint and its
         views?"  YES/NO
4    IF answer = NO THEN
5      ACTION: Refine the implementation.
6    ELSE
7      ACTION: Skip the implementation refinement.
8    ENDIF
9
10   // "Sufficient views" means if every variant of the
         viewpoint is represented
11   // (e.g., Behavioral Viewpoint represented as modified UML
         Activity Diagram OR modified UML Sequence Diagram)
12   ASK: "Are sufficiently many views correctly instantiated in
         the graph dataset?"  YES/NO
13   IF answer = YES THEN
14     RETURN: Exit the Refinement Phase.
15   ENDIF
16
17
18
19
20
21
```

```
22    nds = newly developed views
23    REPEAT
24      ASK: "Are all views of the current viewpoint (from Lammers
           MT report or nds) instantiated in the graph dataset?"
           YES/NO
25      IF answer = YES THEN
26        REPEAT
27          ACTION: Develop nds (that conform to the SDPDM and the
              selected SP).
28          ASK: "Do the nds conform to the SDPDM and the selected
              SP?"  YES/NO
29          IF answer = NO THEN
30            ACTION: Refine nds.
31          ENDIF
32        UNTIL answer = YES
33      ENDIF
34    UNTIL answer = NO
35
36    // Not all views are instantiated in the graph dataset
37    // Modifying the graph dataset means here that we might be
         able to correct a partly instantiated view because the
         implementation of the metamodel got refined
38    ACTION: Add missing views to the graph dataset or modify the
         graph dataset
39
40  ENDWHILE
```

Source Code 4.1: Refinement Phase (textual pseudocode)

Note in Source Code 4.1 that clearly when new *views* have just been developed, they are not yet instantiated, and Lammers' MT report [Lam24] does not provide infinite *views*. Because of that, there cannot occur an infinite loop. In the beginning of Source Code 4.1 we also see why it was necessary in the *Development Phase* to initially add one *view* of the *viewpoint* from the selected *SP increment*, because it is the base for the first *Refinement Phase* iteration, after which we add all remaining *views* in the *Refinement Phase*.

4.3. Case Study Process

After finishing the development of the *SDPDM* implementation and the instantiation of the example *SP* we conducted a small case study, whose goal is to manually validate the instantiation of the example *SP* and by that also validate the correctness of the implemented *SDPDM*. Due to time restrictions, we were unable to provide a large scale validation, such as validating the inter-view relationships as described in chapter 3.

We chose to manually validate each *view* of the example *SP*. That means that we validate the *views* separately. For the *Conceptual View* we decided to count the elements contained and compare the counted number against the expected number of elements, and we also

validate the connection of the elements. We also depict certain elements and validate them further. For the validation of the *Data View* we chose to select certain elements and inspect them in further detail. As explained in chapter 3 the *Behavioral View* is separated in different *Behavioral Model (BM)* which contain a number of *Behavioral Modeling Elements (BME)*. We decided to count each number of elements for each *BM* and add them up for the complete *Behavioral View* and compare the number with the expected number of elements for the *Behavioral View*. We also decided to select certain elements and validate them in further detail. A similar approach is taken for the *Structural View*, here we also count the elements for certain *Architectural Model (AM)* and compare the number to the expected number of elements. We also select certain elements for further inspection.

5. Neo4j Graph Database

In this chapter we explain why Neo4j was chosen as the graph database for implementing the *SDPDM*. We introduce Neo4j and show a feasibility analysis of Neo4j, with respect to the technical requirements we assessed in chapter 4.

5.1. Why Neo4j?

For the selection of Neo4j, we relied on the popularity ranking of graph databases. This ranking clearly shows that Neo4j is the most widely used graph database engine, with a score of 54.47 [DB-25a]. Neo4j is 31.63 points higher than the second most popular option, Microsoft Azure Cosmos DB [DB-25a]. The popularity score is derived from several factors: the number of mentions of the system on websites, the general level of interest in the system, the frequency of technical discussions, the number of job postings referencing the system, the number of professional profiles that mention it and its relevance on social networks [DB-25b]. After choosing Neo4j, we now introduce Neo4j and then show the feasibility of Neo4j as a suitable graph database for implementing the *SDPDM*.

5.1.1. Neo4j

To be able to conduct the feasibility analysis, we need to explain Neo4j, to understand and use its concepts and methods. Neo4j is a native graph database, which means that they use a graph model all the way down to the storage level [Neo25h]. In the Neo4j graph database data gets stored as property graphs, these have following components:

- **Nodes** which represent entities
- **Relationships** that connect two nodes
- **Properties** which are key-value pairs for nodes and relationships

Nodes can have labels, which represent a group of nodes. Relationships have types, are always directed in Neo4j, and need a start and end node [Neo25f] [Neo25a]. Neo4j has a query language with the name *Cypher*. As the data is represented as a property graph, which means that it contains nodes, relationships, and properties, the core syntax structure is `(:nodes)-[:relationships]->(:other_nodes)`. Nodes are represented in round brackets, and behind the `:` we write the already introduced label of the node. As we can see, the relationships are represented as square brackets, with their type after the `:`. Important to note here is that we can also see the direction of the relationships, which is indicated with the `->` after the square brackets, of course we could also set `<-` on the

other side of the square brackets to show the other direction of the relationship. With this syntax structure and keywords, such as `MATCH`, which tries to find the presented structure, we can perform CRUD (create, read, update, and delete) operations on our database. We can also only query nodes by removing the syntax structure behind the first closing round bracket, the same for relationships, where we have to remove the content of the round brackets. Nodes and relationships can also have assigned variables, by writing a string before the `:`, this way we can, for example, only return the node with a certain relationship and not both nodes with the relationship which connects both nodes [Neo25g]. To illustrate *Cypher* queries we want to make a small example, the *Cypher* query `MATCH (n:Student)-[r:writes_Bachelor_Thesis_at]->(n2:Chair) RETURN n, r, n2` returns all `Student` and `Chair` pairs, which are connected through the relationship of writing a bachelor thesis at the `Chair`. As we can see in the example, we used the variables before the `:` to return the nodes and relationship at the end. We can also see the direction of the relationship.

5.1.2. Neo4j GraphQL Toolbox

In the following feasibility analysis, we will see that plain Neo4j is not capable of fulfilling all technical requirements. For that reason, we will need to use the Neo4j GraphQL Toolbox. We will first introduce GraphQL and then continue with the Neo4j GraphQL library which is used by the Neo4j GraphQL Toolbox. GraphQL is a query language for APIs where we can define a type system for our data. The GraphQL specifications are open source, and many companies have created their own GraphQL APIs with tools that use them, such as Neo4j [Gra25]. This is also the reason why we do not build our own GraphQL API and connect it to the Neo4j graph database, as Neo4j built their own one which suits our purpose perfectly. The Neo4j GraphQL Library is the official toolkit for building a GraphQL API over Neo4j databases. It allows one to create GraphQL APIs with a type system suitable for Neo4j graph databases [Neo25c]. Instead of building our own API with the Neo4j GraphQL Library, we use the Neo4j GraphQL Toolbox because it provides a user interface (UI) for schema design, validation, and direct data manipulation in the Neo4j graph database without additional server setup. The Neo4j GraphQL Toolbox uses the Neo4j GraphQL Library and allows us to connect to our Neo4j graph database, define a Neo4j GraphQL schema, where all type definitions for our data are contained, and perform data manipulation on our Neo4j graph database, which conform to the defined schema [Neo25e].

In Figure 5.1 we can see the earlier mentioned `Student` writes a bachelor thesis at `Chair` example as a Neo4j GraphQL schema. The Neo4j GraphQL schema contains the type definitions of `Student` and `Chair`. We can see in line `1` how a type with a name is defined and the `@` declares this type definition as a node for the Neo4j graph database. The attributes of the type `Student` and `Chair` are the defined properties that the node `Student` can have in the Neo4j graph database. The relationship is defined in the type `Student`, and we can see that this is a relationship because of the `@` declaring it as a relationship. We can also see then the already mentioned type of the relationship in Neo4j and the direction of the relationship. The direction states in this example `OUT`, which means that the relationship is outgoing for the node `Student`. In the square brackets, we can see the type that is allowed

to be connected with this relationship, in our case it is the node `Chair`. The `!` declares that the property cannot be null. When we build this schema, we can use mutations, which is the operation to create, update, and delete data that conform to the schema. In Neo4j GraphQL Toolbox we get auto-generated mutations based on our built schema [Neo25d]. In the context of our previous example, we would have create, update, and delete mutations for the nodes `Student` and `Chair`.

