FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Implementation of OntoUML schemas in graph databases – case study
Student:	Bc. Jiří Zikán
Supervisor:	Ing. Michal Valenta, Ph.D.
Study program:	Informatics
Branch / specialization:	Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2022/2023

Instructions

The aim of the thesis is to investigate the suitability of graph databases and the implementation of integrity constraints for implementing models expressed in the OntoUML notation. There is work that has been done on the implementation of OntoUML constructs in relational databases. Graph databases may be better suited for this task. The result of this work can provide a basis for technical publications and also a basis for implementing a specific procedure for transforming an OntoUML schema into a graph database in the OpenPonk tool. The actual implementation of the transformation in OpenPonk can then be implemented as another bachelor's or master's thesis.

1. Familiarize yourself with the current state of the OntoUML specification, which is still evolving.
2. Familiarize yourself with at least two graph databases (such as Neo4j and Janus Graph) and their implementation of checking integrity constraints.
3. Together with your supervisor, propose suitable model examples of OntoUML schemas and propose their implementation in the chosen graph database (or both).
4. Execute and test the implementation.
5. Discuss the results.

Electronically approved by Ing. Michal Valenta, Ph.D. on 8 June 2022 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Implementation of OntoUML schemas in graph databases – case study

Bc. Jiří Zikán

Department of Software Engineering
Supervisor: Ing. Michal Valenta, Ph.D.

April 30, 2023

This thesis is dedicated to the loving memory of my dear grandfather,
RNDr. Milan Kočiřík, CSc. (1940-2023).

Acknowledgements

In the first place, I would like to express my sincere gratitude to my supervisor Ing. Michal Valenta, Ph.D., for his guidance, patience, motivation, and always an optimistic point of view. His expertise led me to the topic of my thesis, and his continuous support enabled me to complete it.

Besides my supervisor, I would like to express my special thank to my dearest Ing. Kristýna Panešová for her understanding, encouragement, and kindness, not only during the time I spent writing this thesis.

Also, I would like to thank my dear friend Ing. Jan Kuběna for his support and valuable advice related to the topic of this thesis.

Last but not least, my thanks go to all members of my family for the care, support, and background with which they have provided me. Without them, none of this would indeed be possible.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on April 30, 2023 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology
© 2023 Jiří Zikán. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Zikán, Jiří. *Implementation of OntoUML schemas in graph databases – case study.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Abstrakt

Diplomová práce se zabývá transformací ontologických konceptuálních modelů do grafových databází a implementací souvisejících integritních omezení. Jako první tato práce analyzuje vhodnost několika rozdílných grafových databázových systémů a zdůvodňuje volbu grafové databáze Neo4j jakožto nejvhodnější pro daný účel. Dále práce představuje obsáhlou proceduru pro transformaci OntoUML diagramů do seznamu specifických integritních omezení a popisuje implementaci těchto integritních omezení v Neo4j. Zmíněná procedura je následně použita pro instanciaci ukázkového OntoUML modelu. Instanciovaný model je také důkladně otestován s cílem zpětně ověřit správnost transformační procedury. Nakonec práce pojednává o dosažených výsledcích a zabývá se možným navazujícím výzkumem. Ve výsledku tato práce ukazuje, že je možné provést instanciaci validního OntoUML modelu do grafové databáze, ale bohužel není možné použít stejné principy instanciace pro všechny typy grafových databází.

Klíčová slova ontologické konceptuální modelování, transformace konceptuálních modelů, vývoj řízený modely, grafové databáze, OntoUML, Neo4j

Abstract

The master's thesis deals with the transformation of ontological conceptual models into graph databases and with the implementation of related integrity constraints. At first, the thesis analyzes the suitability of several different graph database systems and justifies the choice of the Neo4j graph database as the most suitable one for the given purpose. Next, it introduces a comprehensive procedure for the transformation of OntoUML diagrams into a list of specific integrity constraints and describes the precise implementation of these integrity constraints in the Neo4j. The mentioned procedure is subsequently used for the instantiation of an example OntoUML model. Instantiated model is also tested in order to verify the correctness of the transformation procedure. Finally, the thesis discusses the achieved results and addresses possible future work. As a result, this thesis shows that it is possible to instantiate a valid OntoUML model in a graph database, yet it is not possible to use the same principles of instantiation for all types of graph databases.

Keywords ontological conceptual modeling, transformation of conceptual models, model-driven development, graph databases, OntoUML, Neo4j

Contents

List of Figures

List of Tables

List of Listings

Introduction

With the rapid development of information and communication technologies, the demand for high-quality and high-complex software products is also rapidly increasing. An obvious consequence of the growing demand is the growing number of business entities involved in software development. Unfortunately, in practice, it can be observed that there exist many companies which are still unable to properly use software engineering methodologies and implement a truly systematic approach to software development. This fact can potentially have a very negative impact on the delivery time, price, and also on the overall quality of their software products.

Without any doubt, one of the most crucial instruments for a systematic approach to software development is conceptual modeling. Conceptual modeling is a key part of the methodology called *model-driven development* (MDD), but its importance in software engineering goes further beyond. Although there already exist well-founded methodologies for conceptual modeling that are based on coherent formal theories and allow their users to create comprehensive conceptual models, their use is still not part of the common practice of many, especially smaller software development companies. The real way to convince these software companies to use such methodologies is to improve the connection between the resulting conceptual models and their actual implementation in today's commonly used software development technologies, i.e., popular programming languages and database systems.

This master's thesis deals explicitly with the conceptual modeling language *OntoUML* and the related formal theory of *Unified Foundational Ontology* (UFO). Since researchers are well aware of the issues outlined above, research has already been conducted dealing with practical methods of transforming OntoUML conceptual models into relational databases [1][2]. Although relational databases are still among the most used database systems, other types of databases, such as graph databases, are also becoming very popular today thanks to their significant advantages in specific use cases, including social networking or data lineage.

This thesis aims to investigate the suitability of graph databases and the implementation of integrity constraints for the instantiation of models expressed in the OntoUML notation. In accordance with the set goal, the thesis attempts to verify the following research hypotheses:

H1 It is hypothesized that it is possible to instantiate a valid OntoUML model in a graph database and implement all necessary integrity constraints.

H2 It is hypothesized that it is possible to use the same principles of instantiation for all graph databases.

In order to verify the research hypothesis, the thesis primarily contains a case study that demonstrates the proposed way of how can be the example OntoUML model instantiated in the chosen graph database, including the implementation of all necessary integrity constraints resulting from the model and given by underlying theory. The knowledge gained during this case study will become the basis for future research in this area and specifically for the implementation of procedures enabling the fully automated transformation of the OntoUML conceptual models into graph databases. These procedures will subsequently serve not only software developers to ensure the integrity of data in their information systems but also domain experts in the design and validation of OntoUML models describing their domain of interest.

The structure of this master's thesis consists of eight subsequent chapters. The first chapter deals with the ontological conceptual modeling language OntoUML, its important constructs, and related formal theories, which are essential for the conducted case study. The second chapter describes the enumeration of selected graph database systems, their properties, capabilities, and their integrity constraints mechanisms. The third chapter introduces the example OntoUML model and related problem domain on which the instantiation in a graph database is later demonstrated. The fourth chapter analyzes the suitability of individual graph database systems and justifies the choice of the Neo4j graph database as the most suitable one for the instantiation of OntoUML models. The fifth chapter deals with the design of model instantiation – defines the mapping between OntoUML and Neo4j constructs, precisely describes the way of transformation of OntoUML constructs into the set of integrity constraints, and specifies methods for implementation of these integrity constraints in the Neo4j database system. The sixth chapter demonstrates the practical application of the proposed approach on instantiation of the example OntoUML model. The seventh chapter deals with the testing of the instantiated model and test-case-based verification that all necessary integrity constraints are preserved. Finally, the last chapter discusses the benefits and possible issues of the proposed approach and outlines where research in this area should go further.

Chapter 1

OntoUML

OntoUML is an ontology-driven structural modeling language grounded on the Unified Foundational Ontology. It is built as an extension of a popular general-purpose *Unified Modeling Language* (UML) [3]. Its models capture type-level structures expressed by modified UML class diagrams together with constraints written in the *Object Constraint Language* (OCL). The basis for the previous version of the OntoUML was initially proposed in Giancarlo Guizzardi's Ph.D. thesis in 2005 [4]. Recently, the previous OntoUML, together with the underlying UFO, was redefined by a set of publications into a newer version of OntoUML termed OntoUML 2.0 [5] [6] [7].

As an extension of UML, OntoUML defines a set of class and association *stereotypes* carrying a specific ontological meaning defined by the underlying theory. Together with a set of additional *syntactic constraints*, it is ensured that every valid OntoUML model is compliant with the UFO [5, sec. 4].

This modeling language is designed to facilitate the development of precise and expressive domain-specific models that are well-aligned with the underlying ontology. According to its authors, there is empirical evidence showing that OntoUML also significantly improves the quality of conceptual models without requiring additional effort to produce them [6, sec. 1].

1.1 Types and Individuals

In the UFO, there exist two distinct sorts of *things* – *Types* and *Individuals*. Types are abstract things whose purpose is to classify common characteristics of Individuals. Individuals are concrete things that instantiate Types and follow their characterizations. Unlike in the standard theory of *object-oriented paradigm* (OOP), in the theory of the UFO, an Individual can be an instance of multiple Types at the same time [8, sec. 3.1].

Types can be more closely divided into *Endurant Types* and *Relation Types*. Instances of Endurant Types exist in time and can change in a qualitative way while maintaining their *identity* [9, sec. 2]. In the OntoUML, Endurant

Types are represented as *classes*, while Relation Types are represented as *relationships* between classes [6, sec. 4].

From the perspective of formal theory, there exists an *instantiation* relation between the Type and its instantiating Individual. Types are defined as things that can be instantiated, while Individuals are things that cannot. Moreover, between Types, there exists a *specialization* relation. If a *subtype* is a specialization of a *supertype*, all instances of the subtype are also instances of the supertype. The theory of the UFO also states that anytime two Types have a common instance, they have to share a supertype or a subtype for this instance [10, sec. 2.1].

1.2 Identity and Sortality

The theory of UFO precisely distinguishes several different sorts of Endurant Types based on their modal properties. The first considered modal property is called *Sortality* and is closely related to the *principle of identity*.

Identity itself is the ability of every Individual to be distinguished from others. An Individual maintains this ability by following some principle of identity. The principle of identity is a definition of the way in which two instances can be determined to be the same Individual [4, ch. 4].

Based on the identity principle, there exist two different sorts of Endurant Types. *Sortals* are Endurant Types that provide an identity principle that all their instances follow. Sortals can further be divided according to whether they provide the identity principle themselves (termed *Ultimate Sortals*) or have inherited this ability from their supertype. On the other hand, the *NonSortals* are Endurant Types that do not provide any identity principle [11, sec. 3.1]. Its purpose is to aggregate common properties of different Sortals.

Axioms of the UFO state that every Individual must follow exactly one principle of identity. Several implications follow from this statement. From the perspective of Individuals, every Individual must be an instance of exactly one Ultimate Sortal. On the contrary, NonSortals themselves cannot have any direct instances. From the perspective of Endurant Types, Ultimate Sortals cannot specialize different Ultimate Sortals; all Sortals must specialize exactly one Ultimate Sortals; and NonSortals cannot specialize any Sortals [7, sec. 3].

1.3 Rigidity

Endurant Types are further divided by the second considered modal property called *Rigidity*. This modal property determines if instances of a given Endurant Type must remain its instances under various circumstances [4, ch. 4].

Based on the Rigidity, there exist three different sorts of Endurant Types. Firstly, instances of *Rigid* Endurant Types have to be their instances under all circumstances. It means that these Endurant Types classify their instances

necessarily. Secondly, instances of *AntiRigid* Endurant Types are not their instances under some circumstances. It means that these Endurant Types classify their instances accidentally. Lastly, some instances of *SemiRigid* Endurant Types have to be their instances under all circumstances, while others do not. It means that these Endurant Types classify some of their instances necessarily and some accidentally [8, sec. 3.3].

1.4 Existential dependence

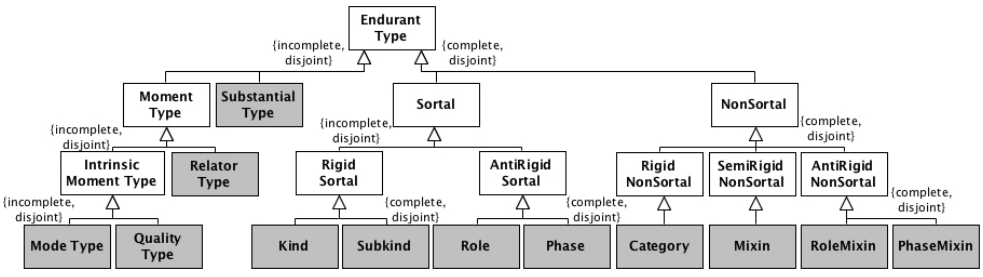
Orthogonally to the division described above, the UFO also divides Endurant Types according to whether they are *existentially dependent* or not. Existential dependence means that the existence of a dependent thing implies the existence of the thing on which it depends. Instances of *Substantial Types* are existentially independent, while instances of *Moment Types* existentially depend on other Individuals [10, sec. 1.1].

The UFO further distinguishes three sorts of Moment Types: *Relator Types*, *Quality Types*, and *Mode Types*. Relator Types are Moment Types whose instances have the ability to connect other instances of Endurant Types. They mediate relations between them, serve as so-called *truthmakers* and carry the properties belonging to the relations themselves. Quality Types are Moment Types whose instances represent intrinsic properties of related entities which have a structured value, i.e., can be mapped to some quality space. On the contrary, Mode Types are Moment Types whose instances represent intrinsic properties of related entities that have no structured value [3].

1.5 Class stereotypes

All ontological distinctions related to Endurant Types mentioned above are captured by the UML class diagram, which can be found in Figure 1.1.

Figure 1.1: Taxonomy of endurant types in UFO [5] – class diagram



Since the OntoUML follows these ontological distinctions, every class of the OntoUML diagram must be decorated with exactly one stereotype that reflects the ontological and modal properties of its instances. The class stereotypes corresponding to different properties are as follows [7, sec. 4]:

- **«kind»**: classes that are Substantial Ultimate Rigid Sortals whose instances are existentially independent regular objects,
- **«relatorKind»**: classes that are Moment Ultimate Rigid Sortals whose instances are existentially dependent Relators,
- **«qualityKind»**: classes that are Moment Ultimate Rigid Sortals whose instances are existentially dependent Qualities,
- **«modeKind»**: classes that are Moment Ultimate Rigid Sortals whose instances are existentially dependent Modes,
- **«subkind»**: classes that are Rigid Sortals whose instances follow the identity principle given by Ultimate Sortal,
- **«phase»**: classes that are relationally independent AntiRigid Sortals whose instances follow the identity principle given by Ultimate Sortal,
- **«role»**: classes that are relationally dependent AntiRigid Sortals whose instances follow the identity principle given by Ultimate Sortal,
- **«categoty»**: classes that are Rigid NonSortals whose instances may follow different identity principles,
- **«phaseMixin»**: classes that are relationally independent AntiRigid NonSortals whose instances may follow different identity principles,
- **«roleMixin»**: classes that are relationally dependent AntiRigid NonSortals whose instances may follow different identity principles,
- **«mixin»**: classes that are SemiRigid NonSortals whose instances may follow different identity principles.

In order to achieve conformity with UFO, OntoUML includes, in addition to stereotypes, a list of syntactic constraints based on individual axioms of this theory. These syntactic constraints are as follows [5, sec. 4]:

1. Every class must be decorated with exactly one stereotype.
2. Every non-Ultimate Sortal specializes an Ultimate Sortal.
3. An Ultimate Sortal cannot specialize another Ultimate Sortal.
4. A class cannot specialize more than one Ultimate Sortal.
5. A Rigid Type cannot specialize an AntiRigid Type.
6. A SemiRigid Type cannot specialize an AntiRigid Type.
7. A NonSortal cannot specialize a Sortal.
8. For every NonSortal, there must be a Sortal that specializes this NonSortal or specializes a NonSortal supertype common to both.

1.6 Relationship stereotypes

Similar to the Endurant Types, the UFO distinguishes a large number of different Relation Types based on their ontological properties. It is important to mention that the original ontological distinction regarding Relation Types presented in [4] can now be considered partially obsolete and has been changed by the new formal theory of UFO introduced in [6].

According to the new formal theory of the UFO, in the first ontological distinction, Relation Types are closely divided into *Internal Relation Types* (which can be defined in terms of the intrinsic properties of related entity) and *External Relation Types* (which cannot be defined in terms of the intrinsic properties of related entity) [6, sec. 3].

In the second ontological distinction, Relation Types are further divided into *Descriptive Relation Types* (which hold in virtue of some Moment of the related entity) and *NonDescriptive Relation Types* (which hold because of the related entity as a whole) [6, sec. 2.2].

In the OntoUML, these two ontological distinctions described above are reflected by the following *association* stereotypes [6, sec. 4]:

- **«characterization»**: associations that are External NonDescriptive and connect Quality or Mode Types to the Endurant Types in which their instances inhere,
- **«mediation»**: associations that are External NonDescriptive and bind Relator Types to all Endurant Types mediated by them,
- **«external dependence»**: associations that are External NonDescriptive and bind externally dependent Mode Types to Endurant Types on which their instances depend,
- **«comparative»**: associations that are Internal Descriptive; represent comparative relations; and their truthmakers are Internal NonDescriptive relations holding between common Qualities of the related entities,
- **«material»**: associations that are External Descriptive and hold in virtue of Relator Types or externally dependent Mode Types that are bound to the related entities,
- **«historical»**: associations that are External NonDescriptive; represent historical relations; and their truthmakers are *Events*.

Also, in this case, the OntoUML has to include a list of syntactic constraints to ensure compliance with the axioms of the UFO. Syntactic constraints regarding the Relation Types are as follows [6, sec. 5]:

1. Associations decorated as «material» must have a derivation association towards a class decorated as «modeKind» (one-sided relations) or «relatorKind» (other relations).

2. Classes decorated as «modeKind» and connected, through derivation, to some «material» relation must have a «characterization» relation towards one of the related classes and an «external dependence» relation towards the other.