```
type Student @node {
  matricalNumber: Int!
  name: String!
  bachelorThesisAt: [Chair!]! @relationship(type: "writes_Bachelor_Thesis_at", direction: OUT)
}
type Chair @node {
  name: String!
}
```

Figure 5.1.: An example Neo4j GraphQL schema containing the type definition of the nodes `Student` and `Chair` and `writes_Bachelor_Thesis_at` relationship.

5.1.3. Feasibility Analysis

For the feasibility analysis, we evaluated Neo4j against each requirement defined in section 4.1, which will be presented in the following:

TR1: Neo4j meets this requirement, as a graph database, it is able to take advantage of the graph database concept, which means it can connect nodes with relationships and define properties, which can be different for every node. In this way, we can express different aspects of an SP. Furthermore, we are able to create disconnected graph components, to ensure clear boundaries between different aspects of an *SP* [Neo25a].

TR2: In the context of graphs, the inter-view relationships, as explained in chapter 3, connect the disconnected graph components. They ensure that all security-relevant information remains available and we still have clear boundaries for each *view* so that the complexity is reduced [Lam24]. Neo4j is able to meet this requirement as it is capable of connecting disconnected graph components by relationships [Neo25a].

TR3: To meet this requirement, Neo4j must allow the definition of constraints on the node attributes, the permissible attribute values, and the types of edges to which nodes may be connected. A major issue is that plain Neo4j and its Cypher query language are not capable of enforcing a schema with strict constraints, which is necessary to implement the *viewpoints* of the *SDPDM*, representing all elements and modeling rules. However, as explained above we are able to overcome this limitation using the Neo4j GraphQL Toolbox. This Neo4j library allows us to define types that correspond to nodes in Neo4j and to restrict them to have specific relationships and properties [Neo25e]. Therefore, we are able to create a schema for Neo4j that restricts the nodes' attributes, their permissible attribute values, and the types of relationships to which

it may be connected. The Neo4j GraphQL Toolbox extends Neo4j here to meet the technical requirement.

TR4: This technical requirement is not obviously met for Neo4j, as we need to ensure that Neo4j is able to represent the *views* of an *SP* in a consumable way for visualization tools. Therefore, we decided to adopt the proposed UML diagrams in chapter 3 and chose two diagrams to test whether Neo4j is capable to represent the structure of the UML diagrams. We are using the introduced concepts of Neo4j and the Neo4j GraphQL Toolbox to model both UML diagram concepts, with respect to the *SDPDM*, as we need to make *SDPDM* conform data consumable for visualization tools.

To show this, we depict an easy UML diagram and explain how we tested this and then a UML diagram with workarounds needed. The UML Class Diagram is used in the *Conceptual Viewpoint* [Lam24]. It includes *Roles*, which are represented as classes without attributes or methods, and the *requires* relation, which is depicted as association between classes. Typically, the associations of a class diagram represent semantic relationships, such as inheritance or aggregation. However, in our case, they represent messages within the *requires* relation between *Roles*.

In the Neo4j GraphQL Toolbox, we can easily construct the type *Role* with all its properties. For example, the storageTag property, which states if a *Role* needs some kind of storage to fulfill its responsibilities [Lam24], requires a selection from predefined types such as session or cache, and must be restricted to exactly one of these types. All of this is representable using constraints in the Neo4j GraphQL Toolbox starting with the type and its name, followed by its properties, an enum to define the possible values for storageTag, and a restriction to ensure that the value is not a list. The *requires* relation between *Roles* is modeled as a relationship between *Role* nodes. This relationship can also have its own properties, which is especially useful in this case, as it includes a message. When examining the UML Class Diagram within the *Conceptual Viewpoint*, we observe that Neo4j and Neo4j GraphQL Toolbox are fully capable of creating a schema that defines all the required types, properties, and relationships.

On the other hand, when considering the UML Sequence Diagram used in the *Behavioral Model* and the dynamic part of the *Architectural Model* workarounds are necessary to represent it within the schema. The basic structure of a Sequence Diagram is straightforward. We can define a message type containing the message text and its type (e.g., Response or Asynchronous). The participants or actors in the *Behavioral Model* correspond to the previously defined *Roles*. To illustrate that we want to provide a small example, which illustrates the mapping of an UML sequence diagram in Neo4j. In Figure 5.2 we can see an example UML sequence diagram. `System` and `User` are in our example *Roles*, which are part of the *Conceptual View* of an arbitrary *SP*. These are nodes in Neo4j, which are used in the UML sequence diagram representation of the *Behavioral View* as participants in this case. The messages in the UML sequence diagrams, which are the arrows are represented as nodes in Neo4j, they contain the message text and their type, such as in our example for the first two messages *Asynchronous* and for the third message *Response*. Given all these nodes we

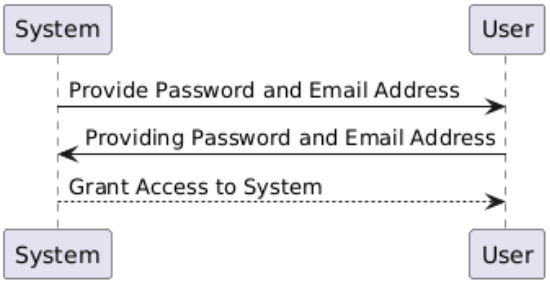

Figure 5.2.: An example UML sequence diagram to illustrate the mapping in Neo4j.

need to connect them using relationships. In this example the `System` sends the message `Provide Password and Email Address` to the `User`. To map this in Neo4j we would have a relationship of the type `sent` from `System` to the message node, which contains the mentioned message. From this message node we have a relationships of type `received` to `User`. This is how we map the basic structure of the UML sequence diagram in Neo4j. The other two messages follow the same structure. After creating all relationships, we would have correctly modeled the example UML sequence diagram in Neo4j.

However, representing Interaction Blocks, such as alternative blocks and their nesting capability, requires additional modeling. To address this, we defined two additional types: one for the Interaction Block itself and one for the Parts within an Interaction Block. An Interaction Block begins with a message, that is, the last message not inside the block. It also has an operator (such as alternative) and references its parts through a relationship. Each part contains its own messages and may include a guard condition. Both types also have parent and child block relationships. These relationships allow us to handle nested Interaction Blocks. Although this constitutes a workaround, it demonstrates that the Neo4j GraphQL Toolbox is capable of representing the concept of the UML Sequence Diagram. With this test we can conclude that Neo4j is capable of representing the *views* of an *SP* in a for a visualization tool consumable way and therefore meets this technical requirement.

Based on that evaluation, we concluded that Neo4j is feasible for the purpose of implementing the *SDPDM*. Given its feasibility and its popularity lead of 31.63 score points, we decided to adopt Neo4j as our database engine rather than extending the feasibility analysis to alternative graph database systems.

5.2. Query Builder

During the process of implementing the *SDPDM* and instantiating an example *SP*, we captured that there is a need for a tool to allow non-experts to view the data of our implementation. The main reason behind that is the usability of the *SDPDM* implementation, we

noticed that *SPs* have a lot of data to store, and queries to read certain data became quite complex. Therefore, we decided to provide a Query Builder that can build queries to read nodes, relationships, and properties in our database, as it builds Cypher queries. We go into more detail on the Query Builder in chapter 6.

5.3. Technology Landscape

The technology landscape resulting from our feasibility analysis is shown in Figure 5.3. It groups the stack into languages, tools, and database. As explained before, there are two main languages that were mainly used in the implementation, namely GraphQL and Cypher. We use Cypher to read data in Neo4j. Because queries become complex, the Query Builder helps users to build Cypher queries. The Neo4j graph database provides a console for direct execution of Cypher queries.

Neo4j does not natively support the GraphQL language. Therefore, we use the Neo4j GraphQL Toolbox, which builds on the Neo4j GraphQL Library, to connect to the Neo4j graph database, host the GraphQL API, and execute mutations for creating, updating, and deleting data. We also define a Neo4j GraphQL schema in the Toolbox, which implements the concrete elements and modeling rules of the *SDPDM*.

In short, we read data in the Neo4j graph database using Cypher, while we write on the graph database using GraphQL. This separation keeps the graph database conform to the defined *SDPDM* schema.

5.4. Change of the SDPDM

As explained in subsection 3.2.1, an *Example* facilitates an example implementation of an *SP* solution within an architecture. The *Architectural Model (AM)* represents the static and dynamic structure of specific aspects of the *SP* solution. The *Architectural Modeling Element (AME)* is the fundamental component of the AM [Lam24]. However, in the implementation of the *SDPDM* in Neo4j we identified a modeling error in the *SDPDM*. We had to adjust the idea behind an *Example* for the *Structural View* of an *SP* from Lammers' MT report [Lam24]. In Lammers' MT report [Lam24] the aggregation between *Example* and *AM*, was also present between *Example* and *AME*. However, this is incorrect and should not be the case. As the definition of an *AME* says that it is a fundamental component of the *AM* chapter 3. The *AME* should not be able to refer to an *Example* without being part of an *AM*. This would also not work for the structure of instantiating the *AMs* of *SPs* as UML Sequence, Activity, or Component diagrams. The *AME* would be able to exist without being part of an *AM*, which causes that we have no correct possibility to represent this *AME*, as it is not part of any UML diagram type. For this reason, we changed the *SDPDM* here. In Figure 5.4 we can see the corrected version of the figure.

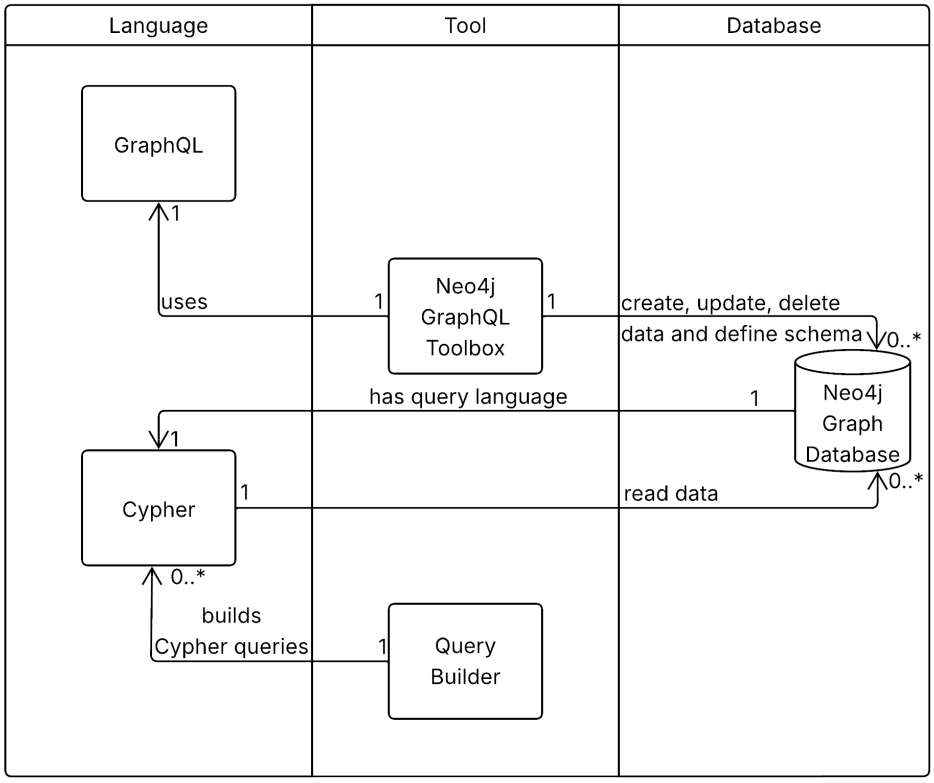

Figure 5.3.: The languages, tools, and the database which were used or developed during the process of this thesis. Explaining which language and tool is used to operate which tasks on the database.

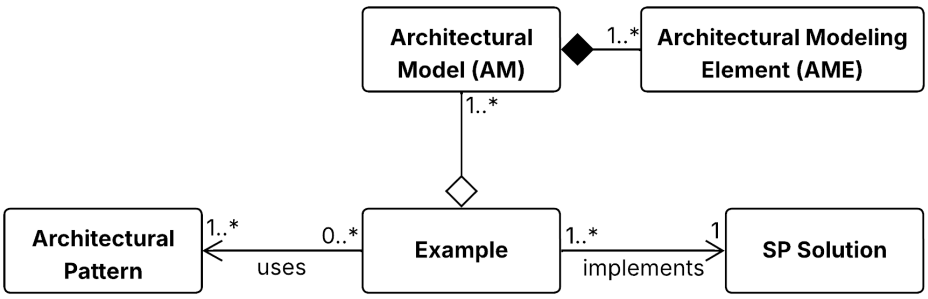

Figure 5.4.: The changed Figure to clarify the false aggregation from *AME* to *Example*. The original Figure is taken from Lammers' MT report [Lam24].

6. Implementation

In this chapter, we present the concrete implementation of the *SDPDM* in a Neo4j GraphQL schema. We first introduce the technologies used, introduce which *SP* was selected as an example *SP*, and then show the implementation of the *SDPDM* schema. We also present the implementation of a design choice we made and introduce the Query Builder, whose purpose is to improve the usability of the implementation. All artifacts, namely the Neo4j GraphQL schema of the *SDPDM*, an export of the graph database containing the example *SP*, and the accompanied descriptions, queries, and implementations are available in the public GitLab repository "Query Builder".[1]

6.1. Technologies

The database was hosted in Neo4j AuraDB, which is Neo4j's fully managed cloud graph database [Neo25b], with version 2025.08. The Aura console was used for reads and inspection in Cypher version 5. For creating a schema and data manipulation, we used the Neo4j GraphQL Toolbox, which used GraphQL type definitions and executed auto-generated GraphQL mutations to create, update, and delete data in the AuraDB instance. The Toolbox relies on the Neo4j GraphQL Library version 7. The introduction to Neo4j and the Neo4j GraphQL Toolbox can be found in chapter 5. Versions as observed on 5 September 2025.

6.2. Password-based Authentication (PBA) as a Suitable Security Pattern

As explained in section 3.4, we selected *PBA SP* as the reference *SP* to implement the *SDPDM*. It was chosen because it satisfies all three criteria mentioned in chapter 4. *PBA SP* covers all four *viewpoints* of the *SDPDM*, we have for each *viewpoint* modified UML diagrams from Lammers' MT report [Lam24] with detailed descriptions on every node, property, and relationship. Therefore, it was straightforward to choose *PBA* as *SP*, as it is described in such detail that it can be instantiated directly in the database, which satisfies our simplicity criteria of having sufficient time to instantiate it. The *PBA SP* is also relevant enough, as Jump Cloud's 2024 IT Trend Report found that 83% of organizations use *password-based authentication* for some IT resources, 83% with multi-factor authentication, still indicating that *PBA* is widely used [Jum25]. *PBA* is for this reason a suitable *SP* for our purpose of implementing the *SDPDM* [Lam24].

[1] `https://git.rwth-aachen.de/michi.zerbe/query-builder-for-sdpdm-neo4j-database`

6.3. Implementation of the Schema

In this section, we will describe the implementation of the schema. The schema is the persistence of the *SDPDM*, as it implements the four *viewpoints*, which define the elements and modeling rules for the *views* of an *SP* as described in chapter 3. We will show example parts of the schema and describe the concrete implementation done. As introduced in chapter 5, we use the Neo4j GraphQL Toolbox to define the *SDPDM* schema. The following listings present the Neo4j GraphQL type definitions. If additional context is needed, we refer to chapter 5 which introduces and explains the Neo4j GraphQL Toolbox. In chapter 4 we described that we used an incremental and iterative approach, where we implement the *SDPDM viewpoint* per *viewpoint* and also instantiate functional independent parts of the example *SP* after implementing the *SDPDM* for the individual *viewpoints*. However, since describing this approach here would lead to mixing implementation and demonstration, we just refer to the finished schema of the *SDPDM* and introduce certain aspects of it.