3. Classes decorated as «relatorKind» and connected, through derivation, to some «material» relation must have a «mediation» relation towards each related class.

4. Classes decorated as «modeKind» and connected, through *part-of* relation, to some «relatorKind» must have a «characterization» relation towards one of the classes mediated by the Relator.

5. Classes decorated as «modeKind» and connected, through *part-of* relation, to some «relatorKind» must have an «external dependence» relation towards at least one of the classes mediated by the Relator.

6. Associations decorated as «comparative» must have a derivation association towards a class decorated as «qualityKind».

7. Classes decorated as «qualityKind» and connected, through derivation, to some «comparative» relation must have a «characterization» relation towards a related class or towards its superclass.

Besides the ontological distinctions of Relation Types mentioned above, the UFO also distinguishes several sorts of Relation Types belonging to the group of so-called *Meronymic Relation Types*. These Relation Types reflect different sorts of *Part-Whole* relations. In the OntoUML, they are represented as *compositions* [4, sec. 8.3]. However, since this thesis does not deal with these types of relations in its practical part, they are not discussed any further in this chapter.

Chapter 2

Graph databases

A *database* can be defined as an organized self-describing collection of interrelated data records. In order to define, store, maintain, share, and control access to the database, a specialized software called *database management system* (DBMS) is used [12, ch. 1]. There exist various types of DBMS that differ in the model according to which they work with processed data.

2.1 SQL versus NoSQL databases

From a historical point of view, the *relational model* can be denoted as the most significant one [13, ch. 1]. In this model, all data is represented as *relations*. A relation can be stored as a two-dimensional table-like structure, where the columns represent relation *attributes*, rows represent unique *records*, and specific values are located at the intersection of a given row and column. Relationships in the relational model are represented as values in the dependent table that references identifiers of records in the referenced table. For querying relational databases (RDB), the declarative domain-specific *Structured Query Language* (SQL) is typically used [14, ch. 1].

Relational databases have some characteristics that can be considered standard to the present day. First, they have a strictly defined schema expressed directly in the SQL language [13, ch. 10]. Second, in terms of transactional data processing, they guarantee properties expressed by the *ACID* acronym (atomicity, consistency, isolation, durability) [15, sec. 3].

Thanks to the properties mentioned above, relational databases are especially suitable for storing loosely connected data with a strictly defined structure. This type of data was common in information systems at the end of the last century. However, along with technological progress, nowadays, there is a need to process data falling into the category called *Big Data*. These data are characterized by their high volume, velocity, and variety, and relational databases reach their limits when processing them [16, sec. 1].

As an alternative to relational databases, a group of databases dubbed *NoSQL* ("Not only SQL") have emerged. NoSQL databases usually do not use SQL as their primary query language and are typically characterized by relaxing or simplifying some of the features standardly provided by relational databases [15, sec. 5]. For example, these databases often do not have any strictly defined schema (they are termed *schema-less*) and are thus able to adapt more easily to process highly-variable data. Also, instead of providing very strict ACID properties of transaction processing, some of them provide so-called BASE properties (basically available, soft state, eventually consistent) aimed at the high availability of data at the cost of temporarily limiting their consistency. Thanks to these facts, NoSQL databases achieve better scalability and higher performance than relational databases in certain situations and thus are more suitable for processing specific types of data which can be found in today's information systems [16, sec. 3].

Since the term NoSQL is very loosely defined, there is a large number of NoSQL databases with entirely different properties and features. From the point of view of their data model, NoSQL databases can usually be classified into the following four groups [17, appx. A]:

1. *Key-value stores* – consists of a distributed list of key-value pairs,
2. *Document stores* – consists of hierarchically structured documents,
3. *Column stores* – consists of tables whose rows contain arbitrary columns,
4. *Graph databases* – consists of a set of vertices and edges between them.

All NoSQL databases belonging to the groups enumerated above have their own advantages and disadvantages and are suitable for processing different types of data. Henceforward, this thesis continues to deal only with the group of graph databases.

2.2 Graph models and properties

Graph databases (GDB) are types of database management systems that represent and store data in graph structures as defined in Graph Theory. In comparison with other types of DBMS, graph databases are especially suitable for processing complex and highly interconnected data [18, p. 13]. For this reason, they are often applied in areas where information about interconnectivity or topology is more important than the underlying data itself – for example, social networking, bioinformatics, or semantic web [19]. However, since graphs can be considered extremely versatile, flexible, and expressive general-purpose structures, they can be used to represent almost any kind of real-world data in any application domain [17, p. 2].

One of the most commonly used data models in graph databases is called *property graph*. Data in this model are represented as a set of *vertices* and *edges*, along with *properties* attached to both of them. In the property graph, vertices usually represent entities in the data, and edges represent the connections between those entities. Edges are oriented, and the number of them between two vertices is not limited. Both vertices and edges are uniquely identified. Properties, which are key-value pairs, can be attached to vertices and edges to provide additional information about them [20, sec. 1]. The property graphs can be more closely divided into the following two types [21]:

1. *Labeled-property graph* – vertices and edges are assigned with text labels; the only purpose of the labels is to group vertices or edges; no additional logic is associated with them,

2. *Typed-property graph* – vertices and edges are assigned with types or classes; types define the structure of the vertices and edges within the graph, including the properties they can have.

Besides the property graph, there also exist other data models used in graph databases. For example, *hypergraphs*, in which an edge can connect any number of vertices, or *triples*, the *subject-predicate-object* data structure used in the semantic web [17, p. 207-208].

The properties of individual graph databases strongly depend on the way they store and process graph structures. From this perspective, they can be divided into *native* and *non-native* graph databases. Native graph databases process and store all data directly in the form of graph structures. On the contrary, non-native graph databases only offer a graph interface over some other non-graph databases, and graph structures need to be serialized into some other model, for example, a relational one [20, sec. 2.1].

Native graph databases provide significant performance advantages in processing queries that require graph traversal – for example when executing common graph algorithms such as *breadth-first search* (BFS) or *depth-first search* (DFS). It is because native graph databases are optimized in a way that traversing a single edge is possible in constant time. For comparison, in the case of relational databases, graph traversals over highly interconnected data would involve complex and performance-demanding join operations, which would make these traversals very inefficient [22].

One method that some native graph databases use to achieve constant time traversal is called *index-free adjacency*. With index-free adjacency, all vertices directly reference their connected neighbors without the need for any type of global adjacency index [23]. However, such an approach also has some disadvantages. Queries that do not use graph traversals can have worse performance. Also, it can be inefficient when vertices with a large number of edges are modified or deleted [24].

2.3 Neo4j

Neo4j is a graph database management system written in Java language that is designed to store, manage, and query large-scale graph data. It was first released in 2007 by Neo4j, Inc., and has since become one of the most widely used graph databases in the world [25]. The key features and properties of Neo4j are the following:

1. Property graph model – it uses the labeled-property graph data model and enables the assignment of multiple labels to a single vertex [26].
2. Native graph storage and processing – it stores data in a native graph format and utilizes index-free adjacency [27].
3. Schema-free database – it does not require any predefined data schema and provides a high level of flexibility [28].
4. High scalability – it is designed to handle large-scale graph datasets and has the ability to distribute data across multiple nodes [29].
5. ACID compliant – it provides strong transactional guarantees that ensure data integrity and consistency [30].
6. Cypher query language – it provides declarative query language with SQL-like syntax specifically designed for working with graph data [31].
7. Triggers and APOC library – it provides support for database triggers and an extensive library of useful procedures called APOC [32].
8. Graph algorithms – it includes built-in graph algorithms, such as the PageRank or shortest path, that can be used to analyze graph data [33].

2.4 JanusGraph

JanusGraph is another graph database management system that enables users to store and manage large-scale graph data. It is a project under The Linux Foundation and includes participants from Expero, Google, GRAKN.AI, Hortonworks, IBM, and Amazon [34]. It was created in 2017 as an open-source fork of the TitanDB [35]. JanusGraph addresses some of the limitations of other graph databases by combining their features with features of column-family stores. The key features and properties of JanusGraph are the following:

1. Property graph model – it uses the labeled-property graph data model but enables only one label to be assigned to a single vertex [36].
2. Non-native graph database – it uses a global adjacency index and provides a graph data model on top of the existing storage layer [37].

3. Various storage backends – it can store graph data into various backends such as Apache Cassandra, Apache HBase, and Oracle BerkeleyDB [38].
4. Data schema definition – it enables the definition of a data schema comprised of the edge labels, property keys, and vertex labels [39].
5. Distributed graph database – it is designed to scale out across multiple machines, supports horizontal scaling and partitioning [40].
6. Tunable consistency and durability – it offers configurable consistency and durability, thus allowing tuning for optimal performance [41].
7. Gremlin query language – it provides functional traversal language for querying graph data, with support for many graph algorithms [42].

2.5 TigerGraph

TigerGraph is a native graph database system that provides high-performance graph storage and processing for analyzing large and complex datasets. It was designed to handle real-time data and advanced analytics applications that require complex traversals of large-scale graphs. TigerGraph was developed by TigerGraph Inc. and released in 2017 as a commercial product [43]. The key features and properties of TigerGraph are the following:

1. Property graph model – it uses a typed-property graph data model that can handle a wide range of complex data types and relationships [44].
2. Distributed native graph database – it is a distributed native graph database optimized for processing and analyzing large-scale graphs [45].
3. Strict data schema – it has a strict definition of a data schema comprised of the types of entities, vertices, and edges [44].
4. High performance – it is designed for high performance and can handle real-time graph data queries with low latency [45].
5. ACID compliant – it provides strong transactional guarantees that ensure data integrity and consistency [46].
6. GSQL query language – it provides declarative query language that is optimized for querying graph data [47].
7. Lack of database triggers support – it does not provide any built-in support for database triggers or change data capture [43].
8. Advanced graph algorithms – it includes advanced graph algorithms such as parallel subgraph enumeration or k-core decomposition [48].

Chapter 3

Example OntoUML model

In order to carry out the intended case study, it is necessary to select a suitable problem domain and introduce an example OntoUML model, on which the instantiation in a graph database can be later demonstrated. There are two general requirements for the selection of a suitable problem domain. The first requirement is that any technically educated person without specific expertise should be able to understand the selected problem domain. The second requirement is that this problem domain has to be adequately complex to cover a sufficient number of OntoUML constructs.

The selected problem domain is related to the ontology of electrical devices and their electrical power sources. Final ontology is described by the proposed example OntoUML model that can be found in Figure 3.1. Below, this chapter also provides a detailed textual description of the individual entities and attributes found in the problem domain.

The example model consists of three different Kinds – `ElectricalDevice`, `DCPowerSupply`, and `Battery`. An `ElectricalDevice` represents any device that needs electricity to operate. For its function, the `ElectricalDevice` needs an energy source – such as a `DCPowerSupply` or a `Battery` – that is capable of providing a `nominalVoltage` and a `nominalCurrent`. A `DCPowerSupply` is a device capable of converting an AC `inputVoltage` to a nominal DC output voltage and is rated for a specific `maximalCurrent`. A `Battery` represents a set of electrochemical cells that are able to provide a `maximalVoltage` when they are fully charged and a `minimalVoltage` when they are discharged.

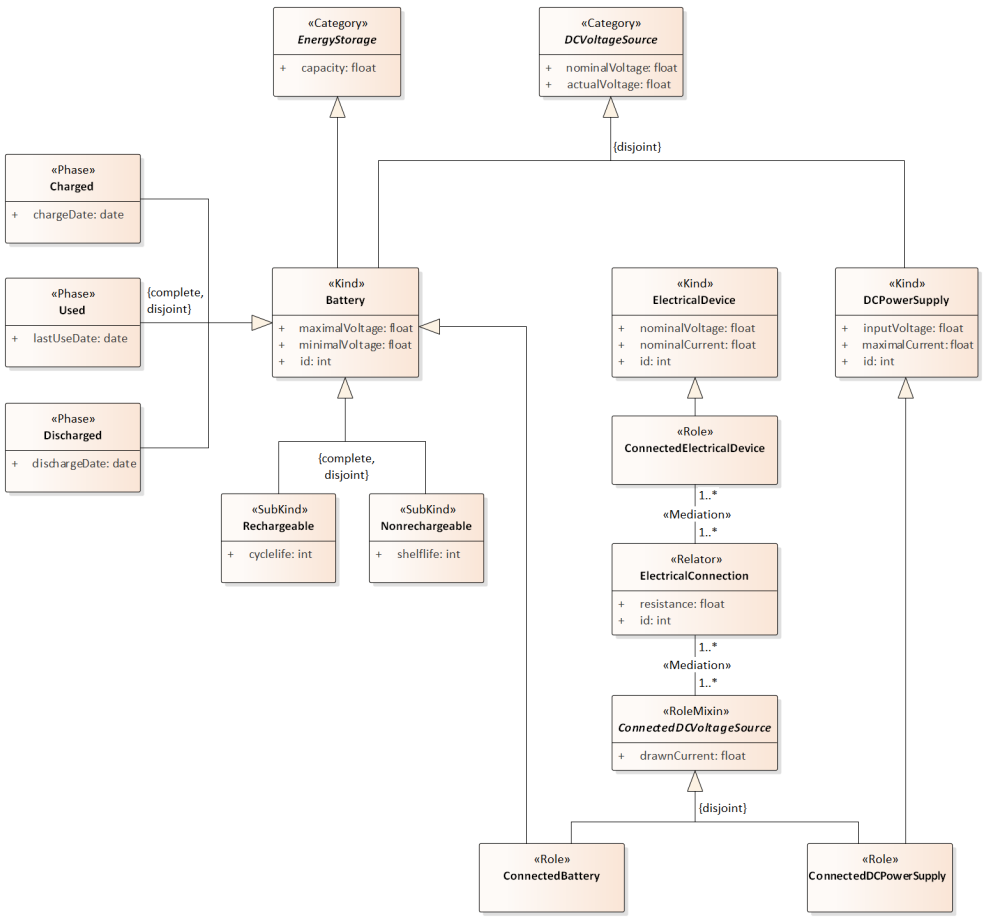
There exist two different Subkinds of `Battery`. A `Rechargeable` battery is a battery that can have its chemical reactions reversed and can be repeatedly recharged. The number of full charges of the `Rechargeable` battery during its lifetime is referred to as `cyclelife`. A `Nonrechargeable` battery, on the other hand, is a battery whose chemical reaction cannot be reversed, and therefore it can be discharged only once. The number of months that the `Nonrechargeable` battery can be stored without becoming unfit for use is referred to as `shelflife`.

A `DCPowerSupply` and a `Battery` belong to a `DCVoltageSource` Category. They both have their `nominalVoltage` and as DC sources, they are able to provide an `actualVoltage` under a specific load. The `Battery` also belongs to a Category of `EnergyStorage` and therefore it has a `capacity`.

A `Battery` can be in three Phases based on its `actualVoltage` – `Charged`, `Discharged`, or `Used`. In the case of a `Charged` battery, there is a certain `chargeDate` when was the `Battery` fully charged. Likewise, in the case of a `Discharged` battery, there is a `dischargeDate`. Also, in the case of a `Used` battery, there exists a `lastUseDate` when was the `Battery` last used.

In the case a `DCPowerSupply` or a `Battery` has been connected to an `ElectricalDevice`, they both act in a Role of a `ConnectedDCPowerSupply` or a `ConnectedBattery` respectively. Since both Roles share the same attribute, i.e., the `drawnCurrent` by a `ConnectedElectricalDevice`, this attribute and the common relationship are represented by a `ConnectedDCVoltageSource` RoleMixin. The physical connection itself, which has a specific `resistance`, is represented by an `ElectricalConnection` Relator.

Figure 3.1: Example OntoUML model – class diagram



Chapter 4

Analysis of database suitability

Graph databases are designed to store and manage data in a highly interconnected manner [18, p. 14]. Therefore, they should be well suited for representing various structures of specific types found in OntoUML models. However, there are many constraints resulting from the OntoUML notation, and it is questionable whether particular graph database systems and their integrity constraint mechanisms are able to preserve them correctly.

Before the example OntoUML model presented in the last chapter can be instantiated in a graph database, it is essential to analyze the capabilities and other properties of all considered graph database systems. After that, based on a set of appropriate criteria, the most suitable graph database system for this specific use case can be chosen.

4.1 Criteria of suitability

In terms of choosing the most suitable graph database system for OntoUML model instantiation, it is necessary to primarily consider what kind of graph model it uses and what kind of integrity constraint mechanisms are available. The first criterion determines the possible ways of mapping between constructs of the OntoUML and a particular graph database. The second criterion will considerably influence how the constraints resulting from the OntoUML notation can be implemented.

As described in chapter 2 of this thesis, the most common graph database models can be, in terms of their vertices, divided into labeled-property graph models and typed-property graph models. In the typed-property graph model, each vertex has an individual type assigned to it that determines what properties and possible relationships the given vertex can have [21]. If the inheritance between types, as an essential element of OntoUML, should be directly implemented in this graph model, it would have to be natively supported by the database engine. Otherwise, it would not be possible for a single vertex to be an instance of two different types at the same time. Even if the inheritance is

natively supported, there could be a problem with multiple inheritance, which is also common in OntoUML [49]. Without database engine native support, there still exists a possibility of using some workarounds similar to the approach for relational databases. For example, create one type for each class of the OntoUML model and then copy properties of a parent to all children [8]. In such a case, however, it would be necessary for the database engine to at least support a sufficiently strong integrity constraint mechanism to handle all possible inconsistencies that arise from this approach.

Compared to the first graph model discussed, the labeled-property graph model provides a significantly higher degree of freedom. In this model, labels are nothing more than text markers assigned to a given vertex, and no additional logic has to be associated with them [50, ch. 2]. Thus, it is theoretically possible to implement inheritance by mapping types to labels and then assigning multiple labels to a single vertex. Unfortunately, there also exist some problems associated with this approach. First, some database engines, although using the labeled-property graph model, do not support assigning more than one label to a single vertex. Second, since the assignment of a label to a given vertex does not ensure setting necessary properties with the correct data types, the database engine needs to provide a sufficiently strong integrity constraint mechanism to be possible to ensure it manually.

As mentioned above, the provided mechanisms of integrity constraints are another significant criterion when choosing a graph database suitable for OntoUML model instantiation. Many graph database systems provide standard integrity constraints such as `not null` or `unique` constraints, following the common practice of relational database systems [51][52]. However, these elemental constraints, especially in the case of graph databases with the labeled-property graph model, cannot ensure everything necessary to preserve the integrity of the instantiated OntoUML model. The only sufficiently effective tool for this purpose is database triggers, i.e., full-fledged code executed in case of certain data manipulation events. Thanks to database triggers, for example, it can be ensured that one label can only be assigned to a vertex simultaneously with another label. Following this, it is possible to create a specific hierarchy and additional logic associated with labels. Unfortunately, as discussed below, only some graph database systems support database triggers.

4.2 Suitability of considered databases

After considering all the above-mentioned criteria, the suitability of individual graph database systems can be evaluated. The TigerGraph database system has two fundamental shortcomings in this case. The first shortcoming is that this database system uses the typed-property graph model, but at the same time, it does not provide any native support for type inheritance [44]. The second major shortcoming is that this database system does not support any

mechanism of database triggers [43]. These two facts effectively prevent the use of TigerGraph for the instantiation of the OntoUML model. Therefore this database system cannot be considered suitable in this case.

The capabilities of the JanusGraph database seem to be much better for the intended use case. JanusGraph uses the labeled-property graph model [53, p. 523], which provides sufficient freedom to implement a custom label-based inheritance, including multiple inheritance. Furthermore, JanusGraph, or rather some of its storage backends, supports database triggers [35], which allows the implementation of almost any additional logic. But unfortunately, there still exists one major problem with JanusGraph. This database system does not allow multiple labels assignment for a single vertex [39]. Although it could be possible to implement the necessary inheritance in ways similar to relational databases, such an approach cannot be considered nearly as elegant compared to the inheritance based on multiple label assignments.

Fortunately, the last considered database system, Neo4j, does not have the same problem. It also uses the labeled-property graph model, but additionally, this database system allows any number of labels to be assigned to a single vertex [26]. Another advantage of Neo4j is that it provides a very convenient interface for database triggers within its library for creating user-defined procedures and functions called APOC. Thanks to this library, it is possible to create triggers directly in the native Cypher query language [32]. Based on these facts, the Neo4j database can be considered suitable for the OntoUML model instantiation. A summary overview of the described features of individual database systems can be found in Table 4.1.

Table 4.1: Comparison of graph database system features

Feature	TigerGraph	JanusGraph	Neo4j
graph model	typed	labeled	labeled
data scheme	yes	yes	optional
inheritance support	no	–	–
multiple labels	–	no	yes
database triggers	no	yes	yes

Chapter 5

Design of model instantiation

Based on the analysis of graph database suitability, the Neo4j database system is henceforward used in this case study for instantiation of the example OntoUML model. After choosing a suitable graph database, it is now possible to accurately design how to map and transform individual OntoUML constructs to the constructs available in the model of the selected database system and how to preserve all necessary constraints using Neo4j integrity constraint mechanisms.

5.1 Mapping between OntoUML and Neo4j constructs

For a description of the mutual mapping, it is first necessary to enumerate the individual constructs available in OntoUML and Neo4j models. As mentioned in the previous chapter, the Neo4j graph database system uses a labeled-property graph model. This graph model consists of four elemental constructs:

- *labels*,
- *vertices* (*nodes* in Neo4j),
- *edges* (*relationships* in Neo4j),
- *properties* (key-value pairs).

The graph itself consists of a set of vertices together with a set of edges between them. All edges in the graph are directed and have a start and an end vertex. Nevertheless, it is still possible to traverse the edges in both directions. Individual vertices can be assigned multiple labels to be divided into groups. Both vertices and edges can contain data in the form of key-value pairs called properties [17, p. 26-27].

The OntoUML is an extension of the UML modeling language, and its models can be expressed by modified UML class diagrams. These diagrams differ in the presence of ontologically motivated stereotypes associated with classes and relationships, but on the other hand, they omit some UML elements, such as interfaces or aggregations [4, ch. 8]. This thesis further deals with the following key UML class diagram constructs, as they are present in the representation of the example OntoUML model:

- *classes*,
- *stereotypes* (of classes or associations),
- *attributes* (with their *names*, *data types* and *multiplicities*),
- *generalizations* (with related *generalization sets*),
- *associations* (with their *multiplicities*).

Classes, as first-class citizens of the UML class diagrams, describe groupings of objects with the same structure, constraints, and semantics [54, p. 40]. Stereotypes give classes and other OntoUML constructs precise ontological meaning. Based on assigned stereotypes, classes represent different endurant types [7, ch. 4]. Attributes contained in classes ensure the presence and structure of data belonging to all instances of a given class [55, p. 42]. Generalizations are a type of relationship between classes where the special class (*subclass*) is based on the general class (*superclass*) and inherits its features. The special class can add new features but still remains compatible with the general class [56, p. 98]. Generalizations can also be combined into generalization sets. Generalization sets can have two different properties. The *IsComplete* property determines if an instance of the superclass must also be an instance of one of the subclasses. On the other hand, the *IsDisjoint* property specifies if any two subclasses can have common instances[57, p. 618]. Associations are a different type of relationship indicating that instances of one class are somehow connected to instances of another class [56, p. 197].

Although it might seem at this point that the transformation between the constructs of the OntoUML model and the labeled-property graph model could be relatively straightforward, it is essential to realize that the two models describe semantically different things at different levels of perception. While the OntoUML model describes the structure and interdependencies between types (classes), the labeled-property graph model captures concrete individuals (instances) and the data associated with them. Similar to how an object instantiates a class, a labeled-property graph model can become an instance of an OntoUML model. It is only necessary to ensure that all data represented by elements of the labeled-property graph model follow the rules given by the OntoUML model.

As outlined in the previous chapter, the crucial elements that enable the interconnection between both models are the labels. The first rule of proposed mapping is that all classes from the OntoUML model become labels in the labeled-property graph model. Unfortunately, this is the only mapping between constructs of both models that can be done directly. All other constructs of the OntoUML model have to become specific integrity constraints (ICs) in the Neo4j database system. The purpose of these integrity constraints is to ensure the correct form of instances represented by constructs from the labeled-property graph model.

Table 5.1: Mapping between OntoUML and Neo4j constructs

OntoUML constructs	Neo4j constructs
classes	labels
attributes	IC (ensuring properties on vertices)
stereotypes	IC (ensuring the combination of labels)
generalizations	IC (ensuring the combination of labels)
associations	IC (ensuring the presence of edges)

As can be seen in Table 5.1, several different types of integrity constraints arise based on the individual OntoUML model constructs. The exact meaning of different types of these integrity constraints and specific methods of their implementation in the Neo4j graph database system are discussed in detail in the following sections.

In the same way, as it is possible to describe the equivalents of OntoUML constructs in the Neo4j database system, it is also possible to infer the reverse meaning of the constructs found in the labeled-property graph model in the context of OntoUML. Firstly, all graph vertices (nodes) represent instances of classes determined by the labels assigned to them. Secondly, all edges (relationships) between connected vertices represent instances of binary associations between corresponding classes. And finally, all property names on vertices correspond to the names of particular attributes, and all property values represent values of these attributes. All the facts mentioned above are summarized in Table 5.2.

Table 5.2: Meaning of Neo4j constructs in the context of OntoUML

Neo4j constructs	meaning in the context of OntoUML
labels	classes
vertices (nodes)	instances of classes
edges (relationships)	instances of associations
properties (keys)	names of attributes
properties (values)	values of attributes

5.2 Transformation of OntoUML into integrity constraints

The description of the mutual mapping between OntoUML and Neo4j constructs revealed that some OntoUML constructs do not have a direct equivalent in the labeled-property graph model and, therefore, must be implemented as general integrity constraints. This section discusses in detail what ICs need to be added to the Neo4j database system to adhere to the semantics resulting from the individual OntoUML constructs. The set of all pseudocodes presented in this section can be understood as a procedure for the transformation of the OntoUML diagrams into Neo4j ICs. The specific methods of implementation of the proposed ICs in the Neo4j database system are later described in the section 5.3. An important assumption for the following considerations is that the input OntoUML diagram must be valid and meet all the syntactic constraints based on the UFO theory.

5.2.1 Integrity constraints imposed by attributes

Integrity constraints imposed by attributes must ensure that all vertices contain correct properties with valid values with respect to the attributes of classes corresponding to the labels these vertices are assigned. There exist three different types of integrity constraints imposed by the semantics of class attributes, their data types, and multiplicities.

The first type of integrity constraint ensures that the property values of vertices are valid with respect to the data types of corresponding class attributes. The pseudocode in Listing 1 iterates through all attributes of all classes in the input diagram and adds the `PROPERTY_MUST_BE_OF_DATATYPE` IC to ensure that the values of a given property on the vertices with a given label match a provided data type. As the input argument of the added IC, the name of a class represents the name of a label, the name of an attribute represents the name of a property, and the name of an attribute's data type corresponds to the equivalent data type from the Neo4j APOC library.

```
1 //INPUT: valid OntoUML diagram; OUTPUT: list of constraints
2 foreach (Class c in diagram.Classes)
3     foreach (Attribute a in c.Attributes)
4         Constraints.Add(PROPERTY_MUST_BE_OF_DATATYPE,
5             c.Name, //labelName
6             a.Name, //propertyName
7             a.DataType.Name //dataTypeName
8         );
```

Listing 1: Pseudocode adding ICs to ensure data type of attributes

The second type of integrity constraint ensures the presence of properties corresponding to the mandatory attributes. An attribute can be considered mandatory if the lower bound of its multiplicity is higher than zero. The pseudocode in Listing 2 iterates through all attributes of all classes in the input OntoUML diagram. If the attribute is mandatory, it adds the `PROPERTY_MUST_BE_PRESENT` IC to ensure the presence of a given property on the vertices with a given label. As the input argument of the added IC, the name of a class represents the name of a label, and the name of a mandatory attribute represents the name of a property.

For simplicity, since there are no such cases in the example OntoUML model, it is assumed that the upper bound of the attribute multiplicity can be at most one. Therefore this thesis does not deal with the constraints of attribute multiplicity upper bound in any way.

```
1 //INPUT: valid OntoUML diagram; OUTPUT: list of constraints
2 foreach (Class c in diagram.Classes)
3     foreach (Attribute a in c.Attributes)
4         if (a.Multiplicity.LowerBound > 0)
5             Constraints.Add(PROPERTY_MUST_BE_PRESENT,
6                 c.Name, //labelName
7                 a.Name //propertyName
8             );
```

Listing 2: Pseudocode adding ICs to ensure presence of attribute values

The third type of integrity constraint ensures the uniqueness of property values corresponding to the identity attributes. The pseudocode in Listing 3 iterates through all attributes of all classes in the input diagram. If the attribute is an identity attribute, it adds the `PROPERTY_MUST_BE_UNIQUE` IC to ensure the uniqueness of a given property on the vertices with a given label. As the input argument, the name of a class represents the name of a label, and the name of an identity attribute represents the name of a property.

```
1 //INPUT: valid OntoUML diagram; OUTPUT: list of constraints
2 foreach (Class c in diagram.Classes)
3     foreach (Attribute a in c.Attributes)
4         if (a.IsID)
5             Constraints.Add(PROPERTY_MUST_BE_UNIQUE,
6                 c.Name, //labelName
7                 a.Name //propertyName
8             );
```

Listing 3: Pseudocode adding ICs to ensure uniqueness of ID attributes

5.2.2 Integrity constraints imposed by stereotypes

Integrity constraints imposed by stereotypes must ensure that all vertices are assigned valid combinations of labels with respect to the ontological meaning given by stereotypes of corresponding classes. From the perspective of instances, two different types of integrity constraints imposed by the semantics of class stereotypes are considered.

The first type of integrity constraint follows an axiom of UFO theory, which states that every instance must have an identity [5, sec. 3]. This axiom always holds for all instances of Sortals since they either provide the identity themselves or have inherited it from their ancestor. In the case of NonSortals, it must be ensured that they cannot have any direct instances. All their instances must simultaneously be instances of a Sortal that is either a subclass of a given NonSortal or a subclass of a common NonSortal superclass.

The pseudocode in Listing 4 iterates through all classes in the input diagram. First, for each class that has a NonSortal stereotype (does not have an identity), it determines a list of related Sortals, which are classes that have an identity and, at the same time, their superclasses include either NonSortal itself or any of its superclass. Then, to simplify the resulting integrity constraint, the pseudocode further determines a list of intransitively related Sortals, which are related Sortals whose superclasses do not contain other related Sortals. Finally, the pseudocode adds the `LABEL_MUST_BE_IN_COMBINATION` IC to ensure that the label corresponding to the given NonSortal class is in a combination with any label representing an intransitively related Sortal.

```
1  //INPUT: valid OntoUML diagram; OUTPUT: list of constraints
2  foreach (Class c in diagram.Classes)
3      if (not c.Stereotype.HasIdentity)
4      {
5          relatedSortals = diagram.Classes
6              .Filter(r => r.Stereotype.HasIdentity)
7              .Filter(r => r.Superclasses
8                  .Exists(s => c.Equals(s) or c.Superclasses.Contains(s)));
9          intransitivelyRelatedSortals = relatedSortals
10             .Filter(i => i.Superclasses
11                 .ForAll(s => not relatedSortals.Contains(s)));
12         Constraints.Add(
13             LABEL_MUST_BE_IN_COMBINATION,
14             c.Name, //labelName
15             intransitivelyRelatedSortals.Map(i => i.Name) //otherLabelNames
16         );
17     }
```

Listing 4: Pseudocode adding ICs to ensure all instances have an identity

The second type of integrity constraint follows another crucial axiom of UFO theory, which states that every instance must have at most one identity [10, sec. 2]. If this axiom is to hold, it must be ensured that any instance cannot instantiate more than a single identity provider.

The pseudocode in Listing 5 iterates through all classes in the input OntoUML diagram. For each class that is an identity provider (has stereotype `Kind`, `Relator`, `Quality`, or `Mode`), the pseudocode first determines a list of all other identity providers in the input diagram. Then, in the case the list of other identity providers is not empty, the pseudocode adds the `LABEL_CANNOT_BE_IN_COMBINATION` integrity constraint to ensure that the label corresponding to the given identity provider is never in a combination with any other label representing another identity provider.

```
1  //INPUT: valid OntoUML diagram; OUTPUT: list of constraints
2  foreach (Class c in diagram.Classes)
3      if (c.Stereotype.ProvidesIdentity)
4      {
5          otherIdentityProviders = diagram.Classes
6              .Filter(p => p.Stereotype.ProvidesIdentity)
7              .OtherThan(c);
8          if(otherIdentityProviders.Count > 0)
9              Constraints.Add(
10                 LABEL_CANNOT_BE_IN_COMBINATION,
11                 c.Name, //labelName
12                 otherIdentityProviders
13                     .Map(p => p.Name) //otherLabelNames
14             );
15     }
```

Listing 5: Pseudocode adding ICs to ensure instances have a single identity

5.2.3 Integrity constraints imposed by generalizations

Integrity constraints imposed by generalizations must ensure that all vertices are assigned valid combinations of labels with respect to the generalizations between corresponding classes. The definition of generalization is included among the axioms of UFO theory. It states that if a superclass is a generalization of the subclass, all instances of the subclass are also instances of the superclass [7, sec. 3]. Furthermore, if generalizations are combined into generalization sets (GS), it is also necessary to ensure that all instances adhere to the properties of generalization sets (IsDisjoint and IsCovering properties). There exist three different types of integrity constraints imposed by the generalizations and generalization sets.

The first type of integrity constraint ensures that for a given generalization, all instances of a subclass are always also instances of a superclass. The pseudocode in Listing 6 iterates through all generalizations in the input OntoUML diagram and adds the `LABEL_MUST_BE_IN_COMBINATION` integrity constraint to ensure that every label representing subclass is always in a combination with the label representing corresponding superclass.

```
1  //INPUT: valid OntoUML diagram; OUTPUT: list of constraints
2  foreach (Generalization g in diagram.Generalizations)
3      Constraints.Add(
4          LABEL_MUST_BE_IN_COMBINATION,
5          g.Subclass.Name, //labelName
6          { g.Superclass.Name } //otherLabelNames
7      );
```

Listing 6: Pseudocode adding ICs to ensure generalizations between classes

The second type of integrity constraint ensures that if individual generalizations are combined into a generalization set with the IsDisjoint property, instances of one subclass are never the instances of another subclass from this generalization set. The pseudocode in Listing 7 iterates through all generalization sets in the input OntoUML diagram. For each generalization set with IsDisjoint property, the pseudocode further iterates through all subclasses in this generalization set and adds the `LABEL_CANNOT_BE_IN_COMBINATION` integrity constraint to ensure that the label representing one subclass in not in a combination with any other label representing another subclass in a given generalization set.

```
1  //INPUT: valid OntoUML diagram; OUTPUT: list of constraints
2  foreach (GeneralizationSet gs in diagram.GeneralizationSets)
3      if (gs.IsDisjoint)
4      {
5          foreach (Class c in gs.Subclasses)
6              Constraints.Add(
7                  LABEL_CANNOT_BE_IN_COMBINATION,
8                  c.Name, //labelName
9                  gs.Subclasses
10                     .OtherThan(c)
11                     .Map(s => s.Name) //otherLabelNames
12             );
13     }
```

Listing 7: Pseudocode adding ICs to ensure GS IsDisjoint property

The third type of integrity constraint ensures that if individual generalizations are combined into a generalization set with the IsCovering property, instances of a superclass are always also instances of at least one subclass of this generalization set. The pseudocode in Listing 8 iterates through all generalization sets in the input diagram. For each generalization set with IsCovering property, the pseudocode adds the `LABEL_MUST_BE_IN_COMBINATION` integrity constraint to ensure that the label representing superclass is in a combination with any label representing a subclass.

```
1  //INPUT: valid OntoUML diagram; OUTPUT: list of constraints
2  foreach (GeneralizationSet gs in diagram.GeneralizationSets)
3      if (gs.IsCovering)
4      {
5          Constraints.Add(
6              LABEL_MUST_BE_IN_COMBINATION,
7              gs.Superclass.Name, //labelName
8              gs.Subclasses
9                  .Map(s => s.Name) //otherLabelNames
10         );
11     }
```

Listing 8: Pseudocode adding ICs to ensure GS IsCovering property

5.2.4 Integrity constraints imposed by associations

Integrity constraints imposed by associations must ensure that the correct number of edges between vertices is present with respect to the associations between classes corresponding to the labels these vertices are assigned. The correct number of edges that must be present is given by the multiplicities at the individual association ends.