```
#SecurityPattern
type SecurityPattern @node {
  name: String!
  summary: String
  has_ConceptualView: [ConceptualView!]! @relationship(type: "has_ConceptualView", direction: OUT)
  has_DataView: [DataView!]! @relationship(type: "has_DataView", direction: OUT)
  has_BehavioralView: [BehavioralView!]! @relationship(type: "has_BehavioralView", direction: OUT)
  has_StructuralView: [StructuralView!]! @relationship(type: "has_StructuralView", direction: OUT)
}
```

Figure 6.1.: The part of the schema where we defined the type `SecurityPattern` which is a node with its properties and relationships.

To begin implementing the *SDPDM* we have implemented the type `SecurityPattern` which represents the *SP*, as we can see in Figure 6.1 we implemented it as a node for Neo4j, it contains the necessary `name`, indicated with the `!`, which means that this property cannot be null and a `summary`. It also contains the outgoing relationships that refer to the four *views* of the *SP*.

Continuing with the *Conceptual View* where we have the type *Role*, which contains the responsibilities necessary for the secure application of *Policies*. In Figure 6.2 we can see that it is implemented as a Neo4j node and contains all its elements, which Lammers defined in his MT report. For example, the `storageTag` which specifies if the *Role* needs certain type of storage to fulfill its responsibilities. This property requires the implementation of an enum, as we only have certain types of storage. The enum can be seen in Figure 6.3, containing the four storage types defined by Lammers for a *Role*. We can see that the enum name is the data type for this property, and it is also not in square brackets, indicating that only one type can be used for one *Role*. The implementation of the *Role* also contains the important relationship `required` between *Roles*, which states the dependency of another *Role* to fulfill its responsibilities [Lam24]. The inter-view relationships are also implemented in the schema, as we can see the *Role* implementation contains relationships, which are used in the *Behavioral View*. Namely, the `sent` and `received` relationships for the implementation

```
type Role @node{
  id: ID!
  name: String!
  description: String
  storageTag: StorageTag
  abstract: Boolean
  controlled: Boolean!
  policyPoint: [PolicyPoint!]
  requires: [Role!]! @relationship(type: "requires", direction: OUT, properties: "RequiresProp")
  requiredRole: [Role!]! @relationship(type: "requires", direction: IN, properties: "RequiresProp")
  decisionPointFor: [Policy!]! @relationship(type: "decisionPointFor", direction: OUT)
  enforcementPointFor: [Policy!]! @relationship(type: "enforcementPointFor", direction: OUT)
  informationPointFor: [Policy!]! @relationship(type: "informationPointFor", direction: OUT)
  #For Sequence Diagram Part of Behavioral Modeling Element
  type: ParticipantsType!
  messagesSent: [SequenceBehavioralModelingElement!]! @relationship(type: "sent", direction: OUT)
  messagesReceived: [SequenceBehavioralModelingElement!]! @relationship(type: "received", direction: IN)
}
```

Figure 6.2.: The part of the schema where we defined the type `Role` which is a node with its properties and relationships.

```
enum StorageTag {
  PERSISTENT_STORAGE
  CACHE
  SESSION_STORAGE
  SECRET_STORAGE
}
```

Figure 6.3.: The enum `StorageTag` for the correct values in the type `Role`.

of the UML sequence diagram structure for the *Behavioral View*.

The implementation of the *Behavioral View* contains implementing *Behavioral Model (BM)* and the *Behavioral Modeling Elements (BME)* of the *BM*. In Figure 6.4 we can see the implementation of the `SequenceBME`, which represent the UML sequence diagram part of a *Behavioral View*. We refer here to the proposed UML diagram representations of the *SDPDM views* in chapter 3. The `SequenceBME` is implemented as a Neo4j node, representing the message of a UML sequence diagram, which connects *Roles* interacting and showing their behavior. It can also cause *Events*, this can be seen in the implemented relationships. Relationships are not always necessary, but the Neo4j GraphQL Toolbox enforces the implementation of them in the schema as non-empty lists, which means it is a type in square brackets with `!` in and outside the brackets to indicate the value should not be null. We used an interface to implement the *BME*, we can see in Figure 6.5 the implemented interface, which contains properties which both UML diagrams share, namely the UML sequence and component diagram. In Figure 6.4 we can see that this type implements the interface and extends its properties and relationships. We can also see in Figure 6.5 that we implemented a union type which represents multiple types as one. In our context, it represents all types of data, which the *Behavioral View* could use.

As a last example, we want to show the implementation of the workaround which was introduced in the feasibility analysis in chapter 5. The implementation is used for the parts

```
type SequenceBehavioralModelingElement implements BehavioralModelingElement @node {
  id: ID!
  message: String!
  type: MessageType!
  interactionOperator: Operator
  parameters: [String!]
  sender: [Role!]! @relationship(type: "sent", direction: IN)
  receiver: [Role!]! @relationship(type: "received", direction: OUT)
  belongsToBlockParts: [InteractionBlockPart!]! @relationship(type: "contains_message", direction: IN)
  belongsToBlocks: [InteractionBlock!]! @relationship(type: "started_by", direction: IN)
  #can cause Events
  can_cause: [Event!]! @relationship(type: "can_cause", direction: OUT)
  #reference Behavioral Model
  ref_BehavioralModel: [BehavioralModel!]! @relationship(type: "ref_BehavioralModel", direction: OUT)
  #uses Data Elements
  uses_DataElement: [Data!]! @relationship(type: "uses_DataElement", direction: OUT)
}
```

Figure 6.4.: The part of the schema where we defined the type `SequenceBehavioralModelingElement` which is a node with its properties and relationships.

of the *SDPDM* that are represented as a UML sequence diagram, namely in the *Behavioral* and *Structural View*. This can also be seen in Figure 6.6 where we implemented the union type for both use cases. We implemented two Neo4j nodes, which represent the *Interaction Block* with its operator and referring to its parts with relationships, which have possible guard conditions. Nesting is implemented through relationships, which refer to their parents and children. The enum contains all types of operators for a UML sequence diagram. The types are taken from the "UML 2 Glasklar" book [RQS12].

6.4. Implementation of ID structure

We made a design choice that does not follow common graph database modeling patterns. In a graph database, node references are typically expressed through relationships. For our case, this would mean that the *view* node, for example, the *Conceptual View* node, would need to have a relationship to each element it contains. Another example is the *BM*, it would need to reference all *BMEs* it has. To solve this exponential growth of relationships, we decided to add an ID structure for the purpose of referencing elements, but only in parts of the *SDPDM* where we reference from top level elements like the *BM* referencing its *BMEs* or the *Data View* referencing its *Data Groups*. To view the complete ID structure for all elements considered, we refer to section A.1. Here we will cover only three examples to give an overview on how the ID structure looks like. In the following, we list the examples and their ID structure:

- Conceptual View
  - `cv_##`
- ComponentArchitecturalModelingElement (ComponentAME)
  - `sv_##-am_##-name`

```
#Behavioral Modeling Element
union Data = OriginField | ReferenceField | TransformationField | NormalDataGroup
| InherantDataGroup | AggregationDataGroup | CompositionDataGroup

interface BehavioralModelingElement {
  id: ID!
  message: String!
  #can cause Events
  can_cause: [Event!]! @declareRelationship
  #reference Behavioral Model
  ref_BehavioralModel: [BehavioralModel!]! @declareRelationship
  #uses Data Elements
  uses_DataElement: [Data!]! @declareRelationship
}
```

Figure 6.5.: The union for a type `Data`, to represent every type of data element which can be referenced and the interface option for the different UML diagram types of the *BM*.

  - the first `##` is the number of the associated structural view.
  - the second `##` is the number of the associated architectural model.
  - `name` is the name of the component.
  - special case: `sv_##-am_##-##_name`
  - the third `##` references the elements which are only able to implement a role in this combination.
  - if part of multiple of these combination then of structure: `sv##-am_##-##_..._##_name`

- StructuralRole
  - `name`
  - `name` stands for the name of the structural role.
  - special case: `##_name`
  - the first `##` references the elements which are only able to implement a role in this combination.
  - if part of multiple of these combination then of structure: `##_..._##_name`

We take a closer look at the `ComponentAME` and `StructuralRole` ID structures. We first define why we have *Structural Roles*. In the context of an *AM* there are some Controller, Services, Stores and more that are part of the *AM*, but are not *AME*, because of that we needed the workaround to introduce the type `StructuralRole` as a node. For example, *Structural Roles* are in the UML component diagram elements that are within one component. The component is an *AME*, but as nesting is allowed also *AMEs* can be part of another *AME* (component). We now explain why we need the proposed ID structure. As

```
enum Operator {
  ALT
  OPT
  BREAK
  NEG
  LOOP
  PAR
  SEQ
  STRICT
  CRITICAL
  IGNORE
  CONSIDER
  ASSERT
}

union ModelingElements = SequenceBehavioralModelingElement | SequenceArchitecturalModelingElement
type InteractionBlock @node {
  id: ID!
  operator: Operator!
  parentBlocks: [InteractionBlock!]! @relationship(type: "parent", direction: IN)
  childBlocks: [InteractionBlock!]! @relationship(type: "child", direction: OUT)
  parts: [InteractionBlockPart!]! @relationship(type: "has_parts", direction: OUT)
  startingMessages: [ModelingElements!]! @relationship(type: "started_by", direction: OUT)
}