Several simplifications are made in terms of associations since only the constraints necessary for the instantiation of the example OntoUML model are discussed in this thesis. First, although associations of any arity can exist in UML, this thesis considers only binary associations. Second, since there is no association in the example OntoUML model whose multiplicity has a limited upper bound, this thesis does not deal with the constraints ensuring that the maximum number of edges has not been exceeded.

The only type of constraint that needs to be treated under the conditions stated above is the presence of edges representing associations mandatory for a given class. An association can be considered mandatory for a given class if the lower bound of multiplicity at the related association end is higher than zero. In such a case, at least one edge connected to a vertex with a label representing a given class must be present.

The pseudocode in Listing 9 iterates through all binary associations that are present in the input OntoUML diagram. If the lower bound of multiplicity at any of the two association ends is greater than zero, the associated class at that association end is considered mandatory, and the second related class on the opposite association end must be in association with it. In this case, the pseudocode adds the `RELATIONSHIP_MUST_BE_PRESENT` integrity constraint to ensure that there is at least one edge for every vertex with a label corresponding to the related class that leads to a vertex with a label corresponding to the mandatory class.

```
1  //INPUT: valid OntoUML diagram; OUTPUT: list of constraints
2  foreach (BinaryAssociation a in diagram.Associations)
3  {
4      if(a.FirstMultiplicity.LowerBound > 0)
5          constraints.Add(RELATIONSHIP_MUST_BE_PRESENT,
6              a.SecondClass.Name //relatedLabelName
7              a.FirstClass.Name, //mandatoryLabelName
8          );
9      if (a.SecondMultiplicity.LowerBound > 0)
10         constraints.Add(RELATIONSHIP_MUST_BE_PRESENT,
11             a.FirstClass.Name //relatedLabelName
12             a.SecondClass.Name, //mandatoryLabelName
13         );
14 }
```

Listing 9: Pseudocode adding ICs to ensure presence of associations

5.3 Implementation of integrity constraints in Neo4j

The previous section described in detail how an OntoUML model represented as a class diagram can be systematically transformed into a list of specific integrity constraints. In order to create an instantiation of the OntoUML model in the target database technology, it is necessary to further describe the way in which the particular integrity constraints should be implemented using the Neo4j integrity constraint mechanisms.

For implementing proposed integrity constraints in the Neo4j database system, it is primarily required to use database triggers in most cases, as the additional logic resulting from most of the proposed integrity constraints is more complex than the logic provided by basic Neo4j built-in constraints. This section presents how to implement all necessary constraints and database triggers in Neo4j using the Cypher language and the APOC library.

5.3.1 Constraint to check if a property is unique

The first integrity constraint used in the transformation of the OntoUML model is denoted as `PROPERTY_MUST_BE_UNIQUE`. It is the only integrity constraint that can be directly implemented by the Neo4j built-in constraint mechanism. The template of a constraint creation statement in Listing 10 creates a new constraint that ensures a unique value of a property with specified `propertyName` for a node having a provided `labelName`.

```
1 //CONSTRAINT: PROPERTY_MUST_BE_UNIQUE
2 //INPUT: labelName, propertyName
3 CREATE CONSTRAINT {labelName}_{propertyName}_must_be_unique IF NOT EXISTS
4 FOR (node:{labelName}) REQUIRE node.{propertyName} IS UNIQUE
```

Listing 10: Template of constraint to check if a property is unique

5.3.2 Trigger to check a data type of a property

The second integrity constraint used in the transformation of the OntoUML model is denoted as `PROPERTY_MUST_BE_OF_DATATAYPE` and must be implemented using a database trigger. The template of trigger query in Listing 11 first obtains all nodes having a label with specified `labelName` that have been assigned a new label or a new property. Then, in the case specified `propertyName` exists on any of obtained nodes, it validates if the value of this property corresponds to a provided `dataTypeName`.

```
1  //CONSTRAINT: PROPERTY_MUST_BE_OF_DATATAYPE
2  //INPUT: labelName, propertyName, dataTypeName
3  UNWIND (
4      apoc.trigger.nodesByLabel($assignedLabels, "{labelName}") +
5      apoc.trigger.nodesByLabel($assignedNodeProperties, "{labelName}")
6  ) AS node
7  CALL apoc.util.validate(
8      exists(node.{propertyName}) and not
9      apoc.meta.cypher.isType(node.{propertyName}, "{dataTypeName}"),
10     "{propertyName} property of {labelName} must be of datatype
   ↪  {dataTypeName}",
11     null
12 )
13 RETURN null
```

Listing 11: Template of trigger to check a data type of a property

5.3.3 Trigger to check if a property is present

The third integrity constraint used in the transformation of the OntoUML model is denoted as `PROPERTY_MUST_BE_PRESENT`. In the case of Neo4j Enterprise Edition, it would be possible to implement this integrity constraint using a built-in constraint mechanism. However, in order to maintain generality for all Neo4j editions, this integrity constraint is also implemented using a database trigger. The template of trigger query in Listing 12 first obtains all newly created nodes or nodes that have been assigned a new label or property. Then, for all obtained nodes that have a label with specified `labelName`, it validates if the property with provided `propertyName` is present.

```
1  //CONSTRAINT: PROPERTY_MUST_BE_PRESENT
2  //INPUT: labelName, propertyName
3  UNWIND (
4      $createdNodes +
5      apoc.trigger.nodesByLabel($assignedLabels, "{labelName}") +
6      apoc.trigger.nodesByLabel($removedNodeProperties, "{labelName}")
7  ) AS node
8  CALL apoc.util.validate(
9      apoc.label.exists(node, "{labelName}") and not
10     exists(node.{propertyName}),
11     "{propertyName} property of {labelName} must be present",
12     null
13 )
14 RETURN null
```

Listing 12: Template of trigger to check if a property is present

5.3.4 Trigger to check if a label is in a combination

The fourth integrity constraint used in the OntoUML model transformation is denoted as `LABEL_MUST_BE_IN_COMBINATION` and must be implemented using a database trigger. For the implementation of this constraint, it is impossible to create a template of a database trigger based only on simple text substitution, but it is necessary to preprocess the inputs into the required form.

Based on the input list of `otherLabelNames`, the pseudocode in Listing 13 determines three specific text strings that become the parts of the following trigger template. The first text string called `nodesByLabelString` is used to select all nodes on which any of the labels from the `otherLabelNames` list have been removed. It is created by wrapping label names from the `otherLabelNames` list with the `nodesByLabel` function and then concatenating these functions into a single string. The second text string called

`labelExistsString` is used to validate whether a node has assigned any of the labels from the `otherLabelNames` list. It is created by wrapping label names from the `otherLabelNames` list with the `exists` function and then concatenating these functions into a single string. The last text string called `otherNamesOrString` is used in a trigger validation message and is created by concatenating label names from the `otherLabelNames` list.

```
//INPUT: otherLabelNames
//OUTPUT: nodesByLabelString, labelExistsString, otherNamesOrString
nodesByLabelString = otherLabelNames
    .Map(el => "apoc.trigger.nodesByLabel($removedLabels, \"{el}\")")
    .Reduce((acc, el) => "{acc} + {el}");
labelExistsString = otherLabelNames
    .Map(el => "apoc.label.exists(node, \"{el}\")")
    .Reduce((acc, el) => "{acc} or {el}");
otherNamesOrString = otherLabelNames
    .Reduce((acc, el) => "{acc} or {el}");
```

Listing 13: Pseudocode calculating prerequisite inputs for trigger templates

Using the preprocessed inputs, the template of trigger query in Listing 14 first obtains all nodes to which a new label having the specified `labelName` has been assigned or on which any label from the `otherLabelNames` list has been recently removed. Then, for all obtained nodes having a label with the specified `labelName`, it validates if any label with a name from the `otherLabelNames` list is simultaneously present.

```
//CONSTRAINT: LABEL_MUST_BE_IN_COMBINATION
//INPUT: labelName, nodesByLabelString, labelExistsString, otherNamesOrString
UNWIND (
    apoc.trigger.nodesByLabel($assignedLabels, "{labelName}") +
    {nodesByLabelString}
) AS node
CALL apoc.util.validate(
    apoc.label.exists(node, "{labelName}") and not
    ({labelExistsString}),
    "{labelName} label must be in a combination with {otherNamesOrString}
↪   labels",
    null
)
RETURN null
```

Listing 14: Template of trigger to check if a label is in a combination

5.3.5 Trigger to check if a label is not in a combination

The fifth integrity constraint used in the transformation of the OntoUML model is denoted as `LABEL_CANNOT_BE_IN_COMBINATION`. This integrity constraint must be implemented using a database trigger, and as in the previous case, the provided inputs must be preprocessed into the form of concatenated text strings. Only this way it is possible to create the appropriate trigger template. Also, in this situation, it is possible to use the already described pseudocode in Listing 13 for the input preprocessing.

The template of trigger query in Listing 15 first obtains all nodes to which a new label with the specified `labelName` has been assigned. Then, for all obtained nodes, the trigger query validates if there is not present any label with a name from the `otherLabelNames` list.

```
1  //CONSTRAINT: LABEL_CANNOT_BE_IN_COMBINATION
2  //INPUT: labelName, labelExistsString, otherNamesOrString
3  UNWIND (
4      apoc.trigger.nodesByLabel($assignedLabels, "{labelName}")
5  ) AS node
6  CALL apoc.util.validate(
7      {labelExistsString},
8      "{labelName} label cannot be in a combination with {otherNamesOrString}
   ↪  labels",
9      null
10 )
11 RETURN null
```

Listing 15: Template of trigger to check if a label is not in a combination

5.3.6 Triggers to check if a relationship is present

The last integrity constraint used in the transformation of the OntoUML model is denoted as `RELATIONSHIP_MUST_BE_PRESENT`. In the case of this integrity constraint, there is no need to preprocess the inputs, but for reasons given by the internal implementation of the database trigger mechanism in the Neo4j database system, this constraint needs to be implemented using three independent database triggers. All three database triggers validate if the node assigned the specified `relatedLabelName` has a relationship with the node assigned the provided `mandatoryLabelName`, but each of the triggers is tied to a different type of data modification event.

The first trigger is tied to the event of label assignment. The template of trigger query in Listing 16 first obtains all nodes to which a new label with the `relatedLabelName` has been assigned. Then, for all obtained nodes, it val-

idates if the number of related nodes assigned with the `mandatoryLabelName` is greater than zero. In that case, it is ensured that at least one relationship leads from the related node to the mandatory node.

```
1  //CONSTRAINT: RELATIONSHIP_MUST_BE_PRESENT (label assignment event)
2  //INPUT: relatedLabelName, mandatoryLabelName
3  UNWIND (
4      apoc.trigger.nodesByLabel($assignedLabels, "{relatedLabelName}")
5  ) AS node1
6  CALL apoc.util.validate(
7      SIZE([(node1)--(n2:{mandatoryLabelName}) | n2]) < 1,
8      "{relatedLabelName} must be in relationship with {mandatoryLabelName}",
9      null
10 )
11 RETURN null
```

Listing 16: Template of trigger to check if a relationship is present (assign)

The second trigger is tied to the event of label removal. The template of trigger query in Listing 17 first obtains all nodes on which a label with the `mandatoryLabelName` has been recently removed. Then, it obtains all nodes that were related to the previously obtained nodes and are assigned with `relatedLabelName`. Finally, it validates if the number of related nodes that are still assigned with `mandatoryLabelName` is greater than zero. This ensures that at least one relationship leads from the related node to the mandatory node, although the label on some of the mandatory nodes has been removed.

```
1  //CONSTRAINT: RELATIONSHIP_MUST_BE_PRESENT (label removal event)
2  //INPUT: relatedLabelName, mandatoryLabelName
3  UNWIND (
4      apoc.trigger.nodesByLabel($removedLabels, "{mandatoryLabelName}")
5  ) AS node2
6  UNWIND (
7      [(n1:{relatedLabelName})--(node2) | n1]
8  ) AS node1
9  CALL apoc.util.validate(
10     SIZE([(node1)--(n2:{mandatoryLabelName}) | n2]) < 1,
11     "{relatedLabelName} must be in relationship with {mandatoryLabelName}",
12     null
13 )
14 RETURN null
```

Listing 17: Template of trigger to check if a relationship is present (remove)

The last database trigger is tied to the event of relationship deletion. The template of trigger query in Listing 18 first obtains all recently deleted relationships. Then, it obtains an aggregated list of all start and end nodes of recently deleted relationships. Finally, for all obtained nodes assigned with specified `relatedLabelName`, the trigger query validates if the number of related nodes assigned with the provided `mandatoryLabelName` is greater than zero. It ensures that at least one relationship leads from the related node to the mandatory node, although one of the relationships has been deleted.

```
1  //CONSTRAINT: RELATIONSHIP_MUST_BE_PRESENT (relationship deletion event)
2  //INPUT: relatedLabelName, mandatoryLabelName
3  UNWIND (
4      $deletedRelationships
5  ) AS rel
6  UNWIND (
7      [apoc.rel.startNode(rel), apoc.rel.endNode(rel)]
8  ) AS node1
9  CALL apoc.util.validate(
10     apoc.label.exists(node1, "{relatedLabelName}") and
11     (SIZE([(node1)--(n2:{mandatoryLabelName}) | n2]) < 1),
12     "{relatedLabelName} must be in relationship with {mandatoryLabelName}",
13     null
14 )
15 RETURN null
```

Listing 18: Template of trigger to check if a relationship is present (delete)

Chapter 6

Instantiation of example model

Using the comprehensive procedure for the transformation of the OntoUML diagrams designed in the previous chapter, the example OntoUML model can now be instantiated in the Neo4j database system. Since it is necessary to use a considerable number of integrity constraints for correct instantiation and preserving all the semantics, manual transformation according to the described procedure would be a very complex task with a high risk of errors. For that reason, an automated transformation tool written in C# programming language has been created as a part of this thesis.

This chapter first introduces this automated transformation tool and then describes the instantiation of the example model in several subsequent steps, in which additional constructs and classes with additional stereotypes are gradually added. Since the complete Cypher code of all Neo4j integrity constraints would be too long, only names and input arguments of necessary integrity constraints are provided in this chapter. The complete Cypher code of the instantiated model, having almost a thousand lines, can be found as the content of the enclosed CD.

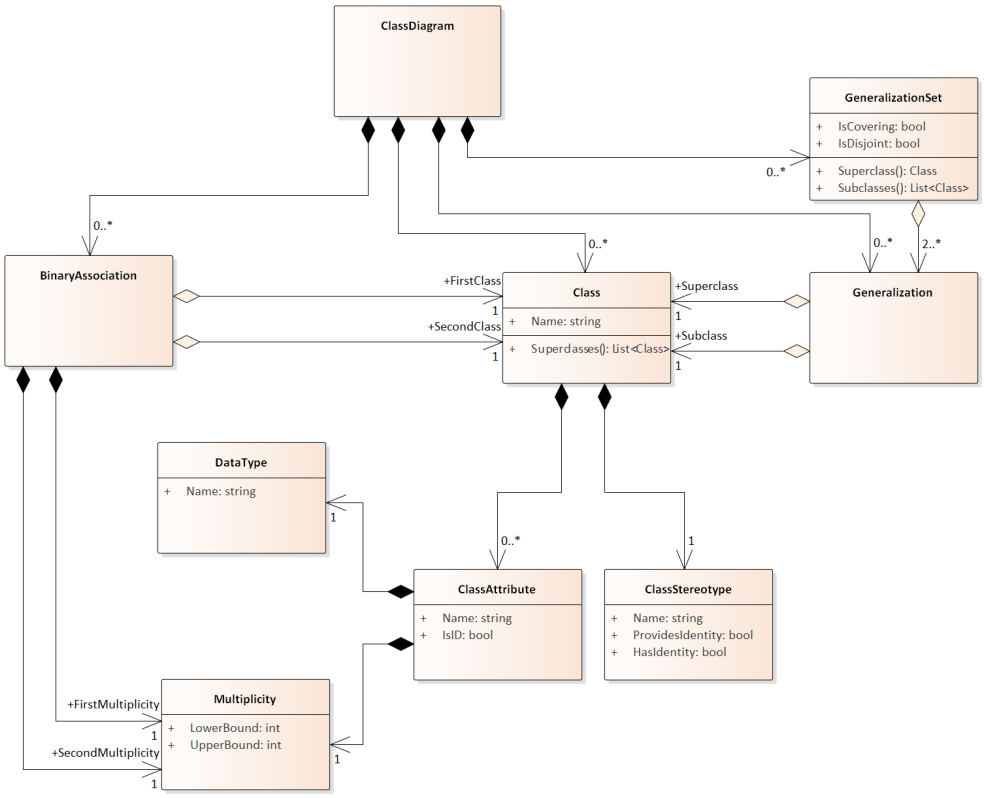
6.1 Automated transformation tool

The automated tool for the transformation of the OntoUML models, which has been created based on the designed transformation procedure, is called *DiagramInstantiator*. It is a CLI application built on the .NET 6.0 cross-platform framework in C# programming language. The source code of this application can also be found as the content of the enclosed CD.

The input of this application can be any syntactically and semantically valid OntoUML diagram. For the purposes of this thesis, the input diagram is represented directly by code in C# language. However, the application can be easily modified in the future to allow the input diagram representation to be loaded from standard formats such as XML or JSON, whereas these formats can be exported, for example, from OntoUML modeling tools.

In order to be able to represent the input OntoUML diagram with C# code and subsequently work with it using the object-oriented paradigm (OOP), the application contains its own OntoUML diagram metamodel adapted to the needs of this thesis. Classes of this metamodel define all the essential elements of the UML class diagram, as they are listed and described in section 5.1. A complete description of the OntoUML diagram metamodel, which is itself described using a UML class diagram, can be found in Figure 6.1.

Figure 6.1: OntoUML diagram metamodel – class diagram

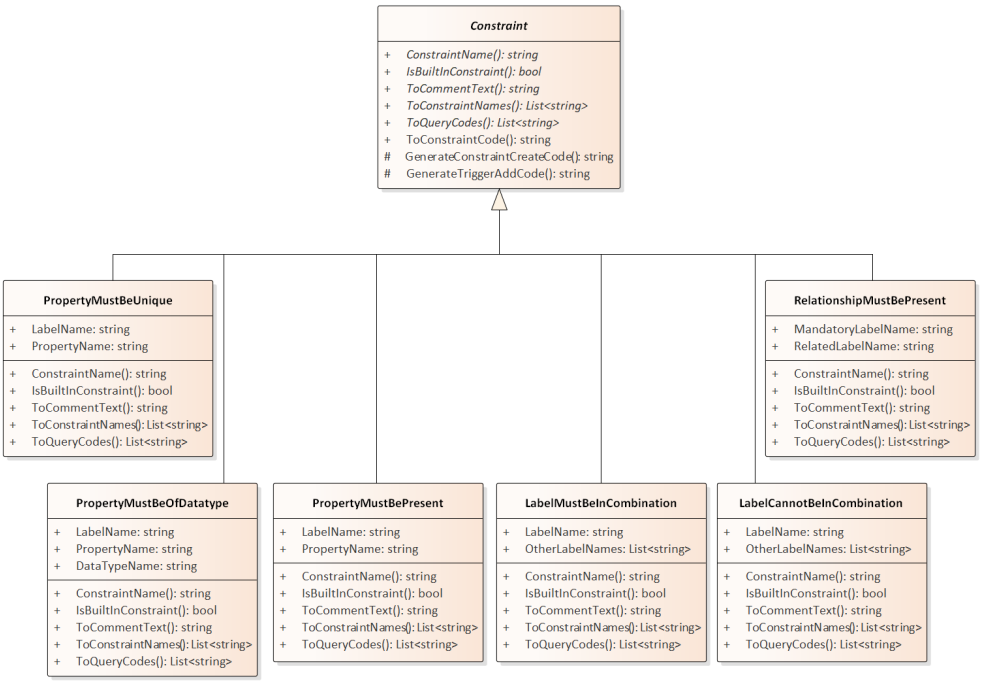


The object representation of an input OntoUML diagram is according to the procedure described in section 5.2 transformed by the created application into the set of Neo4j integrity constraints. In order to work effectively with them, each type of Neo4j integrity constraint described in section 5.3 also has its own object representation.

Every concrete class which can be found in Figure 6.2 encapsulates a template of Neo4j constraint or a template of Neo4j database trigger together with all of the necessary input parameters. Since all of these classes share some capabilities, there is an abstract `Constraint` class from which concrete classes representing integrity constraints inherit these capabilities. In addition, this abstract class provides a uniform interface for working with all considered types of integrity constraints.

Each object created based on these classes represents one specific integrity constraint and has the ability to generate the output in the form of corresponding Neo4j Cypher code. The generated output code can either contain only comments with the names and input arguments of individual integrity constraints for documentation purposes or can contain a full-fledged Cypher code, including constraint creation statements and database triggers.

Figure 6.2: Neo4j constraint model – class diagram



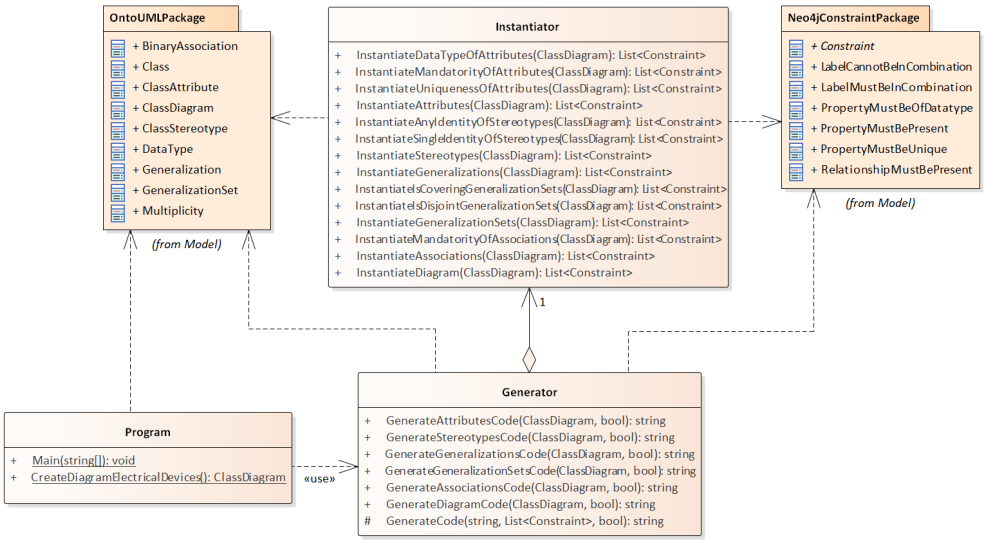
In addition to the OntoUML diagram metamodel and to the classes representing all the individual Neo4j integrity constraints, the application itself primarily consists of three main classes ensuring its functionality. These classes and their dependencies on the other parts of the application are described by the design model, which can be found in Figure 6.3.

The first class, called `Instantiator`, represents a type of object that is able to transform between object representations of the OntoUML diagram and Neo4j integrity constraints. For each OntoUML construct, this class contains a method whose code matches the pseudocode described in section 5.2 and which provides a list of corresponding constraints.

The second class, called `Generator`, represents a type of object that is able to generate Cypher code based on elements of a given OntoUML diagram. For this purpose, it uses the capabilities of the `Instantiator` class, which it contains, and subsequently, the abilities of objects representing specific integrity constraints. Each method of this class has a parameter that decides whether only comments or full-fledged Cypher code should be generated.

The last class, called `Program`, represents an entry point of the entire application. This class first creates an object representation of the OntoUML diagram based on the provided input C# code. The input code corresponding to the example OntoUML model can be found in Appendix C. The `Program` class then uses the `Generator` class to generate the corresponding output code and writes it to the output text file. The entire output code (comments only) generated based on the example OntoUML model can be found in Appendix D. Its parts are separately discussed below in the following sections.

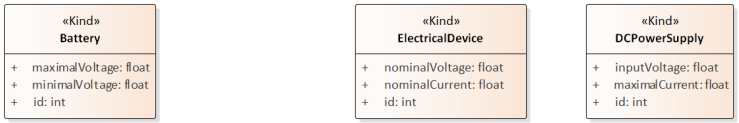
Figure 6.3: DiagramInstantiator design model – class diagram



6.2 Kinds and Properties

In the first step of the example OntoUML model instantiation, all classes with a Kind stereotype are transformed together with all their attributes. There are three different Kinds in the example model – `Battery`, `ElectricalDevice`, and `DCPowerSupply`. The state of the instantiation after the first step is specified by the OntoUML diagram, which can be found below in Figure 6.4.

Figure 6.4: Example OntoUML model (Kinds) – class diagram



Names and input arguments of integrity constraints corresponding to the first instantiation step are listed in Listing 19. In the first place, the set of PROPERTY_MUST_BE_OF_DATATYPE ICs ensures the correct data type for every attribute of all three transformed classes. Furthermore, since all these attributes can be considered mandatory according to their default multiplicity, there also must be the PROPERTY_MUST_BE_PRESENT IC for each of them. Also, in the case of classes with the Kind stereotype, there are ID attributes, whose uniqueness must be ensured by the PROPERTY_MUST_BE_UNIQUE IC. Finally, the last set of LABEL_CANNOT_BE_IN_COMBINATION ICs ensures that nothing can be an instance of two different Kinds and have two identities.

```
1  // ----- ATTRIBUTES
2  PROPERTY_MUST_BE_OF_DATATYPE(ElectricalDevice, id, INTEGER)
3  PROPERTY_MUST_BE_OF_DATATYPE(ElectricalDevice, nominalCurrent, FLOAT)
4  PROPERTY_MUST_BE_OF_DATATYPE(ElectricalDevice, nominalVoltage, FLOAT)
5  PROPERTY_MUST_BE_OF_DATATYPE(DCPowerSupply, id, INTEGER)
6  PROPERTY_MUST_BE_OF_DATATYPE(DCPowerSupply, inputVoltage, FLOAT)
7  PROPERTY_MUST_BE_OF_DATATYPE(DCPowerSupply, maximalCurrent, FLOAT)
8  PROPERTY_MUST_BE_OF_DATATYPE(Battery, id, INTEGER)
9  PROPERTY_MUST_BE_OF_DATATYPE(Battery, maximalVoltage, FLOAT)
10 PROPERTY_MUST_BE_OF_DATATYPE(Battery, minimalVoltage, FLOAT)
11 PROPERTY_MUST_BE_PRESENT(ElectricalDevice, id)
12 PROPERTY_MUST_BE_PRESENT(ElectricalDevice, nominalCurrent)
13 PROPERTY_MUST_BE_PRESENT(ElectricalDevice, nominalVoltage)
14 PROPERTY_MUST_BE_PRESENT(DCPowerSupply, id)
15 PROPERTY_MUST_BE_PRESENT(DCPowerSupply, inputVoltage)
16 PROPERTY_MUST_BE_PRESENT(DCPowerSupply, maximalCurrent)
17 PROPERTY_MUST_BE_PRESENT(Battery, id)
18 PROPERTY_MUST_BE_PRESENT(Battery, maximalVoltage)
19 PROPERTY_MUST_BE_PRESENT(Battery, minimalVoltage)
20 PROPERTY_MUST_BE_UNIQUE(ElectricalDevice, id)
21 PROPERTY_MUST_BE_UNIQUE(DCPowerSupply, id)
22 PROPERTY_MUST_BE_UNIQUE(Battery, id)
23 // ----- STEREOTYPES
24 LABEL_CANNOT_BE_IN_COMBINATION(ElectricalDevice, {DCPowerSupply, Battery,
   ↪ ElectricalConnection})
25 LABEL_CANNOT_BE_IN_COMBINATION(DCPowerSupply, {ElectricalDevice, Battery,
   ↪ ElectricalConnection})
26 LABEL_CANNOT_BE_IN_COMBINATION(Battery, {ElectricalDevice, DCPowerSupply,
   ↪ ElectricalConnection})
27 LABEL_CANNOT_BE_IN_COMBINATION(ElectricalConnection, {ElectricalDevice,
   ↪ DCPowerSupply, Battery})
```

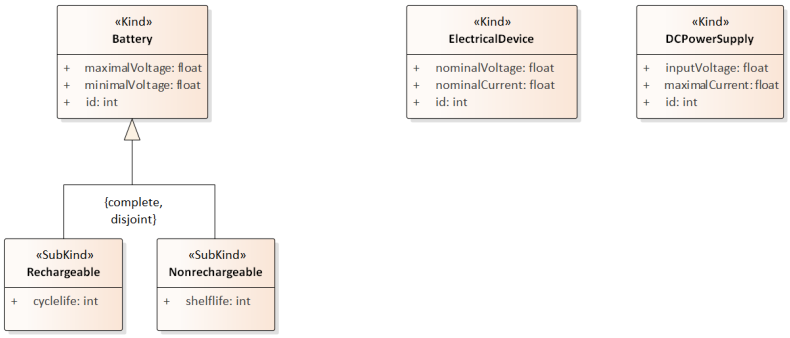
Listing 19: Constraints of instantiated example model – Kinds

It is important to mention that at this point, it is already necessary to consider the existence of the `ElectricalConnection` class, which is otherwise transformed only in the fifth instantiation step. Like the other classes with the Kind stereotype, this class with the Relator stereotype provides its own identity and, therefore, must also be included in the set of `LABEL_CANNOT_BE_IN_COMBINATION` ICs.

6.3 Subkinds

In the second step of the example model instantiation, all classes with a Subkind stereotype are transformed. There exist two different Subkinds in the example model – `Rechargeable` and `Nonrechargeable`, both are subclasses of the `Battery` class. The cumulative state after the second instantiation step is specified by the OntoUML diagram, which can be found below in Figure 6.5.

Figure 6.5: Example OntoUML model (Subkinds) – class diagram



Names and input arguments of integrity constraints corresponding to the second instantiation step are listed in Listing 20. As in the previous instantiation step, there are the `PROPERTY_MUST_BE_OF_DATATYPE` and the `PROPERTY_MUST_BE_PRESENT` ICs representing the attributes of newly added classes, their data types and mandatority. Moreover, the first two `LABEL_MUST_BE_IN_COMBINATION` ICs represent the generalizations between the `Battery` class and both Subkind classes. Since both generalizations are combined into the generalization set, ICs that ensure its properties also have to be present. The last `LABEL_MUST_BE_IN_COMBINATION` IC represents the IsCovering property and ensures that any instance of the `Battery` class is also an instance of either the `Rechargeable` class or the `Nonrechargeable` class. Finally, the last two `LABEL_CANNOT_BE_IN_COMBINATION` ICs represent the IsDisjoint property and ensure that any instance of the `Rechargeable` class cannot be an instance of the `Nonrechargeable` class and vice versa.

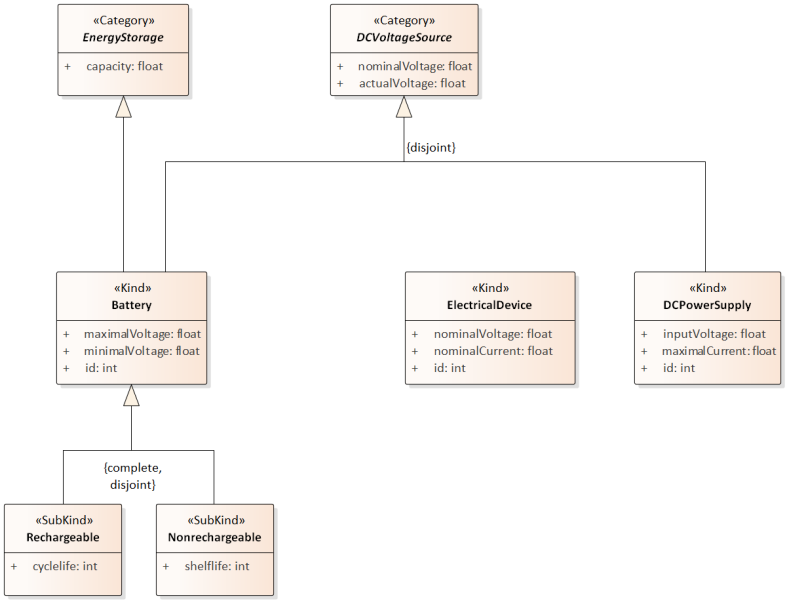
```
// ----- ATTRIBUTES
PROPERTY_MUST_BE_OF_DATATYPE(Rechargeable, cyclelife, INTEGER)
PROPERTY_MUST_BE_OF_DATATYPE(Nonrechargeable, shelflife, INTEGER)
PROPERTY_MUST_BE_PRESENT(Rechargeable, cyclelife)
PROPERTY_MUST_BE_PRESENT(Nonrechargeable, shelflife)
// ----- GENERALIZATIONS
LABEL_MUST_BE_IN_COMBINATION(Rechargeable, {Battery})
LABEL_MUST_BE_IN_COMBINATION(Nonrechargeable, {Battery})
// ----- GENERALIZATION SETS
LABEL_MUST_BE_IN_COMBINATION(Battery, {Rechargeable, Nonrechargeable})
LABEL_CANNOT_BE_IN_COMBINATION(Rechargeable, {Nonrechargeable})
LABEL_CANNOT_BE_IN_COMBINATION(Nonrechargeable, {Rechargeable})
```

Listing 20: Constraints of instantiated example model – Subkinds

6.4 Categories

In the third step of the example model instantiation, all abstract classes with a Category stereotype are transformed. Two different Categories exist in the example model – `EnergySource` and `DCPowerSource`, representing superclasses of previously transformed Kinds. The cumulative state after the third instantiation step is specified by the OntoUML diagram below in Figure 6.6.