type InteractionBlockPart @node {
  id: ID!
  guardCondition: String
  parentBlocks: [InteractionBlock!]! @relationship(type: "parent", direction: IN)
  messages: [ModelingElements!]! @relationship(type: "contains_message", direction: OUT)
  childBlocks: [InteractionBlock!]! @relationship(type: "child", direction: OUT)
}
```

Figure 6.6.: The part of the schema where we defined the workaround for the interaction block of a UML sequence diagram.

explained in chapter 3, *AM* is part of an *Example*, and this *Example* implements the *Roles* of the *Conceptual View* of an *SP*. There are three possibilities:

1. One or more *Roles* are implemented by multiple *AMEs* or *AMEs* and *Structural Roles*.
2. One *AME* or *Structural Role* can implement multiple *Roles*.
3. One *AME* or *Structural Role* can implement one *Role*.

We need the proposed ID structure for the first case, as only multiple elements in combination are able to implement this one *Role* and we decided to choose the ID structure, as we can see in the second list, as a workaround to represent this context.

6.5. Query Builder

The Query Builder is a web-based tool for building Cypher queries for the *SDPDM* database. The main reason behind that is the usability of the *SDPDM* database, as we found that *SPs*

Figure 6.7.: The Query Builder website showing a Cypher query for a *Role* which is a decision and enforcement point for the same *Policy*.

contain a lot of data and therefore the Cypher queries got complex and the graph visualization got cluttered, which may appear overwhelming for non-experts. For the implementation of the Query Builder it was important to be able to represent the structure of the *SDPDM*, we are not implementing a Query Builder for any Neo4j graph database. We have a Query Builder that is specific for the *SDPDM* schema, which defined the nodes, relationships, and properties of the *SDPDM*. It is also important that the ID structure that we decided to model is usable in the Query Builder, as we can easily refer to components of the *views* by that. The main reason for the Query Builder is as already mentioned the usability, and therefore the Query Builder needs to have a well-chosen UI for the User, intuitive design and on-page information that the User can easily use and build Cypher queries with it. But we also need to restrict the User to only perform read operations, as we manipulate our data with the Neo4j GraphQL Toolbox.

6.5.1. Technologies Query Builder

We used Java 11 as the language for the Query Builder. The build tool for our project is Maven from Spring Boot 2.7.5 which was also used in the Backend. In the Frontend we used Bootstrap 5.1.3, jQuery 3.6.0 and Bootstrap Icons 1.8.1.

6.5.2. Example Queries of current state Query Builder

In this subsection, we show an example of the already implemented part of the Query Builder. A description of how to use and what is implemented for the Query Builder can be found in the GitLab repository.1

For example, we can see in Figure 6.7 a Cypher query which was built on the website of the Query Builder. The query returns every *Role*, which is a decision point and an enforcement point of the same *Policy*. We can see in Figure 6.8 the resulting graph. This shows that the Query Builder is able to set multiple restrictions that must hold for the same node.

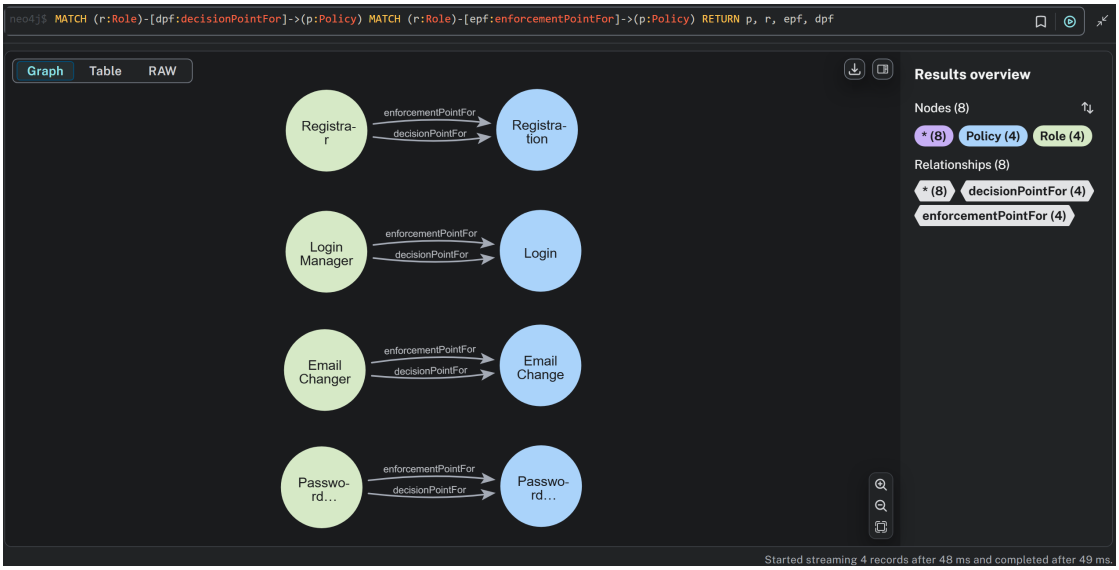

Figure 6.8.: The Neo4j result graph of the Cypher query for a *Role* which is a decision and enforcement point for the same *Policy*.

7. Demonstration

In this chapter we demonstrate the Neo4j GraphQL schema of the *SDPDM* from chapter 6, by showing certain parts of the instantiated *PBA SP* in the Neo4j property graph. As this chapter covers only the demonstration of the schema, we will show only Neo4j queries and the resulting part of the Neo4j property graph. We refer to section A.3 for chosen Neo4j GraphQL mutations, which instantiate the *PBA SP*. We introduced Neo4j and the Neo4j GraphQL Toolbox in chapter 5, we refer to that chapter for further details on Neo4j and the Neo4j GraphQL Toolbox. To demonstrate the Neo4j GraphQL schema we will show the instantiated *PBA SP*. We will begin by showing the complete *PBA SP*, then we will go into more detail and show example parts of every *view* of the *PBA SP*. We will also demonstrate certain implementations that we have done, such as an example for the ID structure we have implemented in chapter 6.

7.1. Instantiating *PBA SP*

In this section, we will demonstrate the *PBA SP* instantiation. In chapter 4 we described an incremental approach for the instantiation of an example *SP*, in this demonstration we will not demonstrate the instantiation step by step, we will refer in the beginning to the complete *PBA SP* instance in the Neo4j graph database which conforms to the Neo4j GraphQL schema and we will show certain parts of the Neo4j property graph as described earlier in this chapter.

We begin by presenting the complete *PBA SP*. The *PBA SP* yields 286 nodes and 744 relationships and can be seen in Figure 7.1. The density and heterogeneity of the resulting property graph make ad-hoc querying difficult. Therefore, it motivates the implementation of the Query Builder mentioned in chapter 6 and the use of Query Templates by users of the database. An introduction and explanation of the Query Templates will follow later in this chapter.

Regarding the *Conceptual View* we have *Roles* and *Policies*, we want to demonstrate an example on how *Roles* and *Policies* as nodes look like in the Neo4j property graph and what relationships they have. In Figure 7.2 we can see the `Password Policy Verifier (PPV)` and `Password Resetter (PR)` nodes with the label *Role*. We can see on the right the properties of the `PPV` node, for example, we can see there that it is a `DECISION_POINT` for a *Policy*. In the property graph, we can then verify that the `PPV` node is the decision point for the `Password Reset` node with the label *Policy*, this can be seen by the relationship between these two nodes. We can also see that the `PR` node requires the `PPV` node to fulfill its responsibilities. The property graph also shows the *PBA SP* node and the node for the *Conceptual View*. To see how an example Neo4j GraphQL mutation would look like to instantiate the mentioned *PBA SP*, *Conceptual View*, *Role* and *Policy* nodes and

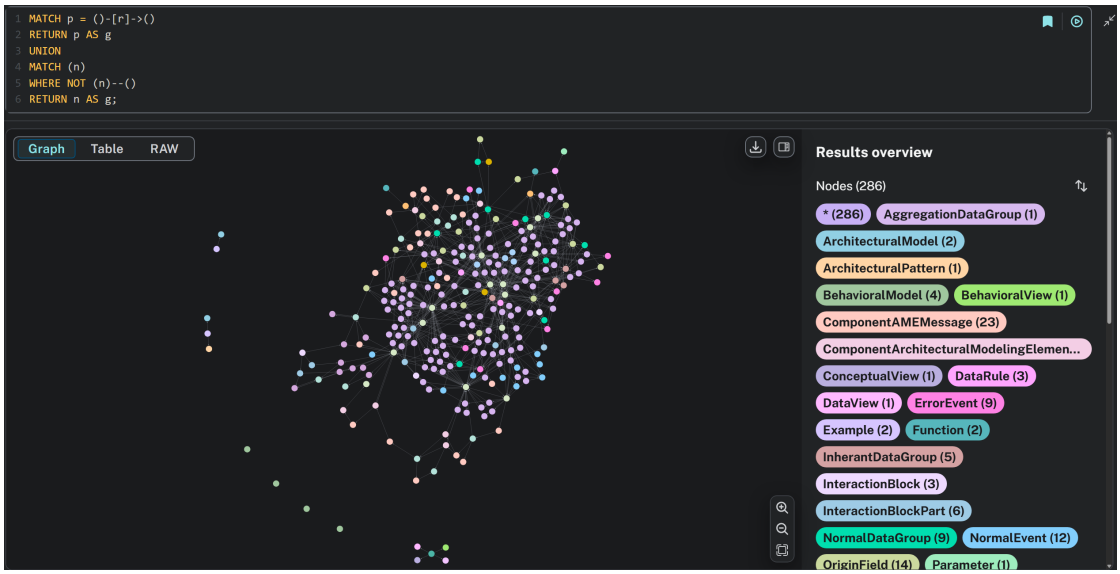

Figure 7.1.: The *PBA SP* instantiated in Neo4j.

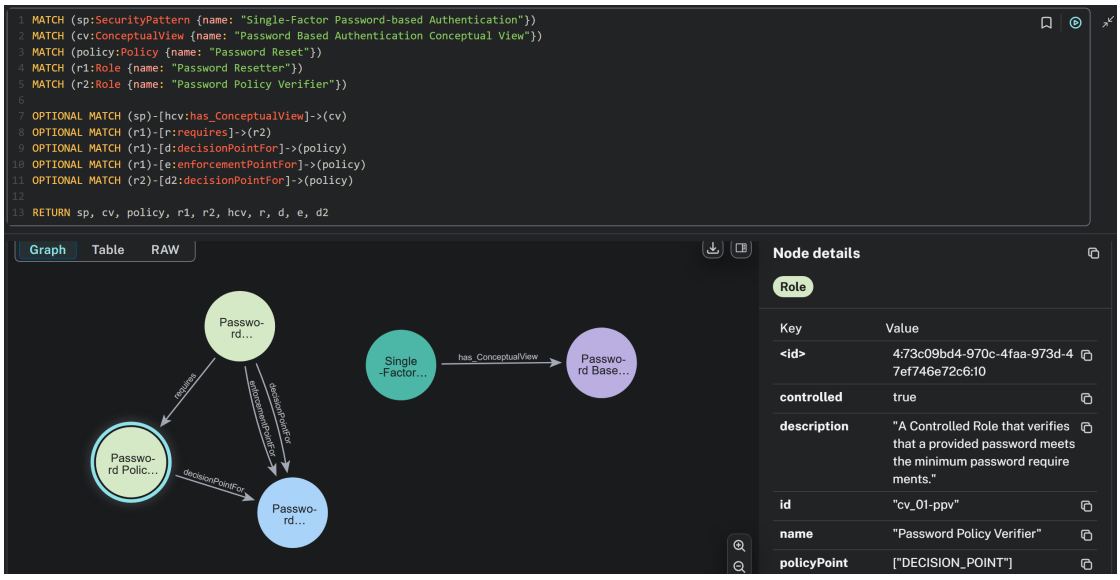

Figure 7.2.: The Neo4j *SDPDM* database property graph part which includes the *PBA SP* node, the *Conceptual View* node, the `Password Reset` *Policy* node, the `Password Resetter` and `Password Policy Verifier` *Role* nodes and the relationships between these nodes.

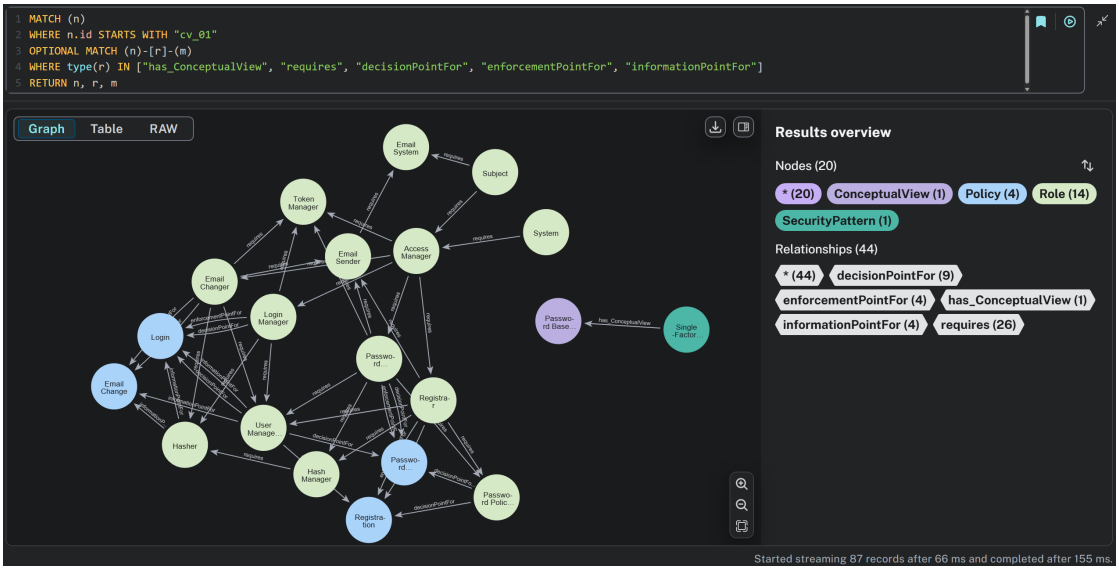

Figure 7.3.: The complete *Conceptual View* of the *PBA SP* in Neo4j.

relationships, we refer to section A.3.

We also want to demonstrate the complete *Conceptual View* of the *PBA SP* in Figure 7.3, it yields 14 *Roles* and four *Policies*, namely `Login`, `Password Reset`, `Email Change`, and `Registration`. These are the four parts into which we divided the complete *PBA SP*. They are also the increments used, as described in chapter 4.

For the *Data View* we want to demonstrate a special node. The `Function` node is used by transformation data fields, which are data fields that derive their data by applying a function to one or more data fields [Lam24]. In Figure 7.4 we can see the `SHA-256` node with the label `Function`. This node can contain examples. These examples are, pseudocode, descriptions, or practical examples to illustrate the intended transformation [Lam24]. In our case, the pseudocode has a code-style structure, so we must preserve the original input formatting. We refer to section A.3 for the Neo4j GraphQL mutation, which is able to preserve the original input format.

As already mentioned in chapter 6, we have implemented a workaround for UML sequence diagrams, which are used in, for example, the *Behavioral View*, as proposed in chapter 3 by Lammers. We want to demonstrate the interaction block structure as it looks in the Neo4j property graph. In Figure 7.5, we can see the nodes that are part of an interaction block. This example shows a part of the `Login` *BM* and its *BME* represent messages in a UML sequence diagram. The interaction block was started by a message, and all subsequent messages are contained in one of the two parts of the interaction block. The parts of the interaction block are the blue nodes.

We also want to demonstrate the implemented ID structure as mentioned in chapter 6. In the *Structural View* we are implementing *Roles*, as the *view* serves to represent an example implementation solution for the *SP* [Lam24]. It can occur that only specific combinations

Figure 7.4.: The `SHA-256` *Function* node with the correctly formatted pseudocode.

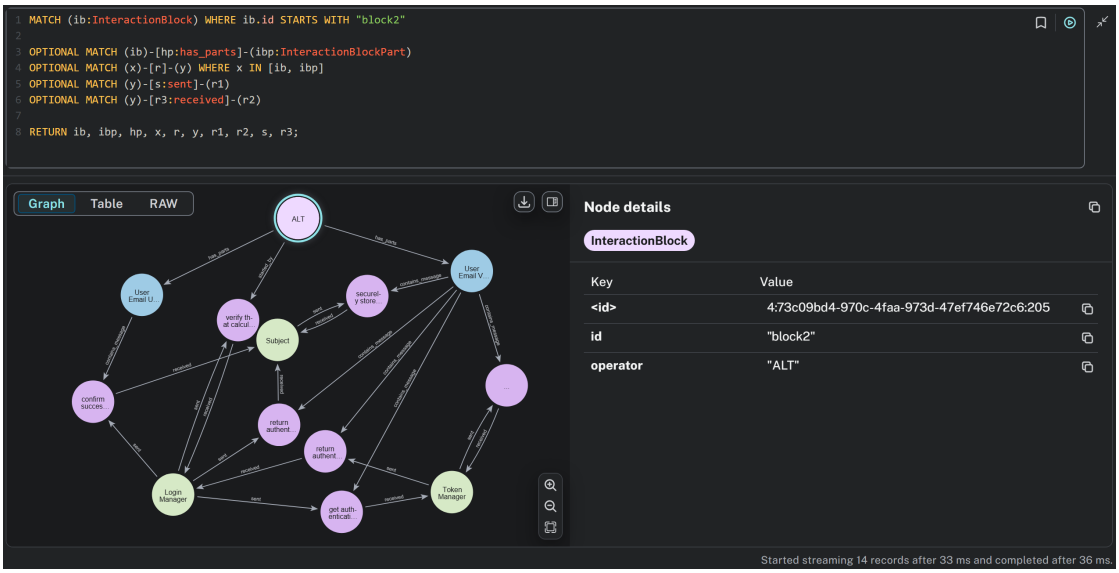

Figure 7.5.: The messages of the `Login` *Behavioral Model* that are inside an interaction block and the message which started the interaction block.

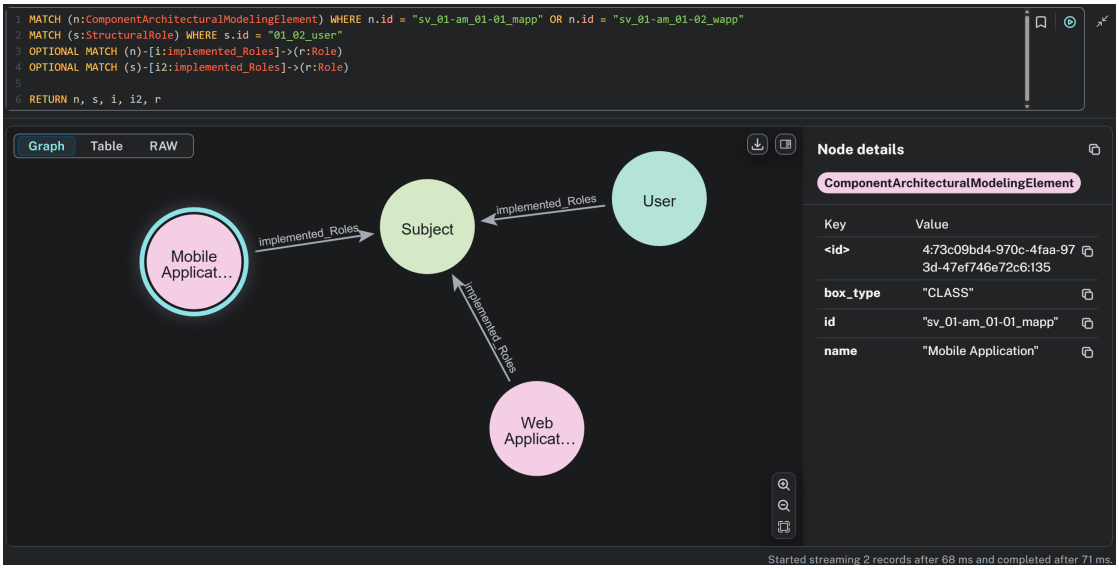

Figure 7.6.: The combination of (`Mobile Application` and `User`) or (`Web Application` and `User`) implement the *Role* `Subject`.

of *AME* and *Structural Roles* can implement a *Role*. In Figure 7.6 we can see one of these examples. Here only the combination of `Mobile Application` and `User` or `Web Application` and `User` can implement the *Role* `Subject`. On the right we can see the ID of the `Mobile Application` node with the label `ComponentAME`. `sv_01` refers to the *Structural View* node, `am_01` refers to the *AM* node this *AME* is part of, and the last `01` refers to another node with the label `ComponentAME` or `Structural Role` indicating that only the combination of these is able to implement the *Role* which is indicated by the `implemented_Roles` relationship. The `Web Application` has the same ID structure, only with `02` at the end. The ID of the `User` node with the label `Structural Role` starts with `01_02`, as it needs to represent that it is part of two combinations for a *Role* implementation.

7.2. Query Templates

To maintain usability of the *SDPDM* database and the instantiated *PBA SP*, we provide a set of Query Templates for users of the *SDPDM* database. The Query Templates are pre-created Cypher Queries, which show relevant parts of the *PBA SP*. For example, in Figure 7.3 we used one of these Query Templates, as it is the Query Template to show only the *Conceptual View* of the *PBA SP*. We have created the following Query Templates for the *PBA SP* that can be found in the Query Builder GitLab repository:1

- "Everything": Representing the complete *PBA SP*.
- Conceptual View
- Data View

- Behavioral View
  - "Password Based Authentication": Complete Behavioral View
  - Behavioral Model: Password Reset
  - Behavioral Model: Registration
  - Behavioral Model: Login
  - Behavioral Model: Email Change
- Structural View
  - "Password Based Authentication": Complete Structural View
  - Architectural Model: User Login
  - Architectural Model: Password Based Authentication Roles implemented

8. Case Study

In this chapter we validate the *SDPDM* schema by validating the instantiated *PBA SP*. We follow the validation approach introduced in chapter 4, by concretizing the idea of the approach and performing it. Due to time restrictions, this is only a manual and rather small validation of the instantiated *PBA SP* and the *SDPDM* schema.

8.1. Test of *PBA SP* instance

After being finished instantiating the *PBA SP* in Neo4j we need to validate that we inserted the *views* from Lammers' MT report [Lam24] correctly. Due to time restrictions, we decided to validate the *PBA SP* instance by hand and state a precise validation as a future work. We tested the validation of the instance by checking each *view* for itself. An evaluation of whether all inter-view relations, meaning the relationships between the *views*, are instantiated correctly for the *PBA SP* was infeasible due to time restrictions.

8.1.1. Conceptual View

The *Conceptual View* was manually checked by comparing each message, which was represented by the `requires` relation between *Roles* in our instance and each node represented by the node type *Role* in our instance with its properties, to the *Conceptual View* of Lammers' MT Report [Lam24]. In Lammers' MT Report we got 14 nodes, in our instance we got 14 *Role* nodes. For example the *Role* `Password Resetter` has the properties to have the storage tag `session storage`, it is an `Enforcement` and `Decision Point` for the `Password Reset` *Policy* and it has outgoing `requires` relations to the `Password Policy Verifier` *Role*, `Email Sender` *Role*, `Hash Manager` *Role*, `Token Manager` *Role* and `User Manager` *Role*. All these properties and relations can be seen in Figure 8.1 to validate that we correctly instantiated the `Password Resetter` *Role* in our instance of the *PBA*. We did that with every *Role* and also evaluated the message attribute in the `requires` relationship. As a result, we conclude that we instantiated the data of the *Conceptual View* of *PBA* correctly by manually checking it.