Figure 6.6: Example OntoUML model (Categories) – class diagram



Names and input arguments of integrity constraints corresponding to the third instantiation step are listed in Listing 21. As can be seen in this listing, attributes of the newly added classes are transformed in the same way as in the previous instantiation steps. The main difference from the previous instantiation steps is that classes with the Category stereotype are NonSortals, and thus it is necessary to ensure their instances always have an identity. For this reason, the first two `LABEL_MUST_BE_IN_COMBINATION` ICs ensure that instances of the `EnergySource` class or the `DCPowerSource` class are also instances of their direct Sortal subclasses, and obtain an identity from them. Besides that, the last three `LABEL_MUST_BE_IN_COMBINATION` ICs represent generalizations between the superclasses with Category stereotypes and all their subclasses. In the case of the `DCPowerSource` superclass, its generalizations are likewise combined into a generalization set, but this time only with the IsDisjoint property. Therefore, the last two `LABEL_CANNOT_BE_IN_COMBINATION` ICs ensure that all instances of the `Battery` class and all instances of the `DCPowerSupply` class are entirely disjoint.

```
1  // ----- ATTRIBUTES
2  PROPERTY_MUST_BE_OF_DATATYPE(EnergyStorage, capacity, FLOAT)
3  PROPERTY_MUST_BE_OF_DATATYPE(DCVoltageSource, actualVoltage, FLOAT)
4  PROPERTY_MUST_BE_OF_DATATYPE(DCVoltageSource, nominalVoltage, FLOAT)
5  PROPERTY_MUST_BE_PRESENT(EnergyStorage, capacity)
6  PROPERTY_MUST_BE_PRESENT(DCVoltageSource, actualVoltage)
7  PROPERTY_MUST_BE_PRESENT(DCVoltageSource, nominalVoltage)
8  // ----- STEREOTYPES
9  LABEL_MUST_BE_IN_COMBINATION(EnergyStorage, {Battery})
10 LABEL_MUST_BE_IN_COMBINATION(DCVoltageSource, {DCPowerSupply, Battery})
11 // ----- GENERALIZATIONS
12 LABEL_MUST_BE_IN_COMBINATION(Battery, {EnergyStorage})
13 LABEL_MUST_BE_IN_COMBINATION(DCPowerSupply, {DCVoltageSource})
14 LABEL_MUST_BE_IN_COMBINATION(Battery, {DCVoltageSource})
15 // ----- GENERALIZATION SETS
16 LABEL_CANNOT_BE_IN_COMBINATION(DCPowerSupply, {Battery})
17 LABEL_CANNOT_BE_IN_COMBINATION(Battery, {DCPowerSupply})
```

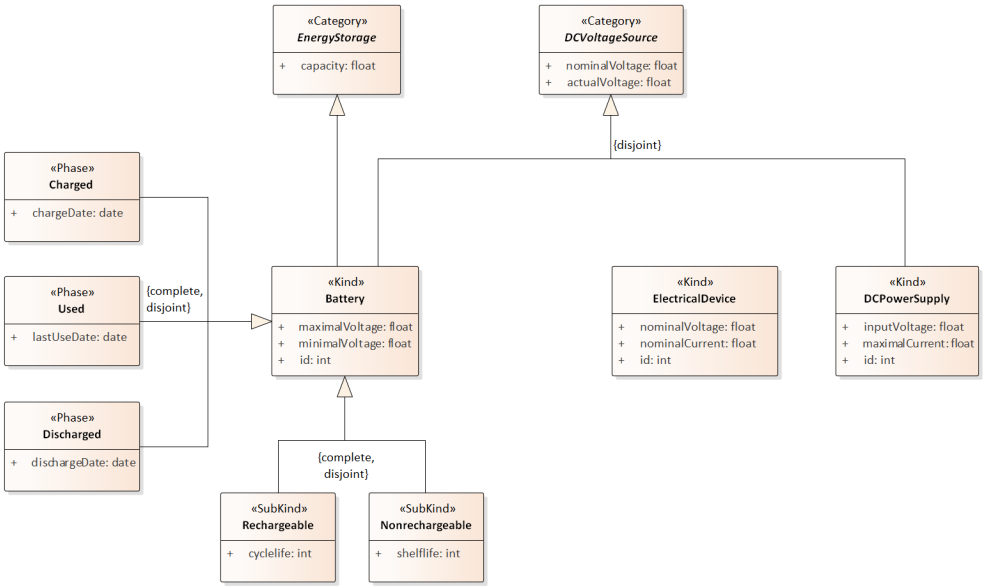
Listing 21: Constraints of instantiated example model – Categories

6.5 Phases

In the fourth step of the example model instantiation, all classes with a Phase stereotype are transformed. In the example model, the already transformed `Battery` class with the Kind stereotype can be in one of three states repre-

sented by the following three Phases – `Charged`, `Used`, or `Discharged`. The cumulative state after the fourth instantiation step is specified by the OntoUML diagram below in Figure 6.7.

Figure 6.7: Example OntoUML model (Phases) – class diagram



Names and input arguments of integrity constraints corresponding to the fourth instantiation step are listed in Listing 22. The transformation of classes with the Phase stereotype can be considered almost identical to the transformation of classes with the Subkind stereotype. It is given by the fact that both stereotypes denote Sortals classes that do not provide identity and differ primarily in their rigidity. However, since there is no point in treating rigidity within the proposed transformation, as discussed and explained in chapter 8, the only difference is that classes with the Phase stereotype must always be found in the generalization set with both IsCovering and IsDisjoint properties. Regarding the listed integrity constraints, the `PROPERTY_MUST_BE_OF_DATATYPE` ICs and the `PROPERTY_MUST_BE_PRESENT` ICs again represent attributes of newly transformed classes. Likewise, the first three `LABEL_MUST_BE_IN_COMBINATION` ICs represent generalizations between the `Battery` superclass and all three subclasses with Phase stereotype. As mentioned above, even in this case, the generalizations are combined into a generalization set with both IsCovering and IsDisjoint properties. Therefore the last `LABEL_MUST_BE_IN_COMBINATION` IC ensures that instances of the `Battery` superclass are always instances of one of the Phase subclass and the last three `LABEL_CANNOT_BE_IN_COMBINATION` ICs ensure that all instances of Phase subclasses are mutually disjoint.

```
1  // ----- ATTRIBUTES
2  PROPERTY_MUST_BE_OF_DATATYPE(Charged, chargeDate, DATE)
3  PROPERTY_MUST_BE_OF_DATATYPE(Used, lastUseDate, DATE)
4  PROPERTY_MUST_BE_OF_DATATYPE(Discharged, dischargeDate, DATE)
5  PROPERTY_MUST_BE_PRESENT(Charged, chargeDate)
6  PROPERTY_MUST_BE_PRESENT(Used, lastUseDate)
7  PROPERTY_MUST_BE_PRESENT(Discharged, dischargeDate)
8  // ----- GENERALIZATIONS
9  LABEL_MUST_BE_IN_COMBINATION(Charged, {Battery})
10 LABEL_MUST_BE_IN_COMBINATION(Used, {Battery})
11 LABEL_MUST_BE_IN_COMBINATION(Discharged, {Battery})
12 // ----- GENERALIZATION SETS
13 LABEL_MUST_BE_IN_COMBINATION(Battery, {Charged, Used, Discharged})
14 LABEL_CANNOT_BE_IN_COMBINATION(Charged, {Used, Discharged})
15 LABEL_CANNOT_BE_IN_COMBINATION(Used, {Charged, Discharged})
16 LABEL_CANNOT_BE_IN_COMBINATION(Discharged, {Charged, Used})
```

Listing 22: Constraints of instantiated example model – Phases

6.6 Roles, RoleMixin and Relator

In the last step of the example model instantiation, all classes with Role and RoleMixin stereotypes are transformed. In the example model, there exist three different Roles – the `ConnectedElectricalDevice` as a Role of the `ElectricalDevice` Kind, the `ConnectedBattery` as a Role of the `Battery` Kind, and the `ConnectedDCPowerSupply` as a Role of the `DCPowerSupply` Kind. Although the last two Roles have different identity providers, they still share a common attribute and a common association, which they inherit from `ConnectedDCPowerSource` RoleMixin. Since the mandatory association of RoleMixin must be implemented using the class with the Relator stereotype, the `ElectricalConnection` Relator is as well transformed in this instantiation step. The final state of instantiation after the last step is specified by the OntoUML diagram in Figure 6.7, which can be found in chapter 3.

Names and input arguments of integrity constraints corresponding to the last instantiation step are listed in Listing 23. This instantiation step can be considered the most complex of all, as it includes all types of designed integrity constraints. First, in addition to the already standard transformation of attributes using the `PROPERTY_MUST_BE_OF_DATATYPE` ICs and the `PROPERTY_MUST_BE_PRESENT` ICs, there is also the `PROPERTY_MUST_BE_UNIQUE` IC, as the `ElectricalConnection` class with Relator stereotype is another identity provider, and its ID attribute must be unique. Second, regarding the transformation of the semantics of class stereotypes, integrity constraints ensuring that instances of the `ElectricalConnection` class are not simulta-

neously instances of another identity provider have already been added in the first instantiation step. Besides that, the first `LABEL_MUST_BE_IN_COMBINATION` IC ensures that instances of the `ConnectedDCVoltageSource` NonSortal class get their identity from one of its Sortal subclasses. Third, the generalizations between Roles and Kinds and between Roles and RoleMixin are represented by another five `LABEL_MUST_BE_IN_COMBINATION` ICs. Since some generalizations here are once more combined into a generalization set, the last two `LABEL_CANNOT_BE_IN_COMBINATION` ICs represent its IsDisjoint property. Finally, the last four `RELATIONSHIP_MUST_BE_PRESENT` ICs represent the mandatory binary associations between `ConnectedElectricalDevice`, `ElectricalConnection`, and `ConnectedDCVoltageSource` classes.

```
1  // ----- ATTRIBUTES
2  PROPERTY_MUST_BE_OF_DATATYPE(ElectricalConnection, id, INTEGER)
3  PROPERTY_MUST_BE_OF_DATATYPE(ElectricalConnection, resistance, FLOAT)
4  PROPERTY_MUST_BE_OF_DATATYPE(ConnectedDCVoltageSource, drawnCurrent, FLOAT)
5  PROPERTY_MUST_BE_PRESENT(ElectricalConnection, id)
6  PROPERTY_MUST_BE_PRESENT(ElectricalConnection, resistance)
7  PROPERTY_MUST_BE_PRESENT(ConnectedDCVoltageSource, drawnCurrent)
8  PROPERTY_MUST_BE_UNIQUE(ElectricalConnection, id)
9  // ----- STEREOTYPES
10 LABEL_MUST_BE_IN_COMBINATION(ConnectedDCVoltageSource, {ConnectedBattery,
   ↪  ConnectedDCPowerSupply})
11 // ----- GENERALIZATIONS
12 LABEL_MUST_BE_IN_COMBINATION(ConnectedElectricalDevice, {ElectricalDevice})
13 LABEL_MUST_BE_IN_COMBINATION(ConnectedDCPowerSupply,
   ↪  {ConnectedDCVoltageSource})
14 LABEL_MUST_BE_IN_COMBINATION(ConnectedBattery, {ConnectedDCVoltageSource})
15 LABEL_MUST_BE_IN_COMBINATION(ConnectedDCPowerSupply, {DCPowerSupply})
16 LABEL_MUST_BE_IN_COMBINATION(ConnectedBattery, {Battery})
17 // ----- GENERALIZATION SETS
18 LABEL_CANNOT_BE_IN_COMBINATION(ConnectedDCPowerSupply, {ConnectedBattery})
19 LABEL_CANNOT_BE_IN_COMBINATION(ConnectedBattery, {ConnectedDCPowerSupply})
20 // ----- ASSOCIATIONS
21 RELATIONSHIP_MUST_BE_PRESENT(ConnectedElectricalDevice, ElectricalConnection)
22 RELATIONSHIP_MUST_BE_PRESENT(ElectricalConnection, ConnectedElectricalDevice)
23 RELATIONSHIP_MUST_BE_PRESENT(ElectricalConnection, ConnectedDCVoltageSource)
24 RELATIONSHIP_MUST_BE_PRESENT(ConnectedDCVoltageSource, ElectricalConnection)
```

Listing 23: Constraints of instantiated example model – Roles, RoleMixin

Chapter 7

Testing of instantiated model

As the last step of the conducted case study, it is necessary to verify that the proposed procedure for the transformation of OntoUML diagrams in the Neo4j database system can be considered correct. Since the aim of this thesis is not to provide complete formal proof of the correctness of the proposed procedure, the verification is based on a series of test cases testing different aspects of the instantiated example OntoUML model.

Testing is divided into two subsequent phases. In the first phase of positive testing, it is verified whether the instantiated model behaves as expected when using only valid testing data. In the second phase of negative testing, a series of test data manipulation Cypher queries that violate the semantics given by the OntoUML model is gradually executed. Then, it is verified if their execution is prevented by the relevant integrity constraints. In addition, the last section of this chapter also describes how this process can be automated so that the testing can be performed flawlessly whenever the transformation procedure is changed or extended.

7.1 Positive testing

Positive testing aims to determine whether an instantiated example model can be successfully deployed on the actual Neo4j database system and whether it is possible to load valid testing data into it. The instantiated example model consists of a total of 87 separate integrity constraints represented mainly by database trigger add statements. In Appendix D, there are only names and input arguments of generated ICs, while the complete Cypher code of the instantiated model can be found on the enclosed CD.

The Cypher code representing the instantiated model is designed for the Neo4j in version 4.4.18 with the APOC library installed in version 4.4.0.14. These versions are henceforward used for the entire process of testing described below. In order to enable database triggers in the Neo4j, it is necessary to create an `apoc.conf` configuration file and add two lines specified in Listing 24.

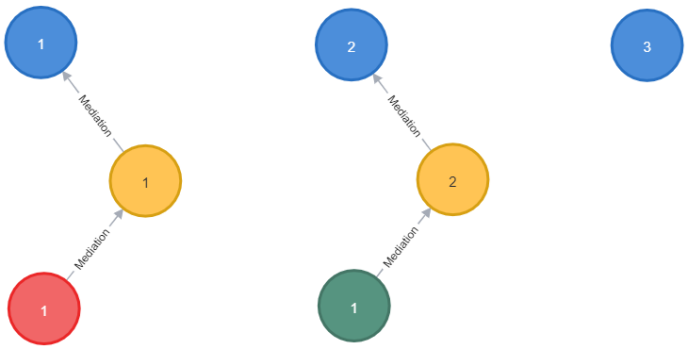
```
apoc.trigger.enabled=true
apoc.trigger.refresh=60000
```

Listing 24: Configuration enabling database triggers in Neo4j

On a properly set up Neo4j database, the instantiated model can be deployed by simply running the corresponding Cypher code using tools such as Neo4j Browser or Neo4j Cypher Shell. This first part of positive testing proves that all 87 Cypher statements are syntactically correct and that the instantiated model can be successfully deployed. The presence of all deployed integrity constraints can be confirmed using the `CALL apoc.trigger.list()` and the `SHOW CONSTRAINTS` commands.

After successful deployment, it is now possible to load testing data into the database. The Cypher code for insertion of valid testing data can be found in Appendix E. This testing data includes the creation statements of three different electrical devices, one rechargeable battery, one DC power supply, and electrical connections between the first two electrical devices and suitable DC power sources. Even in this case, it is possible to insert testing data simply by running the above-mentioned Cypher code. This second part of positive testing proves that all 7 nodes and 4 relationships can be successfully inserted, and their insertion does not violate any of the 87 previously deployed ICs. Inserted testing data can be shown using the `MATCH (n) RETURN n` commands.

Figure 7.1: Visualization of inserted testing data – labeled-property graph



Visualization of inserted testing data can be seen in Figure 7.1. Circles represent nodes as instances of classes as well as lines represent relationships as instances of associations. Different colors of nodes denote different Kinds – blue corresponds to `ElectricalDevice`, yellow indicates `ElectricalConnection`, red matches `Battery`, and green denotes `DCPowerSupply`. Numbers on nodes describe values of ID attributes.

A summary overview describing the results of both steps of positive testing can be found below in Table 7.1.

Table 7.1: Summary results of positive testing

Testing step	# statements	# successful
model deployment	87	87
valid data loading	7	7

7.2 Negative testing

Negative testing tries to verify that the deployed instantiated model contains all of the necessary integrity constraints and that it is not possible to load invalid data violating the semantics given by the original OntoUML model into the database. The goal is not to test every single integrity constraint but rather to test a sufficiently comprehensive sample of invalid data, based on which it is already possible to detect potential integrity errors or, on the contrary, to assume that the OntoUML model is instantiated correctly.

Negative tests are represented by data manipulation Cypher queries that verify the presence and functionality of individual integrity constraints. Only the names and several examples of these queries are listed in this chapter. They are divided into subsections according to the corresponding step of the example model instantiation. The complete list of negative test Cypher queries can be found on the enclosed CD. As a default state for the correct execution of these queries, it is assumed that the instantiated example OntoUML model has already been deployed to the database and that valid testing data has been loaded as a part of previous positive testing.

7.2.1 Kinds and Properties

The first set of negative tests verifies the correct instantiation of Kinds and their properties. Names of 7 test cases from this set are listed in Listing 25.

```
1 TestPropertyDatatypeOnCreate()
2 TestPropertyDatatypeOnUpdate()
3 TestPropertyPresentOnCreate()
4 TestPropertyPresentOnRemove()
5 TestPropertyUniqueOnCreate()
6 TestPropertyUniqueOnUpdate()
7 TestNotCombinationSingleOnCreate()
```

Listing 25: Names of negative test cases for Kinds

The `TestPropertyDatatypeOnCreate` negative test attempts to create an `ElectricalDevice` instance that has a `String` instead of a `Float` as a value of the `nominalVoltage` attribute. As an example, the Cypher query of this test can be seen in Listing 26. Similarly, the `TestPropertyDatatypeOnUpdate` negative test attempts to update the `nominalVoltage` value of an existing `ElectricalDevice` instance and change it to a `String` datatype. Execution of both queries should be prevented by the `PROPERTY_MUST_BE_OF_DATATYPE` integrity constraint.

```
1 //TEST: TestPropertyDatatypeOnCreate()
2 //FAIL PROPERTY_MUST_BE_OF_DATATYPE(ElectricalDevice, nominalCurrent, FLOAT)
3 CREATE (e
4     :ElectricalDevice {
5         id: 10,
6         nominalVoltage: "some text",
7         nominalCurrent: 0.025
8 });
```

Listing 26: Negative test of attribute value datatype – Kinds

The `TestPropertyPresentOnCreate` negative test attempts to create an `ElectricalDevice` instance without setting a value for the `nominalVoltage` attribute. In the same manner, the `TestPropertyPresentOnRemove` negative test attempts to remove the `nominalVoltage` attribute of an existing `ElectricalDevice` instance. The Cypher query of the secondly mentioned negative test can be seen in Listing 27. Execution of both negative test queries should be prevented by the `PROPERTY_MUST_BE_PRESENT` integrity constraint.

```
1 //TEST: TestPropertyPresentOnRemove()
2 //FAIL PROPERTY_MUST_BE_PRESENT(ElectricalDevice, nominalCurrent)
3 MATCH (e:ElectricalDevice {id: 1})
4 REMOVE e.nominalVoltage;
```

Listing 27: Negative test of attribute value presence – Kinds

The `TestPropertyUniqueOnCreate` negative test attempts to create an `ElectricalDevice` instance with the value of `id` attribute that has been already set to another `ElectricalDevice` instance. Almost identically, the `TestPropertyUniqueOnUpdate` negative test attempts to update the value of `id` attribute of one existing `ElectricalDevice` instance to the value that has been already set to another existing `ElectricalDevice` instance. Execution of both queries should be prevented by the `PROPERTY_MUST_BE_UNIQUE` integrity constraint.

The last `TestNotCombinationSingleOnCreate` negative test attempts to create an `ElectricalDevice` instance that is a `DCPowerSupply` instance at the same time. Although attribute values of both mentioned Kinds are correctly set, execution of this negative test query should be prevented by corresponding `LABEL_CANNOT_BE_IN_COMBINATION` integrity constraint since all the instances can only have a single identity.

7.2.2 Subkinds

The second set of negative tests verifies the correct instantiation of Subkinds. Names of 5 test cases from this set are listed in Listing 28.

```
1 TestInCombinationGeneralizationOnCreate()
2 TestInCombinationCompleteOnCreate()
3 TestInCombinationCompleteOnRemove()
4 TestNotCombinationDisjointOnCreate()
5 TestNotCombinationDisjointOnUpdate()
```

Listing 28: Names of negative test cases for Subkinds

The `TestInCombinationGeneralizationOnCreate` test attempts to create a `Rechargeable` instance that is not a `Battery` instance at the same time. Execution of this negative test query should be prevented by corresponding `LABEL_MUST_BE_IN_COMBINATION` integrity constraint because there exists a generalization between these two classes.

```
1  //TEST: TestInCombinationCompleteOnCreate()
2  //FAIL LABEL_MUST_BE_IN_COMBINATION(Battery, {Rechargeable, Nonrechargeable})
3  CREATE (b
4      :EnergyStorage :DCVoltageSource
5      :Battery :Charged {
6          id: 10,
7          capacity: 3.0,
8          actualVoltage: 1.45,
9          nominalVoltage: 1.2,
10         maximalVoltage: 1.45,
11         minimalVoltage: 1.2,
12         chargeDate: date("2022-10-28")
13 });
```

Listing 29: Negative test of IsCovering GS property – Subkinds

The `TestInCombinationCompleteOnCreate` negative test attempts to create a `Battery` instance that is neither `Rechargeable` nor `Nonrechargeable`

instance. The Cypher query of this test can be seen in Listing 29. In a similar way, the `TestInCombinationCompleteOnRemove` negative test attempts to remove a `Rechargeable` label of an existing `Battery` instance. Execution of both test queries should be prevented by the `LABEL_MUST_BE_IN_COMBINATION` integrity constraint as all mentioned classes are in a generalization set with IsCovering property.

```
1 //TEST: TestNotCombinationDisjointOnUpdate()
2 //FAIL LABEL_CANNOT_BE_IN_COMBINATION(Nonrechargeable, {Rechargeable})
3 MATCH (b:Battery {id: 1})
4 SET b:Nonrechargeable
5 SET b += {shelflife: 18};
```

Listing 30: Negative test of IsDisjoint GS property – Subkinds

The `TestNotCombinationDisjointOnCreate` negative test attempts to create a `Rechargeable` instance that is a `Nonrechargeable` instance at the same time. Equivalently, the `TestNotCombinationDisjointOnUpdate` negative test attempts to assign a `Nonrechargeable` label to an existing `Rechargeable` instance. The Cypher query of the secondly mentioned negative test can be seen in Listing 30. Execution of both negative test queries should be prevented by the corresponding `LABEL_CANNOT_BE_IN_COMBINATION` integrity constraint as all the mentioned classes are in a generalization set with IsDisjoint property.

7.2.3 Categories

The third set of negative tests verifies the correct instantiation of Categories. Names of 4 test cases from this set are listed in Listing 31.

```
1 TestInCombinationHasIdentityOnCreate()
2 TestInCombinationGeneralizationOnCreate()
3 TestInCombinationGeneralizationOnRemove()
4 TestNotCombinationDisjointOnCreate()
```

Listing 31: Names of negative test cases for Categories

The `TestInCombinationHasIdentityOnCreate` negative test attempts to create an `EnergyStorage` instance without being a `Battery` instance at the same time. The Cypher query of this negative test can be seen as an example in Listing 32. Execution of this negative test query should be prevented by corresponding `LABEL_MUST_BE_IN_COMBINATION` integrity constraint since all the instances must have an identity.

```
//TEST: TestInCombinationHasIdentityOnCreate()
//FAIL LABEL_MUST_BE_IN_COMBINATION(EnergyStorage, {Battery})
CREATE (e
    :EnergyStorage {
        capacity: 3.0
});
```

Listing 32: Negative test of assigned identity – Categories

The `TestInCombinationGeneralizationOnCreate` test attempts to create a `Battery` instance that simultaneously is not an `EnergyStorage` instance. Similarly, the `TestInCombinationGeneralizationOnRemove` negative test attempts to remove a `EnergyStorage` label of an existing `Battery` instance. The Cypher query of this test can be seen in Listing 33. Execution of both negative test queries should be prevented by the `LABEL_MUST_BE_IN_COMBINATION` integrity constraint as there exists a generalization between both classes.

```
//TEST: TestInCombinationGeneralizationOnRemove()
//FAIL LABEL_MUST_BE_IN_COMBINATION(Battery, {EnergyStorage})
MATCH (b:Battery {id: 1})
REMOVE b:EnergyStorage;
```

Listing 33: Negative test of generalization – Categories

The last `TestNotCombinationDisjointOnCreate` negative test attempts to create a `Battery` instance that is a `DCPowerSupply` instance at the same time. Execution of this test query should be prevented by the corresponding `LABEL_CANNOT_BE_IN_COMBINATION` integrity constraint because both classes are in a generalization set with IsDisjoint property.

7.2.4 Phases

The fourth set of negative tests verifies the correct instantiation of Phases. Names of 5 test cases from this set are listed in Listing 34.

```
TestInCombinationGeneralizationOnCreate()
TestInCombinationCompleteOnCreate()
TestInCombinationCompleteOnRemove()
TestNotCombinationDisjointOnCreate()
TestNotCombinationDisjointOnUpdate()
```

Listing 34: Names of negative test cases for Phases

The `TestInCombinationGeneralizationOnCreate` negative test tries to create a `Charged` instance that simultaneously is not a `Battery` instance. The Cypher query of this negative test can be seen below in Listing 35. Execution of this negative test query should be prevented by the corresponding `LABEL_MUST_BE_IN_COMBINATION` integrity constraint as there exists a generalization between both classes.

```
1  //TEST: TestInCombinationGeneralizationOnCreate()
2  //FAIL LABEL_MUST_BE_IN_COMBINATION(Charged, {Battery})
3  CREATE (c
4      :Charged {
5          chargeDate: date("2022-10-28")
6  });
```

Listing 35: Negative test of generalization – Phases

The `TestInCombinationCompleteOnCreate` negative test tries to create a `Battery` instance that simultaneously is not an instance of any of Phase classes. The Cypher query of this test can be seen in Listing 36. In the same manner, the `TestInCombinationCompleteOnRemove` negative test attempts to remove a `Charged` label of an existing `Battery` instance. Execution of both test queries should be prevented by the `LABEL_MUST_BE_IN_COMBINATION` integrity constraint since all mentioned classes are in a generalization set with IsCovering property.

```
1  //TEST: TestInCombinationCompleteOnCreate()
2  //FAIL LABEL_MUST_BE_IN_COMBINATION(Battery, {Charged, Used, Discharged})
3  CREATE (b
4      :EnergyStorage :DCVoltageSource
5      :Battery :Rechargeable {
6          id: 10,
7          capacity: 3.0,
8          actualVoltage: 1.45,
9          nominalVoltage: 1.2,
10         maximalVoltage: 1.45,
11         minimalVoltage: 1.2,
12         cyclelife: 1000
13 });
```

Listing 36: Negative test of IsCovering GS property – Phases

The `TestNotCombinationDisjointOnCreate` negative test tries to create a `Charged` instance that is a `Used` instance at the same time. In the same way,

the `TestNotCombinationDisjointOnUpdate` negative test attempts to assign a `Used` label to an existing `Charged` instance. Execution of both negative test queries should be prevented by the `LABEL_CANNOT_BE_IN_COMBINATION` integrity constraint as all mentioned classes are in a generalization set with IsDisjoint property.

7.2.5 Roles, RoleMixin and Relator

The last set of negative tests verifies the correct instantiation of Roles, RoleMixin, and Relator. Names of 8 test cases are listed in Listing 37.

```
1 TestInCombinationHasIdentityOnCreate()
2 TestInCombinationHasIdentityOnRemove()
3 TestInCombinationGeneralizationMixinOnCreate()
4 TestInCombinationGeneralizationMixinOnRemove()
5 TestInCombinationGeneralizationKindOnCreate()
6 TestRelationshipPresentOnCreate()
7 TestRelationshipPresentOnRemove()
8 TestRelationshipPresentOnDelete()
```

Listing 37: Names of negative test cases for Roles

The `TestInCombinationHasIdentityOnCreate` negative test tries to create a `ConnectedDCVoltageSource` instance that is neither `ConnectedBattery` instance nor `ConnectedDCPowerSupply` instance. As an example, the Cypher query of this negative test can be seen in Listing 38. Almost identically, the `TestInCombinationHasIdentityOnRemove` negative test attempts to remove a `ConnectedBattery` label of an existing `ConnectedDCVoltageSource` instance. Execution of both negative test queries should be prevented by the `LABEL_MUST_BE_IN_COMBINATION` IC since all instances must have an identity.

```
1  //TEST: TestInCombinationHasIdentityOnCreate()
2  //FAIL LABEL_MUST_BE_IN_COMBINATION(ConnectedDCVoltageSource, { ... })
3  MATCH (e:ElectricalDevice {id: 3})
4  SET e:ConnectedElectricalDevice
5  CREATE (c
6      :ConnectedDCVoltageSource {
7          drawnCurrent: 0.025
8  })-[:Mediation]->(r
9      :ElectricalConnection {id: 10, resistance: 0.05}
10 )-[:Mediation]->(e);
```

Listing 38: Negative test of assigned identity – Roles

The `TestInCombinationGeneralizationMixinOnCreate` test tries to create a `ConnectedBattery` instance that is not a `ConnectedDCVoltageSource` instance. Similarly, the `TestInCombinationGeneralizationMixinOnRemove` negative test attempts to remove a `ConnectedDCVoltageSource` label of an existing `ConnectedBattery` instance. Execution of both negative test queries should be prevented by the corresponding `LABEL_MUST_BE_IN_COMBINATION` IC because there is a generalization between both classes.

The `TestInCombinationGeneralizationKindOnCreate` test attempts to create a `ConnectedBattery` instance that simultaneously is not a `Battery` instance. Execution of this negative test query should be prevented by the corresponding `LABEL_MUST_BE_IN_COMBINATION` IC because also, in this case, there is a generalization between these classes.

The `TestRelationshipPresentOnCreate` negative test attempts to create an `ElectricalConnection` instance that does not have a relationship with any `ConnectedElectricalDevice` instance. In a similar manner, both the `TestRelationshipPresentOnRemove` negative test and the `TestRelationshipPresentOnDelete` negative test attempt to simulate an equivalent situation by removing a `ConnectedElectricalDevice` label of an `ElectricalDevice` instance or by deleting the `ElectricalDevice` instance completely. The Cypher query of the lastly mentioned negative test can be seen in Listing 39. Execution of all three test queries should be prevented by the `RELATIONSHIP_MUST_BE_PRESENT` IC because a mandatory association exists between the two mentioned classes.

```
1 //TEST: TestRelationshipPresentOnDelete()
2 //FAIL RELATIONSHIP_MUST_BE_PRESENT(ElectricalConnection, ...)
3 MATCH (e:ElectricalDevice {id: 1})
4 DETACH DELETE e;
```

Listing 39: Negative test of association presence – Roles

A summary overview describing the results of individual steps of negative testing can be found below in Table 7.2.

Table 7.2: Summary results of negative testing

Testing step	# test cases	# successful
Kinds, Properties	7	7
Subkinds	5	5
Categories	4	4
Phases	5	5
Roles, RoleMixin	8	8
Total	29	29

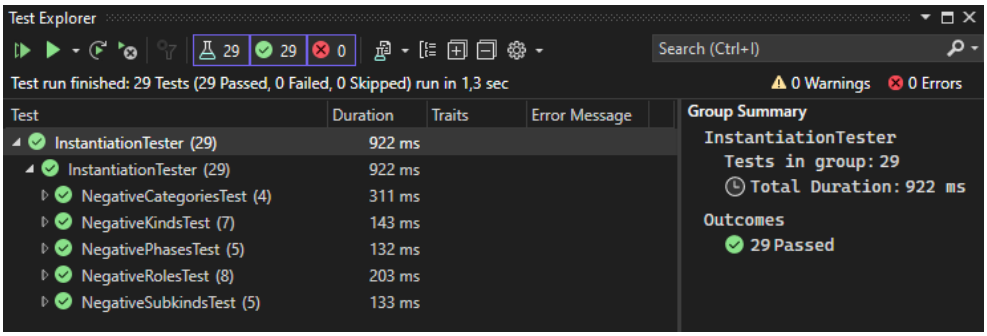
7.3 Automated testing

Both positive and negative testing can be done by manually executing individual Cypher queries of test cases from previous sections. However, such an approach cannot be considered effective, especially not in a situation when testing has to be often repeated during the change or extension of the transformation procedure.

Fortunately, the process of testing can be easily automated. As a demonstration of this fact, the thesis contains the `InstantiationTester` utility, also written in the C# language and using the NUnit framework. This framework is usually used for unit testing applications written on top of the .NET platform. However, in this case, the goal is not to test separate parts of the automated transformation tool developed as a part of this thesis but rather to test the procedure of transformation of OntoUML diagrams into Neo4j graph database integrity constraints itself. The benefit of the NUnit framework is that it can be used for this purpose as well.

The `InstantiationTester` utility contains a test method for every negative test mentioned in the previous section. Test methods are divided into test suites according to their testing steps. Every test method connects to the Neo4j database with preset credentials and tries to verify that the execution of its negative test query is prevented by an integrity constraint mechanism. The results of negative testing performed using this utility can be seen in Figure 7.2.

Figure 7.2: Results of NUnit automated testing utility – screenshot



Chapter 8

Discussion

After the case study that demonstrates the example OntoUML model instantiation in the Neo4j graph database has been successfully conducted, the achieved results need to be further discussed. Therefore, this thesis's last chapter evaluates the benefits, possible issues, and some other aspects of the proposed approach. Furthermore, because the concluded case study is intended to serve as a basis for further technical publications in this field of research, the last section of this chapter also addresses possible future work.

8.1 Benefits of proposed approach

The main benefit of the conducted case study is that the comprehensive procedure for the transformation of OntoUML diagrams in the Neo4j database system has been successfully designed, used, and tested. The proposed procedure already covers a significant part of the OntoUML, including most of the OntoUML class stereotypes. Moreover, since the procedure is expressed with algorithmic precision, it can be fully automated.

The possibility of full automation results in a considerable degree of convenience for all types of potential users. Thanks to this fact, the proposed procedure can be used not only by software developers to ensure the integrity of data in their information systems but also by domain experts during the design and validation of OntoUML models describing their domain of interest. The OntoUML diagrams can de facto become the high-level data schemas for otherwise schema-free graph database systems.

Even though it might seem that the whole idea of this approach goes against the philosophy of schema-free database systems, it is not the case. Another benefit is that the instantiated models are closed to modifications but still open for extension. In other words, it is still possible to add arbitrary data to the database beyond the scope of the defined OntoUML model as long as there is no violation of the semantics imposed by this model. The benefits of the schema-free philosophy can thus be considered preserved.

8.2 Possible issues of proposed approach

The first issue of the proposed approach is given by the fact that although it covers a significant part of the OntoUML, there still exists a considerable number of constructs whose instantiation needs to be addressed. It mainly includes the Mixin class stereotype and most of the relationship stereotypes, except for the Mediation stereotype. In addition to these stereotypes, it is also necessary to resolve the instantiation of attributes with a multiplicity higher than one and the instantiation of associations with a limited multiplicity upper bound. The question remains whether the instantiation of the above-mentioned OntoUML constructs in graph databases is possible or whether it results in as yet undiscovered problems.

The second issue relates to another important part of OntoUML and UFO theory, the rigidity principle, which also has not been addressed in this case study. The reason is that by using the available integrity constraint mechanisms, it cannot be ensured in a meaningful way that an instance of a rigid type remains the instance of this type under all circumstances. Although it would theoretically be possible, for example, to prevent the removal of a label representing a rigid type, it is not possible to prevent the deletion of the entire instance and the creation of a new instance with the same identifier that is no longer an instance of a given rigid type. Keeping a history of instance identifiers and a list of associated rigid types would be necessary to solve this problem. However, due to its complexity, such an approach would not be considered applicable in practice.

The third problem arises from the need to use the OCL in the case of more complex OntoUML models. While some OntoUML models, such as the example OntoUML model, can be entirely expressed by only one UML class diagram supplemented with the appropriate stereotypes, more complex OntoUML models must also contain code in the OCL. Their semantics would otherwise be impossible to express only by a class diagram itself. The transformation of semantics captured using the OCL into graph database integrity constraints represents another critical area of research that needs to be addressed in the future beyond the scope of this case study.

Another possible issue of the proposed approach is its performance. This case study aims to provide a proof of concept, not to measure or try to estimate the overall performance of the proposed solutions. However, as can be seen from the conducted case study, a considerable number of integrity constraints are required to ensure the semantics resulting from the example OntoUML model. Therefore, with more complex OntoUML models, performance problems could occur due to the high number of complex integrity constraints. Measuring the actual performance achieved by the proposed approach is an important prerequisite for its subsequent practical use.

The last issue is that only the multi-labeled graph databases with powerful enough constraint mechanisms can be used in the case of the proposed

approach. As explained in chapter 3, not all graph databases have the same capabilities. Assigning multiple labels to one vertex is critical to express that a given vertex represents an instance of multiple classes at the same time. A sufficiently strong constraint mechanism is, in turn, necessary to preserve the semantics resulting from the individual OntoUML constructs and axioms of the UFO theory. Unfortunately, many graph databases do not meet these two conditions.

8.3 Future works

Future technical publications based on this thesis should primarily address the issues outlined in the previous section. The proposed procedure needs to be extended to include an instantiation of the remaining class and relationship stereotypes, different multiplicities of attributes or associations, and constraints written in the OCL. The ultimate goal should be to create a complete procedure covering all concepts and elements of OntoUML.

Moreover, in this thesis, the proposed procedure has been tested only using the set of appropriately chosen test cases. As part of future technical publications, formal proof using the modal logic should be made that the instantiated models always satisfy all axioms of underlying UFO theory.

For future practical use, measuring the data manipulation performance of models instantiated using the proposed procedure is necessary. It can be assumed that the specific database triggers used in this thesis as a proof of concept can be optimized to achieve better results. Measured performance results should ideally be compared across multiple different graph databases. This comparison, of course, includes modifying the proposed procedure for the transformation of OntoUML models in other suitable multi-labeled property graph databases. Further research should also provide a performance comparison between models instantiated in graph and relational databases.