8.1.2. Data View

We manually checked the *Data View* by verifying that each variant of each element of the *Data View* is present in our instance. For example, we checked whether we have every *Inherent Data Group*, in Lammers' MT report [Lam24] we have five *Inherent Data Groups*: `Password Reset Data`, `Registration Data`, `Initiate Email Change Data`,

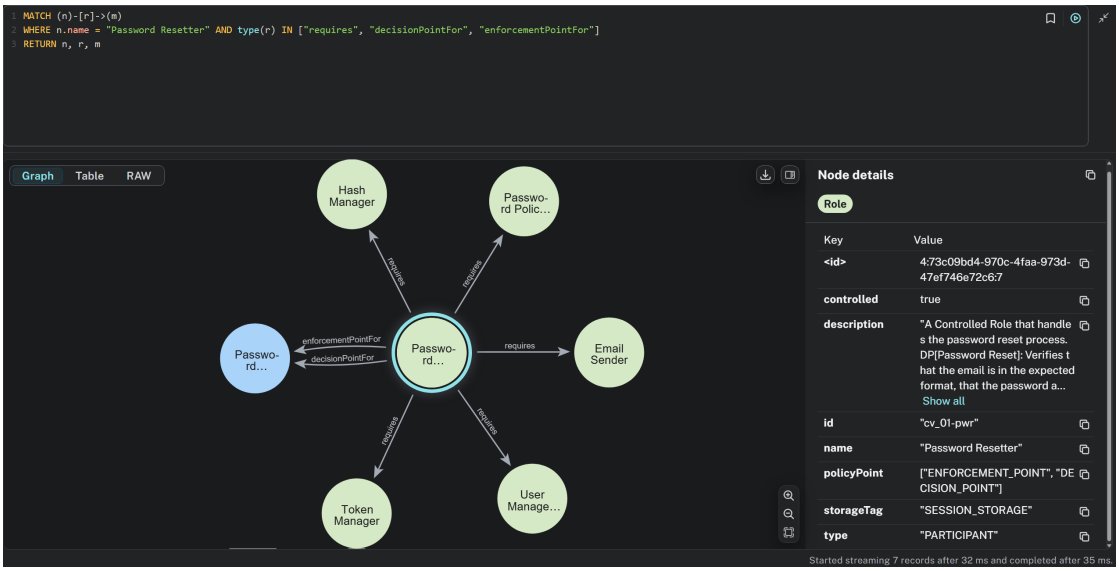

Figure 8.1.: The `Password Resetter` *Role* with its Node details and outgoing relationships.

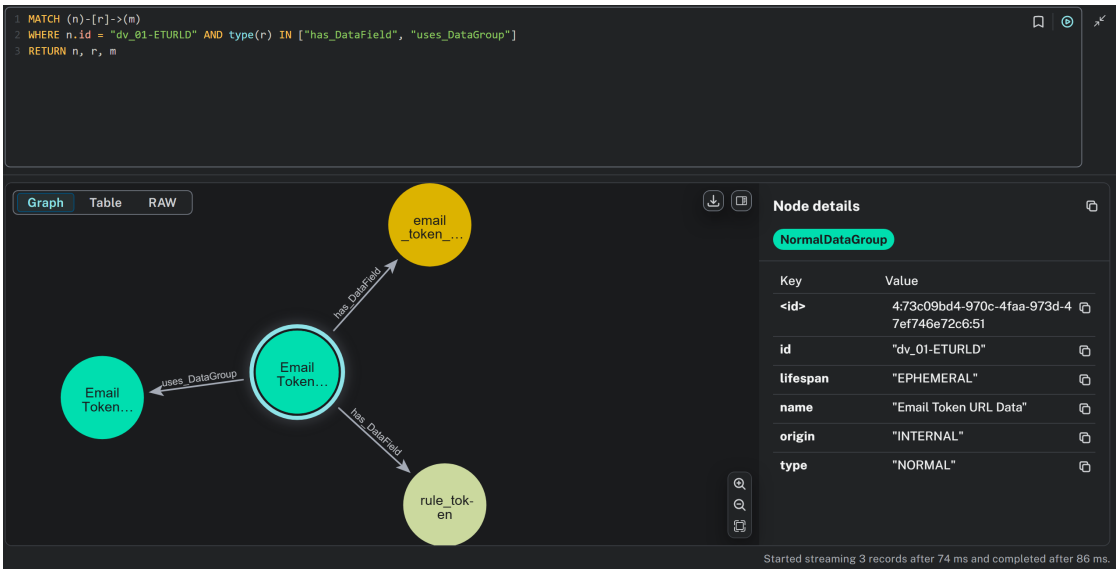

Figure 8.2.: The `Email Token URL Data` with the label `Normal Data Group` with its Node details and outgoing relationships.

Login Data and Email Change Token Session Data. All five of them are also instantiated in our instance. Important to note here is that we also have captured in our instance on which *Data Group* they inherit. Looking at the Password Reset Data, we see in Lammers' MT report [Lam24] that it inherits from Password Management Data, which is also instantiated like this in our instance by the relationship base_DataGroup from Password Reset Data to Password Management Data. A further element with variants is the *Data Field*, here we checked, for example, if we have instantiated every *Transformation (Data) Field*, in Lammers' MT report [Lam24] we see that there are two *Transformation (Data) Fields*: token_hash and password_hash. password_hash is present two times, but they just differ from the heritage of the password *Data Field*, therefore we only instantiated it one time, from which heritage the password *Data Field* comes is represented, by the relationship has_DataField, the relationship references, which *Data Group* has which *Data Field* and therefore, we can trace back the password *Data Field* heritage. Also important for the *Transformation (Data) Field* is that we have instantiated the *Function* they use correctly. The password_hash uses the *Function* password_hash in Lammers' MT report [Lam24]. We have correctly instantiated this *Function*. For the token_hash *Transformation (Data) Field* we have also instantiated the *Function* SHA-256 correctly. The *Functions* also have provided Pseudocode in Lammers' MT report [Lam24], we provide that Pseudocode by using the triple ", to keep the format of the Pseudocode untouched. After finishing the check of every variants of elements in the *Data View*, we checked further elements, such as *Data Rule*, the relationships and properties of every element of the *Data View*, for example the relationships and properties of Email Token URL Data, we can see in Lammers' MT report [Lam24] that it has Internal Data Origin, its *Lifespan* is Ephemeral, it has token as String and email_token_endpoint as String as *Data Fields* and it uses Email Token Config Data, because it has a *Reference (Data) Field* email_token_endpoint which comes from the Email Token Config Data endpoint *Data Field*. All these relationships and properties are correctly instantiated by our instance, as you can see in Figure 8.2. After finishing the manual check, we can conclude that we have correctly instantiated the data of the *Data View* of *PBA* in our instance.

8.1.3. Behavioral View

Here we counted the messages of each *Behavioral Model (BM)* and when we query the complete *Behavioral View* the number of messages should add up perfectly, because the *Behavioral View* contains each *BM*. In our case we have in the *BM* Registration 29 *Sequence Behavioral Modeling Elements (SequenceBME)* which correspond to a message, this corresponds to the given modified Sequence Diagram of Lammers' MT report [Lam24] for the Registration *BM*. For the *BM* Login we have 15 *SequenceBME*, in Lammers' MT report [Lam24] we will count only 14 numbered messages, but in our instance we also instantiated the "..." self-message from Token Manager which is our 15th message. The *BM* Password Reset has 37 *SequenceBME* in our instance of the *PBA*. We have in the *BM* Password Reset a self-message "..." from Token Manager which is not counted in Lammers' MT report [Lam24], but which is instantiated in our instance and therefore we have one *Sequence Behavioral Modeling Element* more than stated in Lammers' MT report

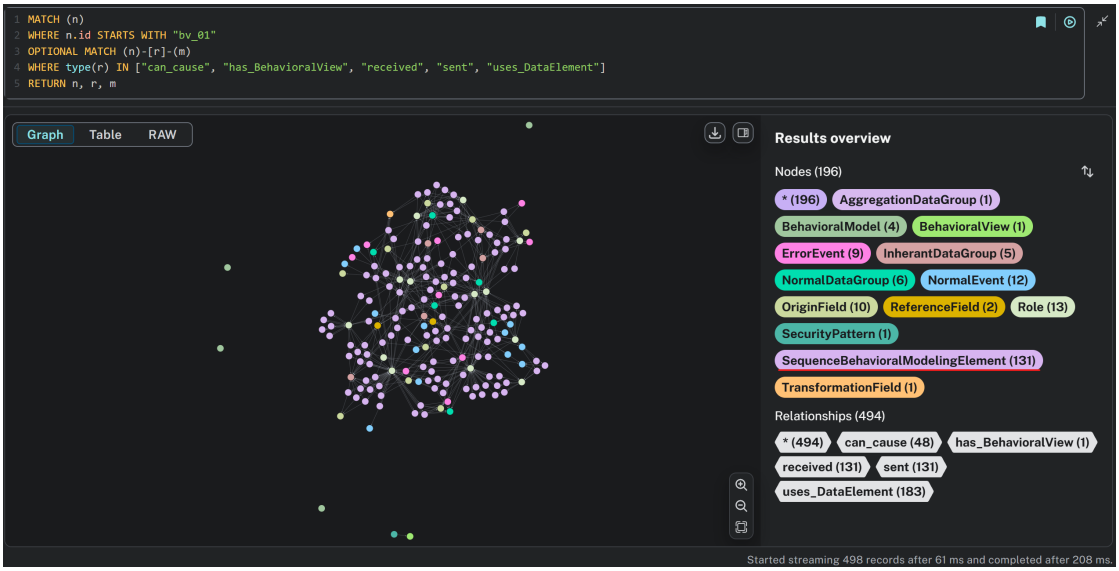

Figure 8.3.: Number of *SequenceBME* in the *Behavioral View*.

[Lam24]. In Lammers' MT report [Lam24] we have 47 messages in the `Email Change` *BM*, in our instance we have 50 *SequenceBME*, because we have the already mentioned self-message from `Token Manager` three times. Summing all *SequenceBME* from each *BM* up leads to 131 *SequenceBME* which can also be seen in Figure 8.3. A further check of elements in the *Behavioral View* were made by selecting a few random elements and checking the correct instantiation of the required properties due to time restrictions. Therefore, we manually checked whether we inserted the data of the *Behavioral View* of *PBA* correctly.

8.1.4. Structural View

For the *Structural View* we manually checked both *Architectural Models (AMs)* we have for *PBA*. For the *AM* `PBA Roles implemented`, we picked a few random *Component Architectural Modeling Elements (CompAME)* and *Component AME Messages* to check their properties and relationships. For example, the `Password Security Component`, in Lammers' MT report [Lam24] it implements the `Password Policy Verifier` *Role*, the `Hash Manager` *Role* and the `Hasher` *Role*. It also receives four *Component AME Messages*, one from the `Login Controller`, one from the `Registration Controller`, one from the `Reset Controller` and one from the `Email Change Controller`. The `Password Security Component` is part of the `User Service Component`, which is also needed to be instantiated. We have instantiated all the properties of the `Password Security Component` correctly in our instance of *PBA* as we can see in Figure 8.4. For the *AM* `User Login` we checked every element due to the small size of the UML Sequence Diagram. Important for the *Structural View* is that we also have *Structural Roles*, these are elements that are used as *Participants* or *Components* in *AMs*, but do not occur as *Role* in our *Conceptual View* due

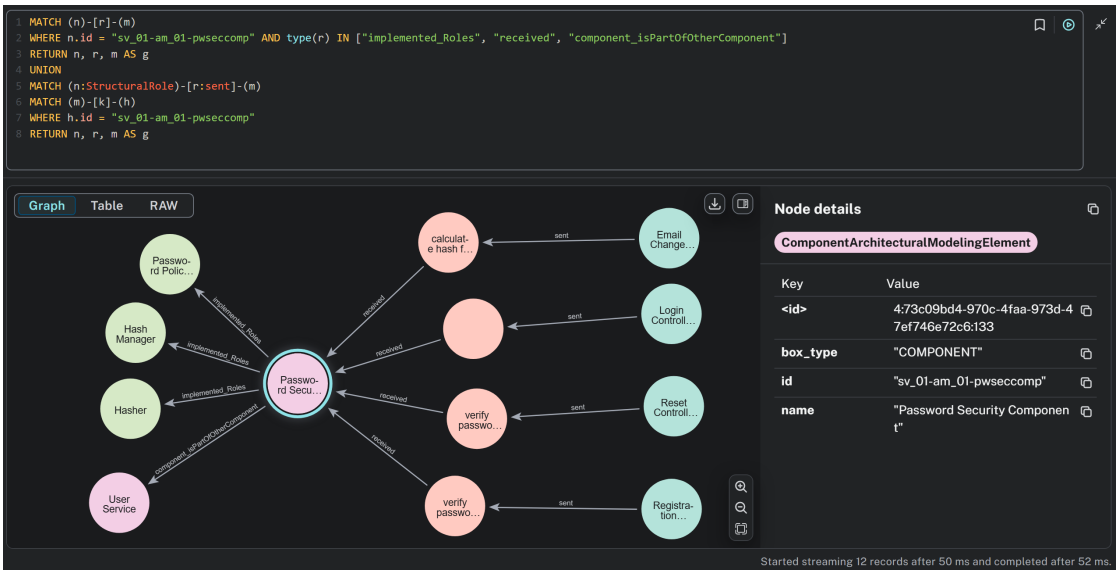

Figure 8.4.: The `Password Security Component` with its Node details and its relationships.

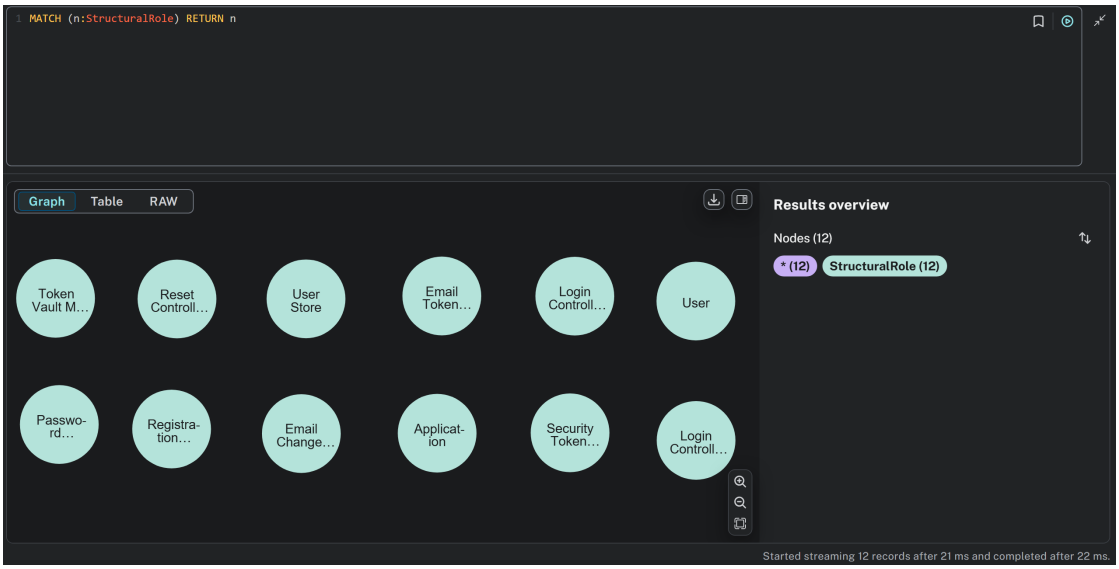

Figure 8.5.: The Structural Roles Nodes.

to them not being a *Role*, but of relevance to be depicted in the *Structural View*, such as a *Service* or a *Store*. Because of that, we also checked how many of these *Structural Roles* were in the two *AM* in Lammers' MT report [Lam24] and counted them distinctly. We counted 12 in Lammers' MT report: the `Application`, the `Login Controller`, the `Security Token Service`, the `User Store`, the `Registration Controller`, the `Reset Controller`, the `Email Change Controller`, the `Email Token Store`, the `(Protected) Services`, the `Email Service`, the `Password Service` and the `Token Vault Manager`. All *Structural Roles* are instantiated in our instance of the *PBA* as we can see in Figure 8.5. We manually checked the *Structural View* and inserted the data of the *Structural View* of *PBA* correctly.

9. Discussion

In this thesis, our goal was to implement the *SDPDM*. We did that by implementing a schema, as described in chapter 6 and by demonstrating the schema with an example *SP* that was instantiated into the Neo4j database, as described in chapter 7. In this chapter, we critically evaluate the implementation of the *SDPDM*. In addition, strengths, limitations, and potential improvements of the approach are discussed.

9.1. How the *SDPDM* can be implemented in a graph database

The research question of this thesis asks how a graph database can implement the *SDPDM* while preserving its semantics and supporting its fundamental concepts. We answer this by outlining the concrete mechanisms we used.

Using Neo4j together with the Neo4j GraphQL Toolbox, we mapped the *SDPDM* to a schema that implements the *viewpoint* hierarchy, including all relevant elements, modeling rules, and inter-view relationships. Concretely, we defined Neo4j GraphQL types that correspond to nodes in Neo4j and relationship fields that correspond to relationships in Neo4j. We specified constraints within the schema to enforce the required properties, value restrictions, cardinalities, and relationship directions. We also instantiated an example *SP*, which served as a schema validation. A targeted small case study then manually verified that the instantiated *SP* conforms to the intended elements, relationships, and constraints, thus preserving the semantics of the *SDPDM* for the example. To support usable access, we provide users with Query Templates and a prototype Query Builder. Although the Query Builder does not cover all the features needed, it already enables simple Cypher queries.

9.2. Limitations and Challenges

Neo4j was able to host the *SDPDM*, but several limitations became apparent during the process.

First, there is the already mentioned restriction to Neo4j GraphQL Toolbox for data manipulation. Although this might not be concerning, it would be better in a usability context to have one query language or tool for all CRUD operations. Non-experts can easily get confused in the beginning and create data using Cypher queries instead of Neo4j GraphQL mutations and by that make the *SDPDM* database not conform to the *SDPDM*.

Second, we were unable to test the UML activity diagram, and as we described in chapter 4, the refinement of the schema is a crucial part for implementing a correct schema, as the first

idea might not be the correct one. This is especially important because, for example, for the elements of the *views* we had a strict and detailed description by Lammers' MT report [Lam24], but for the *view* structure to be in a UML diagram style, we had to research the UML standards and had to interpret them for the schema. This can lead to mistakes in the implementation of the schema.

Third, the validation of the schema and the instantiated *PBA SP* were done manually in a small case study. We declare the schema as correctly implemented and the *PBA SP* instance as correct, but the manual validation is not substantive, as errors can occur during verification. For the schema, it is also not representative to have only one instantiated *SP*, since we might have only considered caveats in the schema implementation for this one *SP* instance and not for all *SPs* in general.

Fourth, we proposed the ID structure workaround that is unusual for graph databases, as we use an ID structure to get rid of high-level elements referencing their elements. The problem about that is that it is not consistent because we have decided that for a chosen number of elements to provide the ID structure, there are also elements that do not need the ID structure, as they do not meet our requirement of possibly many relationships that make the graph appear even more cluttered. Therefore, further data should be added with caution, as the ID structure should remain consistent for further *SPs* instantiated in the *SDPDM* database.

Fifth, the Query Builder does not satisfy all the important parts mentioned in chapter 6. To this point, we cannot query the introduced ID structure that we proposed in chapter 6. Therefore, non-experts are able to view every *view* of the *PBA SP* through the Query Templates, but the Query Builder for now is only capable of simple Cypher queries, as shown in chapter 6.

Finally, the process of manually inserting *SPs* into the database involves a lot of effort, as the scope of our example *PBA SP* is 286 nodes and 744 relationships. Considering that *PBA SP* is a rather simple *SP*, as we declared it as such in chapter 6, we underline the high effort the instantiation of *SPs* has. This challenge highlights the potential value of developing a tool or automation for *SP* instantiation and also highlights the need for a performance and scalability test of the *SDPDM* database.

9.3. Methodological Reflection

The process we described in chapter 4 was appropriate for the implementation of the schema and the instantiation of an example *SP*. The incremental approach was especially helpful as it allowed us to extend the schema and validate the correctness of the schema by instantiating examples in the database in small steps, and we were able to directly refine the schema if necessary. The selection of Neo4j as graph database engine was appropriate. The only restriction we had is that the constraints necessary for the implementation of the *SDPDM* had to be done in the Neo4j GraphQL Toolbox. But with the Neo4j GraphQL Toolbox we had a powerful tool to describe constraints in an efficient way and to insert, update, and delete data with auto-generated mutations. The tool extended the Neo4j database and made the work with the database more efficient. As we only considered Neo4j as a possible

candidate, because it met our requirements and performed well in our feasibility analysis, we do not have a reference to argue if another graph database, such as Microsoft Azure Cosmos DB, would be a better option for the purpose of implementing the *SDPDM*.

10. Conclusion and Future Work

This chapter summarizes the methodology and results of this thesis and highlights the directions for future research and development.

10.1. Summary

The goal of this thesis was to implement the *SDPDM*. We have done that by proposing an incremental approach to implement a schema and instantiating an example *SP* in the database to demonstrate the correctness of the schema. The first step was to extract technical requirements from the *SDPDM* that our graph database engine needed to meet. Neo4j was identified as a suitable graph database engine considering popularity and the positive feasibility analysis outcome. After selecting a suitable *SP*, we proposed the process of implementing the schema and instantiating the chosen *SP*. We implemented the schema viewpoint per viewpoint and instantiated the *SP* by splitting it into functionally independent parts. The development phase of the process was the initial step for creating the schema and adding an initial *view* to the schema. The refinement phase added all remaining *views* until sufficiently many were instantiated. It also refined the schema, if necessary. The concrete implementation of the schema consisted of defining the schema according to the *SDPDM*. We also implemented an ID structure to reduce relationships across the graph database. The instantiation of the *PBA SP* served as a demonstration that the schema is correct. We have shown examples of the instantiated *PBA SP*. As a last step, we answered the research question of this thesis and highlighted key limitations and challenges.

10.2. Main contributions

The main contributions of this thesis are that we first formalized the *SDPDM* as a Neo4j GraphQL schema and provided a reference implementation. This operationalization establishes a *SDPDM* conform way to create, collect, and review *SPs*, laying a foundation for future research on *SPs*. Second, we present a fully instantiated Password-based Authentication security pattern (*PBA SP*) that demonstrates how to use the schema and database in practice. Third, we conduct a small case study that manually validated both the schema and the *PBA SP* instance.

10.3. Future work

The implemented schema, which represents the *SDPDM* serves as a solid foundation for future work on persisting and collecting *SPs*. But there is still a need for improvement.

Considering the usability of the database, future work needs to improve and extend the Query Builder, as it enables the usage of the *SDPDM* database for non-experts. The important aspects proposed in chapter 6 can serve as a guide to extend the Query Builder. There should also be additional Query Templates for more *SP* instances, as they allow quick access to *views* and the most important parts of the *SP*.

There is a need to reduce the effort of instantiating *SPs* in the database. We got a solid foundation with the auto-generated GraphQL mutations from the Neo4j GraphQL Toolbox, but a tool or automation of the instantiation would decrease the effort and allow for easier extension of the database.

There is also the need to validate the schema and instantiated *PBA SP* in more detail, we have manually verified the schema and *PBA SP* instance, but manual verification is not substantive, as errors can occur during verification. For example, in future work a *SP* should be chosen which consists of the UML activity diagram to test the schema to see whether it is able to represent a UML activity diagram correctly. Further *SP* instances should be also added to the database to test the scalability and performance of the database.

A. Appendix

A.1. Implementation of the ID structure

The following list shows the considered candidates which, instead of referencing their elements through relationships, do so in our schema implementation by using IDs. The top level items in this list are the top level elements of the *SDPDM*. The low level items are then the relationships that we canceled by using the ID structure.

- For the Conceptual View:
  - `has_Role`
  - `has_Policy`
- Data View:
  - `has_DataGroup`
- Behavioral View:
  - `has_BehavioralModel`
- Behavioral Model:
  - `has_BehavioralModelingElement`
- Structural View:
  - `has_ArchitecturalModel`
- Architectural Model:
  - `has_ArchitecturalModelingElement`

The following list shows the ID structure for all elements in the schema. We understand that not all elements used in the following list were introduced in this chapter to this point, but for the purpose of keeping the ID structure in this format for future work on the *SDPDM* database it is important to write down every special ID structure. To view every type definition of the schema, we refer to the GitLab repository of the Query Builder, where all database files and also the schema are part of.1 The top level items are the names of the elements, the low level items the id structure they should have. # is a number from 0-9 for every # in this whole list.

- Conceptual, Data, Behavioral and Structural View
  - `xv_##`

  - x is c,d,b or s for the corresponding first letter of the views name.

- Policy
  - cv_##-p-name
  - name is the name of the policy.
  - p stands for policy.
  - ## is the number of the associated conceptual view.
- Role, Data Group, Data Field and Data Rule
  - kv_##-name
  - k is either c (conceptual view) or d (data view).
  - name is the name of the element.
  - ## is the number of the associated conceptual or data view.
- Behavioral and Architectural Model
  - hv_##-um_##
  - h stands for b (behavioral view) and s (structural view).
  - u stands for b (behavioral model) and a (architectural model).
  - the first ## is the number of the associated behavioral or structural view.
- SequenceBehavioralModelingElement and SequenceArchitecturalModelingElement
  - hv_##-um_##-##
  - h stands for b (behavioral view) and s (structural view).
  - u stands for b (behavioral model) and a (architectural model).
  - the first ## is the number of the associated behavioral or structural view.
  - the second ## is the number of the associated behavioral or architectural model.
- InteractionBlock
  - block##
- InteractionBlockPart
  - block##_part##
  - the first ## is the number of the associated block it is a part of.
- ComponentArchitecturalModelingElement
  - sv_##-am_##-name
  - the first ## is the number of the associated structural view.
  - the second ## is the number of the associated architectural model.

  - `name` is the name of the component.
  - special case: `sv_##-am_##-##_name`
  - the third ## references the elements which are only able to implement a role in this combination.
  - if part of multiple of these combination then of structure: `sv##-am_##-##_..._##_name`

- StructuralRole
  - `name`
  - `name` stands for the name of the structural role.
  - special case: `##_name`
  - the first ## references the elements which are only able to implement a role in this combination.
  - if part of multiple of these combination then of structure: `##_..._##_name`

- ComponentAMEMessage
  - `sv_##-am_##-M##`
  - the first ## is the number of the associated structural view.
  - the second ## is the number of the associated architectural model.
  - `M` stands for message.

A.2. How to build concrete Instances?

In this section, we explain how to build concrete instances using the schema created in chapter 6.

To create, update, and delete data in our database, we use the Neo4j GraphQL Toolbox. To read data, we use Cypher. To build Cypher queries, we can also use the Query Builder. In the GitLab repository of the Query Builder, we can also find the *SDPDM* schema and an export of the *SDPDM* database from Neo4j with the instantiated example *SP*. There is also a description on how to use all the proposed tools, for example, also a description on how to use the Neo4j GraphQL Toolbox, as we do here now.1

To use the Neo4j GraphQL Toolbox we first need to log in with "Neo4j" as user and the connection URL and password from our database. This only holds if we did not change the default user and if we use a Neo4j database. To be able to use the schema that we built in chapter 6, we need to build the schema in the Neo4j GraphQL Toolbox. After that we can choose to build and execute Queries or Mutations. In our case, we only need the GraphQL mutations, as they allow us to create, update, and delete data for our database [Neo25d]. We can select every type defined in the Explorer on the left side of the screen to use an auto-generated mutation for creation, update, and deletion of this type. When we choose,

Figure A.1.: An example create mutation in the Neo4j GraphQL Toolbox.

Figure A.2.: An example update mutation in the Neo4j GraphQL Toolbox.

Figure A.3.: An example delete and update mutation in the Neo4j GraphQL Toolbox.

for example, to create a *Role*, we then see the properties and relationships that the type *Role* can have. We can insert values into the properties and connect to other nodes in the database, by, for example, using the id of another *Role* to connect the *requires* relationship. We use a running example in the following description on how to use the three mutation types in the Neo4j GraphQL Toolbox that is introduced in the following. We have two *Roles*, the first *Role* is the `Hello World` *Role* which says "`Hello World!`" to the other *Role*. The `Hello World` *Role* is not a policy point for any *Policy*.

In Figure A.1 we can see an example of the explained creation of a *Role* with the name `Role Hello World`. It connects a relationship to the *Role* with the id `role2` and its message to `role2` is "`Hello World, Role2!`". In Figure A.1 we can also see the Explorer on the left side, which shows every valid property or relationship for a type *Role*. As we can see in Figure A.1 we defined the `Role Hello World` as `DECISION_POINT`. But we forgot to connect a relationship to a *Policy* declaring to which *Policy* the `Role Hello World` is a decision point. For this purpose, we can use the update mutation. In Figure A.2 we see an update mutation where we connect our `Role Hello World` by referencing it with its id `role1` to the *Policy* with the id `policy1` to declare that the `Role Hello World` is a decision point for the *Policy* with the id `policy1`. When we get to the case where we have to delete a property, relationship, or a complete node from the database, we can use the delete mutation. In our example case, we remember in our example specification that the `Role Hello World` *Role* is not a decision point for the *Policy* with the id `policy1`, as it is not a policy point for any *Policy*. In this case, we need to delete the relationship `decisionPointFor` between *Role* `Role Hello World` and the *Policy* with the id `policy1`. But we also need to delete the entry `DECISION_POINT` in property `policyPoint` for the *Role* `Role Hello World`, we can do that by using an update mutation. In the Neo4j GraphQL Toolbox we are able to concatenate multiple mutations, as we can see in Figure A.3. When we execute the mutation, we get a response.json on the right side of the window. It shows the name of the action

and the chosen information, requiring at least one information to be selected. An info is, for example, how much nodes got created by executing the mutation. The purpose of this example is to explain the use of auto-generated Neo4j GraphQL mutations. Although it is not specific to creating *SPs*, the approach can be adapted to instantiate an *SP* and to carry out data manipulation in the database.

A.3. Neo4j GraphQL Mutation Examples

In this section, we show example Neo4j GraphQL mutations, which are instantiating concrete parts of the *PBA SP*.

In Figure A.4, we see a create mutation in the Neo4j GraphQL Toolbox. It creates the *Conceptual View* node with the corresponding ID structure introduced in chapter 6. The newly created *view* node gets connected with the *PBA SP* node which can be seen in the update mutation. We also see the creation of one of the four *Policy* nodes which are contained in the *Conceptual View*, here it is the `Password Reset` *Policy* also with the corresponding ID structure.

In Figure A.5, two nodes with the type *Role* get created, namely the `Password Resetter` and `Password Policy Verifier` *Roles*. We can see the properties of the *Roles* and also the update mutation, which connects both *Roles*, by creating a relationship of the type `requires` from the `Password Resetter` *Role* to the `Password Policy Verifier` *Role*. The relationship contains a message stating that the `Password Policy Verifier` *Role* should validate the password strength for the `Password Resetter` *Role*.

In Figure A.6 we can see the mutation for the `SHA-256` node of the type `Function`. Important to note here is that we need to use the triple " to preserve the format of the pseudocode. The way this property looks in the Neo4j graph database can be seen in chapter 7.

query.graphql

```graphql
mutation MyMutation {
  createConceptualViews(
    input: {
      id: "cv_01"
      name: "Password Based Authentication Conceptual View"
      description: "Shows the Roles responsible for securely applying Password Based Authentication.[...]"
    }
  ) {
    info {
      nodesCreated
      relationshipsCreated
    }
  }
  createPolicies(
    input: {
      id: "cv_01-p-pw-reset"
      name: "Password Reset"
      description: """The user initiates the password reset by providing the email address associated
      with the registered account.[...]"""
    }
  ) {
    info {
      nodesCreated
      relationshipsCreated
    }
  }
  updateSecurityPatterns(
    where: { name: { eq: "Single-Factor Password-based Authentication" } }
    update: {
      has_ConceptualView: {
        connect: { where: { node: { id: { eq: "cv_01" } } } }
      }
    }
  ) {
    info {
      nodesCreated
      relationshipsCreated
    }
  }
}
```

Figure A.4.: The Neo4j GraphQL mutation to instantiate the `Password Based Authentication` Conceptual View and the `Password Reset` *Policy*. The *PBA SecurityPattern* node gets also its relationship to the new *Conceptual View* node.

query.graphql

```graphql
mutation MyMutation {
  createRoles(
    input: [
      {
        id: "cv_01-pwr"
        name: "Password Resetter"
        controlled: true
        type: PARTICIPANT
        description: "A Controlled Role that handles the password reset process.[...]"
        policyPoint: [DECISION_POINT, ENFORCEMENT_POINT]
        storageTag: SESSION_STORAGE
      }
      {
        id: "cv_01-ppv"
        name: "Password Policy Verifier"
        controlled: true
        type: PARTICIPANT
        description: "A Controlled Role that verifies that a provided password meets the minimum password requirements."
        policyPoint: DECISION_POINT
      }
    ]
  ) {
    info {
      nodesCreated
      relationshipsCreated
    }
  }

  updateRoles(
    where: { id: { eq: "cv_01-pwr" } }
    update: {
      requires: {
        connect: {
          edge: { message: "validate password strength" }
          where: { node: { id: { eq: "cv_01-ppv" } } }
        }
      }
    }
  ) {
    info {
      nodesCreated
      relationshipsCreated
    }
  }
}
```

Figure A.5.: The Neo4j GraphQL mutation to instantiate the `Password Resetter` and `Password Policy Verifier` *Role* with their *requires* relationship.

query.graphql

```
mutation MyMutation {
  createFunctions(
    input: {
      name: "SHA-256"
      function_input: "EmailTokenURLData.token"
      examples: """
      import hashlib

      input = "..."
      sha256_hash = hashlib.sha256(input.encode('utf-8'))
      hash_dig = sha256_hash.hexdigest()
      """
      examples_type: "Pseudocode"
    }
  ) {
    info {
      nodesCreated
      relationshipsCreated
    }
  }
}
```

Figure A.6.: The mutation to create the SHA-256 *Function* node with the triple ", to keep the Pseudocode in the correct format.

Bibliography

[Cyb22] Cybersecurity Ventures. *Cybercrime To Cost The World 8 Trillion Annually In 2023*. https://cybersecurityventures.com/cybercrime-to-cost-the-world-8-trillion-annually-in-2023/. Accessed: 2025-04-23. Oct. 2022 (cit. on p. 1).

[DB-25a] DB-Engines. *DB-Engines Ranking of Graph DBMS*. https://db-engines.com/en/ranking/graph+dbms. Accessed: 2025-08-24. Ranking updated monthly. 2025 (cit. on p. 21).

[DB-25b] DB-Engines. *Method of calculating the scores of the DB-Engines Ranking*. https://db-engines.com/en/ranking_definition. Accessed: 2025-08-24. 2025 (cit. on p. 21).

[DJS19] D. Deogun, D. B. Johnsson, and D. Sawano. *Secure By Design*. 1st. Manning Publications, 2019. ISBN: 978-1-61729-435-8 (cit. on p. 11).

[DSC16] G. Daniel, G. Sunyé, and J. Cabot. "UMLtoGraphDB: Mapping Conceptual Schemas to Graph Databases." In: *Conceptual Modeling (ER 2016)*. Vol. 9974. Lecture Notes in Computer Science. Springer, 2016, pp. 430–444. DOI: 10.1007/978-3-319-46397-1_33 (cit. on pp. 6, 7).

[Fer13] E. Fernandez-Buglioni. *Security Patterns in Practice: Designing Secure Architectures Using Software Patterns*. Wiley Software Patterns Series. Wiley, 2013. ISBN: 9781119998945 (cit. on p. 9).

[Gon+18] F. Gong et al. "Neo4j graph database realizes efficient storage performance of oilfield ontology." In: *PLOS ONE* 13.11 (Nov. 2018), pp. 1–16. DOI: 10.1371/journal.pone.0207595. URL: https://doi.org/10.1371/journal.pone.0207595 (cit. on p. 6).

[Gra25] GraphQL Foundation. *Introduction to GraphQL*. https://graphql.org/learn/. Accessed: 2025-09-18. GraphQL Foundation, 2025 (cit. on p. 22).

[Hey+07] T. Heyman et al. "An Analysis of the Security Patterns Landscape." In: *Proceedings of the Third International Workshop on Software Engineering for Secure Systems (SESS'07), in conjunction with ICSE 2007*. IEEE Computer Society, 2007, pp. 3–10. ISBN: 0-7695-2952-6. DOI: 10.1109/SESS.2007.4 (cit. on p. 9).

[Jum25] JumpCloud. *Multi-Factor Authentication Statistics*. https://jumpcloud.com/blog/multi-factor-authentication-statistics. Accessed: 2025-08-29. JumpCloud Inc., 2025 (cit. on p. 29).

[Lam24] D. Lammers. *Conception of a Security Design Pattern Catalog for Constraint-based Recommender Systems*. https://swc.rwth-aachen.de/theses/conception-of-a-security-design-pattern-catalog-for-constraint-based-recommender-systems/. Mar. 2024 (cit. on pp. 1–3, 5, 9–16, 19, 23, 24, 26, 27, 29, 30, 39, 43, 45, 46, 48, 50).

[LPK23] S. Lazarova, D. Petrova-Antonova, and T. Kunchev. "Ontology-Driven Knowledge Sharing in Alzheimers Disease Research." In: *Information* 14.3 (2023), p. 188. DOI: 10.3390/info14030188. URL: https://doi.org/10.3390/info14030188 (cit. on p. 6).

[Neo25a] Neo4j. *Graph database concepts*. https://neo4j.com/docs/getting-started/appendix/graphdb-concepts/. Accessed: 2025-08-24. Neo4j, Inc., 2025 (cit. on pp. 21, 23).

[Neo25b] Neo4j. *Neo4j AuraDB (Classic)*. https://neo4j.com/docs/aura/classic/auradb/. Accessed: 2025-09-18. Neo4j, Inc., 2025 (cit. on p. 29).

[Neo25c] Neo4j. *Neo4j GraphQL Library Introduction*. https://neo4j.com/docs/graphql/current/. Accessed: 2025-09-18. Neo4j, Inc., 2025 (cit. on p. 22).

[Neo25d] Neo4j. *Neo4j GraphQL Library: Mutations*. https://neo4j.com/docs/graphql/current/mutations/. Accessed: 2025-08-26. Neo4j, Inc., 2025 (cit. on pp. 23, 57).

[Neo25e] Neo4j. *Neo4j GraphQL Library: Toolbox*. https://neo4j.com/docs/graphql/current/getting-started/toolbox/. Accessed: 2025-08-26. Neo4j, Inc., 2025 (cit. on pp. 22, 23).

[Neo25f] Neo4j. *What is a graph database*. https://neo4j.com/docs/getting-started/graph-database/. Accessed: 2025-09-18. Neo4j, Inc., 2025 (cit. on p. 21).

[Neo25g] Neo4j. *What is Cypher*. https://neo4j.com/docs/getting-started/cypher/. Accessed: 2025-09-18. Neo4j, Inc., 2025 (cit. on p. 22).

[Neo25h] Neo4j. *What is Neo4j?* https://neo4j.com/docs/getting-started/whats-neo4j/. Accessed: 2025-09-18. Neo4j, Inc., 2025 (cit. on p. 21).

[NIS06] NIST. *Minimum security requirements for federal information and information systems*. U.S. Department of Commerce, Gaithersburg, MD, Tech. Rep. FIPS PUB 200. https://csrc.nist.gov/publications/detail/fips/200/final. Mar. 2006 (cit. on p. 9).

[OWA21] OWASP Foundation. *OWASP Top Ten 2021 – A04:2021-Insecure Design*. Accessed: 2025-09-15. 2021. URL: https://owasp.org/Top10/A04_2021-Insecure_Design/ (cit. on p. 1).

[RQS12] C. Rupp, S. Queins, and die SOPHISTen. *UML 2 glasklar: Praxiswissen für die UML-Modellierung*. 4., aktualisierte und erweiterte Auflage. München: Carl Hanser Verlag, 2012, p. 580. ISBN: 978-3-446-43057-0 (cit. on p. 32).

[RW11] N. Rozanski and E. Woods. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley, 2011. ISBN: 978-0-13-290612-8 (cit. on pp. 10, 12).

[Sch+06] M. Schumacher et al. *Security Patterns: Integrating Security and Systems Engineering*. Wiley Software Patterns Series. Wiley, 2006. ISBN: 9780470858844 (cit. on p. 9).

[Seh25] J. Sehbaoui. "Development of a Graphical User Interface for Viewpoint-based Security Design Pattern Descriptions." expected to be published in 2025. 2025 (cit. on p. 5).

[SLL25] A. R. Sabau, D. Lammers, and H. Lichter. *SecuRe – An Approach to Recommending Security Design Patterns*. 2025. arXiv: 2501.14973 [cs.SE]. URL: https://arxiv.org/abs/2501.14973 (cit. on pp. 1, 2, 9, 11).

[SLP24] I. Spasov, S. Lazarova, and D. Petrova-Antonova. "Alzheimers Disease Knowledge Graph Based on Ontology and Neo4j Graph Database." In: *Proceedings of Data Analytics and Management (ICDAM 2023)*. Ed. by A. Swaroop et al. Vol. 785. Lecture Notes in Networks and Systems. Springer, 2024, pp. 71–80. DOI: 10.1007/978-981-99-6544-1_6. URL: https://link.springer.com/chapter/10.1007/978-981-99-6544-1_6 (cit. on p. 6).

[Sof23] Software Construction, RWTH Aachen University. *SCAM – Security-Centric Architecture Modelling*. https://swc.rwth-aachen.de/research/projects/scam-security-centric-architecture-modelling/. Accessed: 2025-09-15. Project description. 2023 (cit. on p. 1).

[vYJ22] A. van den Berghe, K. Yskout, and W. Joosen. "A Reimagined Catalogue of Software Security Patterns." In: *Proceedings of the 3rd International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS '22)*. Association for Computing Machinery, 2022, pp. 25–32. DOI: 10.1145/3524489.3527301 (cit. on p. 9).

[YB] J. W. Yoder and J. Barcalow. *Architectural patterns for enabling application security*. in *4th Pattern Languages of Programming Conference*. https://www.plopcon.org/pastplops/plop97/Proceedings/yoder.pdf 3–5 Sept. 1997 (cit. on p. 9).