The last but no less important direction that future research should take is the extension and improvement of tools enabling the automated transformation of OntoUML models. Reading the input data from standard formats (e.g., JSON, XML) should be possible. Furthermore, there should be as close a connection as possible with modeling tools such as OpenPonk. Also, deploying the transformed models to the target database should be made as easy as possible. The goal is to maximize convenience. Only this way will real users use the proposed procedure in practice.

Conclusion

This thesis aimed to investigate the suitability of graph databases and the implementation of integrity constraints for the instantiation of models expressed in the OntoUML notation. In accordance with the set goal, the thesis attempted to verify the following research hypotheses:

H1 It was hypothesized that it is possible to instantiate a valid OntoUML model in a graph database and implement all necessary integrity constraints.

H2 It was hypothesized that it is possible to use the same principles of instantiation for all graph databases.

As a preliminary step to achieve the set goal, a literature review summarizing knowledge on topics related to this thesis was carried out. The first part of this literature review defined and explained all necessary terms and principles related to OntoUML and UFO theory. The second part then dealt with graph databases, their specifics, and their capabilities.

As its main part, this thesis included a case study. Its purpose was to demonstrate a proposed way of how can be an example OntoUML model instantiated in a chosen graph database system.

At first, this case study introduced an appropriately chosen example OntoUML model on which the instantiation in graph database could be demonstrated. This example model was related to the ontology of electrical devices and their electrical power sources.

Second, the case study analyzed the suitability of several different graph database systems for the instantiation of the OntoUML models. This analysis found that the graph databases with a multi-labeled property graph model and database trigger support can be considered the most suitable. Based on this finding, the Neo4j graph database system was hence used during the rest of the case study since it met all the mentioned criteria.

Third, the case study designed the way of OntoUML model instantiation in the Neo4j database. During the design process, the case study first defined how the constructs of the OntoUML model could be mapped to the constructs available in the Neo4j. Next, it introduced a comprehensive procedure for the transformation of OntoUML diagrams into a list of specific integrity constraints. As a last step of the design process, it described the precise implementation of these integrity constraints in the Neo4j, mainly using the database triggers and the APOC library.

Then, the case study carried out an instantiation of the example OntoUML model according to the designed procedure. In order to perform this instantiation most effectively and avoid possible errors, it also introduced an automated transformation tool written in the C# programming language. The instantiation itself was performed in several subsequent steps, in which additional constructs and classes with additional stereotypes were gradually added. The result of the instantiation was a list of integrity constraints preserving the semantics given by the example OntoUML model.

Finally, the case study verified the correct instantiation of the example OntoUML model through a series of test cases. The testing itself was divided into two subsequent phases. In the first phase of positive testing, it was verified that the instantiated model behaves as expected when using valid testing data. In the second phase of negative testing, it was verified that the execution of queries that violate the semantics given by the instantiated example OntoUML model is prevented by the relevant integrity constraints. All test cases in both testing phases were successful. As a result, the instantiation of the example OntoUML model can be considered correct.

As its last part, this thesis contained a discussion of achieved results. This discussion evaluated the benefits and the issues of the proposed approach and also addressed possible future work following the outputs of this thesis.

Based on the successfully conducted case study that provided a tested code of Neo4j integrity constraints corresponding to the example OntoUML model, the research hypothesis **H1** can be *verified*.

Based on the results of the conducted analysis of graph database suitability as well as based on the conclusions of the final discussion, unfortunately, the research hypothesis **H2** has to be *falsified*.

In view of all the facts mentioned above – the results of the analysis of graph database suitability, the overall outputs of the successfully conducted case study, including the procedure for the transformation of OntoUML diagrams in the Neo4j database system, the conclusions of the final discussion, and the verification or falsification of the set research hypotheses – the goal of this thesis can be considered accomplished.

Bibliography

1. RYBOLA, Z.; PERGL, R. Towards OntoUML for software engineering: introduction to the transformation of OntoUML into relational databases. In: *Enterprise and Organizational Modeling and Simulation: 12th International Workshop, EOMAS 2016, Held at CAiSE 2016, Ljubljana, Slovenia, June 13, 2016, Selected Papers 12*. Cham: Springer, 2016, pp. 67–83. ISBN 978-3-319-49453-1.
2. GUIDONI, G. L.; ALMEIDA, J. P.; GUIZZARDI, G. Preserving Conceptual Model Semantics in the Forward Engineering of Relational Schemas. *Frontiers in Computer Science*. 2022, vol. 4, p. 155.
3. SUCHÁNEK, M. OntoUML specification Documentation. In: *OntoUML community portal* [online]. OntoUML Community, 2022 [visited on 2023-03-12]. Available from: `https://ontouml.readthedocs.io/_/downloads/en/latest/pdf/`.
4. GUIZZARDI, G. *Ontological foundations for structural conceptual models*. Enschede: Telematica Instituut / CTIT, 2005. ISBN 90-75176-81-3. PhD thesis. University of Twente.
5. GUIZZARDI, G.; FONSECA, C.; BENEVIDES, A. B.; ALMEIDA, J. P.; PORELLO, D.; SALES, T. P. Endurant types in ontology-driven conceptual modeling: Towards OntoUML 2.0. In: *International conference on conceptual modeling*. Cham: Springer, 2018, pp. 136–150. ISBN 978-3-030-00847-5.
6. FONSECA, C.; PORELLO, D.; GUIZZARDI, G.; ALMEIDA, J. P.; GUARINO, N. Relations in ontology-driven conceptual modeling. In: *International Conference on Conceptual Modeling*. Cham: Springer, 2019, pp. 28–42. ISBN 978-3-030-33223-5.
7. GUIZZARDI, G.; FONSECA, C.; ALMEIDA, J. P.; SALES, T. P.; BENEVIDES, A. B.; PORELLO, D. Types and taxonomic structures

in conceptual modeling: A novel ontological theory and engineering support. *Data & Knowledge Engineering*. 2021, vol. 134, p. 101891.

8. RYBOLA, Z.; PERGL, R. Towards OntoUML for software engineering: Transformation of kinds and subkinds into relational databases. *Computer Science and Information Systems*. 2017, vol. 14, no. 3, pp. 913–937.
9. ALMEIDA, J. P.; FALBO, R.; GUIZZARDI, G. Events as entities in ontology-driven conceptual modeling. In: *Conceptual Modeling: 38th International Conference, ER 2019, Salvador, Brazil, November 4–7, 2019, Proceedings 38*. Cham: Springer, 2019, pp. 469–483. ISBN 978-3-030-33223-5.
10. GUIZZARDI, G.; FONSECA, C.; ALMEIDA, J. P.; BOTTI BENEVIDES, A.; PORELLO, D.; PRINCE SALES, T. UFO: Unified foundational ontology. *Applied ontology*. 2022, vol. 17, no. 1, pp. 167–210.
11. PERGL, R.; SALES, T. P.; RYBOLA, Z. Towards OntoUML for software engineering: from domain ontology to implementation model. In: *Model and Data Engineering: Third International Conference, MEDI 2013, Amantea, Italy, September 25-27, 2013. Proceedings 3*. Berlin: Springer, 2013, pp. 249–263. ISBN 978-3-642-41365-0.
12. RAJIV, C. *Database Management System (DBMS): A Practical Approach, 5th Edition*. New Delhi: S. Chand Publishing, 2016. ISBN 978-93-856-7634-5.
13. HARRINGTON, J. L. *Relational Database Design and Implementation*. Cambridge: Elsevier Science, 2016. ISBN 978-0-12-804399-8.
14. DARWEN, H. *An Introduction to Relational Database Theory*. Frederiksberg: Ventus Publishing, 2009. ISBN 978-87-403-0202-8.
15. POKORNÝ, J. NoSQL databases: a step to database scalability in web environment. In: *Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services*. New York: Association for Computing Machinery, 2011, pp. 278–283. ISBN 978-1-4503-0784-0.
16. POKORNÝ, J. Integration of relational and NoSQL databases. *Vietnam Journal of Computer Science*. 2019, vol. 6, no. 04, pp. 389–405.
17. ROBINSON, I.; WEBBER, J.; EIFREM, E. *Graph databases: new opportunities for connected data*. Sebastopol: O'Reilly Media, Inc., 2015. ISBN 978-1-491-93089-2.
18. RAJ, S. *Neo4j High Performance*. Birmingham: Packt Publishing Ltd., 2015. ISBN 978-1-78355-515-4.

19. SHIMPI, D.; CHAUDHARI, S. An overview of graph databases. In: *IJCA proceedings on international conference on recent trends in information technology and computer science*. New York: Foundation of Computer Science, 2012, pp. 16–22. ISSN 0975 - 8887.

20. POKORNÝ, J. Graph databases: their power and limitations. In: *Computer Information Systems and Industrial Management: 14th IFIP TC 8 International Conference, CISIM 2015, Warsaw, Poland, September 24-26, 2015, Proceedings 14*. Cham: Springer, 2015, pp. 58–69. ISBN 978-3-319-24369-6.

21. AVEY, B. Labeled vs Typed Property Graphs – All Graph Databases are not the same. In: *Geek Culture* [online]. A Medium Corporation, 2021 [visited on 2023-01-20]. Available from: `https://medium.com/geekculture/labeled-vs-typed-property-graphs-all-graph-databases-are-not-the-same-efdbc782f099`.

22. DE VIRGILIO, R.; MACCIONI, A.; TORLONE, R. Model-driven design of graph databases. In: *Conceptual Modeling: 33rd International Conference, ER 2014, Atlanta, GA, USA, October 27-29, 2014. Proceedings 33*. Cham: Springer, 2014, pp. 172–185. ISBN 978-3-319-12206-9.

23. MCCREARY, D. The Neighborhood Walk Story. In: *Medium* [online]. A Medium Corporation, 2018 [visited on 2023-01-20]. Available from: `https://dmccreary.medium.com/how-to-explain-index-free-adjacency-to-your-manager-1a8e68ec664a`.

24. WEINBERGER, C. Index Free Adjacency or Hybrid Indexes for Graph Databases. In: *ArangoDB Blog* [online]. ArangoDB, Inc., 2016 [visited on 2023-01-20]. Available from: `https://www.arangodb.com/2016/04/index-free-adjacency-hybrid-indexes-graph-databases/`.

25. SOLIDIT GMBH. Neo4j System Properties. In: *DB-Engines* [online]. solidIT gmbh, 2023 [visited on 2023-01-21]. Available from: `https://db-engines.com/en/system/Neo4j`.

26. NEO4J, INC. Graph database concepts. In: *Neo4j Documentation* [online]. Neo4j, Inc., 2023 [visited on 2023-01-24]. Available from: `https://neo4j.com/docs/getting-started/current/appendix/graphdb-concepts/`.

27. NEO4J, INC. Welcome to Neo4j. In: *Neo4j Documentation* [online]. Neo4j, Inc., 2023 [visited on 2023-01-24]. Available from: `https://neo4j.com/docs/getting-started/current/`.

28. NEO4J, INC. Modeling designs. In: *Neo4j Documentation* [online]. Neo4j, Inc., 2023 [visited on 2023-01-24]. Available from: `https://neo4j.com/docs/getting-started/current/data-modeling/modeling-designs/`.

29. NEO4J, INC. Clustering Introduction. In: *Neo4j Operations Manual* [online]. Neo4j, Inc., 2023 [visited on 2023-01-24]. Available from: https://neo4j.com/docs/operations-manual/current/clustering/introduction/.

30. NEO4J, INC. Introduction. In: *Neo4j Operations Manual* [online]. Neo4j, Inc., 2023 [visited on 2023-01-24]. Available from: https://neo4j.com/docs/operations-manual/current/introduction/.

31. NEO4J, INC. Overview. In: *Cypher Manual* [online]. Neo4j, Inc., 2023 [visited on 2023-01-25]. Available from: https://neo4j.com/docs/cypher-manual/current/introduction/cypher_overview/.

32. NEO4J, INC. Triggers. In: *APOC Documentation 4.4* [online]. Neo4j, Inc., 2023 [visited on 2023-01-24]. Available from: https://neo4j.com/labs/apoc/4.4/background-operations/triggers/.

33. NEO4J, INC. Graph algorithms. In: *Neo4j Graph Data Science* [online]. Neo4j, Inc., 2023 [visited on 2023-01-25]. Available from: https://neo4j.com/docs/graph-data-science/current/algorithms/.

34. JANUSGRAPH AUTHORS. JanusGraph. In: *JanusGraph Homepage* [online]. The Linux Foundation, 2023 [visited on 2023-01-29]. Available from: https://janusgraph.org/.

35. SOLIDIT GMBH. JanusGraph System Properties. In: *DB-Engines* [online]. solidIT gmbh, 2023 [visited on 2023-01-21]. Available from: https://db-engines.com/en/system/JanusGraph.

36. JANUSGRAPH AUTHORS. Introduction. In: *JanusGraph Documentation* [online]. The Linux Foundation, 2023 [visited on 2023-02-02]. Available from: https://docs.janusgraph.org/.

37. JANUSGRAPH AUTHORS. JanusGraph Data Model. In: *JanusGraph Documentation* [online]. The Linux Foundation, 2023 [visited on 2023-02-02]. Available from: https://docs.janusgraph.org/advanced-topics/data-model/.

38. JANUSGRAPH AUTHORS. Storage Backends Introduction. In: *JanusGraph Documentation* [online]. The Linux Foundation, 2023 [visited on 2023-02-03]. Available from: https://docs.janusgraph.org/storage-backend/.

39. JANUSGRAPH AUTHORS. Schema and Data Modeling. In: *JanusGraph Documentation* [online]. The Linux Foundation, 2023 [visited on 2023-02-02]. Available from: https://docs.janusgraph.org/schema/.

40. JANUSGRAPH AUTHORS. Graph Partitioning. In: *JanusGraph Documentation* [online]. The Linux Foundation, 2023 [visited on 2023-02-03]. Available from: https://docs.janusgraph.org/advanced-topics/partitioning/.

41. JANUSGRAPH AUTHORS. Transactions. In: *JanusGraph Documentation* [online]. The Linux Foundation, 2023 [visited on 2023-02-03]. Available from: `https://docs.janusgraph.org/interactions/transactions/`.

42. JANUSGRAPH AUTHORS. Gremlin Query Language. In: *JanusGraph Documentation* [online]. The Linux Foundation, 2023 [visited on 2023-02-03]. Available from: `https://docs.janusgraph.org/getting-started/gremlin/`.

43. SOLIDIT GMBH. TigerGraph System Properties. In: *DB-Engines* [online]. solidIT gmbh, 2023 [visited on 2023-01-21]. Available from: `https://db-engines.com/en/system/TigerGraph`.

44. TIGERGRAPH, INC. Defining a Graph Schema. In: *TigerGraph Documentation* [online]. TigerGraph, Inc., 2023 [visited on 2023-01-25]. Available from: `https://docs.tigergraph.com/gsql-ref/current/ddl-and-loading/defining-a-graph-schema`.

45. TIGERGRAPH, INC. Internal Architecture. In: *TigerGraph Documentation* [online]. TigerGraph, Inc., 2023 [visited on 2023-01-26]. Available from: `https://docs.tigergraph.com/tigergraph-server/current/intro/internal-architecture`.

46. TIGERGRAPH, INC. Transaction Processing and ACID Support. In: *TigerGraph Documentation* [online]. TigerGraph, Inc., 2023 [visited on 2023-01-29]. Available from: `https://docs.tigergraph.com/tigergraph-server/current/intro/transaction-and-acid`.

47. TIGERGRAPH, INC. GSQL Language Reference. In: *TigerGraph Documentation* [online]. TigerGraph, Inc., 2023 [visited on 2023-01-25]. Available from: `https://docs.tigergraph.com/gsql-ref/current/intro/`.

48. TIGERGRAPH, INC. TigerGraph Graph Data Science Library. In: *TigerGraph Documentation* [online]. TigerGraph, Inc., 2023 [visited on 2023-01-29]. Available from: `https://docs.tigergraph.com/graph-ml/current/intro/`.

49. GUIDONI, G. L.; ALMEIDA, J. P.; GUIZZARDI, G. Transformation of ontology-based conceptual models into relational schemas. In: *International Conference on Conceptual Modeling*. Cham: Springer, 2020, pp. 315–330. ISBN 978-3-030-62521-4.

50. NEEDHAM, M.; HODLER, A. E. *Graph Algorithms: Practical Examples in Apache Spark and Neo4j*. Sebastopol: O'Reilly Media, Inc., 2019. ISBN 978-1-492-05781-9.

51. NEO4J, INC. Constraints. In: *Cypher Manual* [online]. Neo4j, Inc., 2023 [visited on 2023-01-24]. Available from: `https://neo4j.com/docs/cypher-manual/current/constraints/`.

52. ORIENTDB LTD. Graph Database Properties. In: *OrientDB Manual - version 2.2.x* [online]. OrientDB Ltd., 2016 [visited on 2023-01-21]. Available from: `https://orientdb.com/docs/2.2.x/Graph-Schema-Property.html`.

53. VAISMAN, A.; ZIMÁNYI, E. *Data Warehouse Systems: Design and Implementation*. Berlin: Springer, 2022. Data-Centric Systems and Applications. ISBN 978-3-662-65167-4.

54. CHONOLES, M. J.; SCHARDT, J. A. *UML 2 For Dummies*. New York: Wiley, 2011. ISBN 978-1-118-08538-7.

55. WEILKIENS, T.; OESTEREICH, B. *UML 2 Certification Guide: Fundamental and Intermediate Exams*. New York: Elsevier Science, 2010. The MK/OMG Press. ISBN 978-0-08-046651-4.

56. OBJECT MANAGEMENT GROUP. *The OMG Specifications Catalog*. OMG Unified Modeling Language (OMG UML) Version 2.5 [online]. Object Management Group, Inc., 2015 [visited on 2023-03-12]. Available from: `https://www.omg.org/spec/UML/2.5/PDF`.

57. MILICEV, D. *Model-Driven Development with Executable UML*. Indianapolis: Wiley, 2009. Wrox guides. ISBN 978-0-470-53599-8.

Appendix A

Abbreviations

AC Alternating Current

ACID Atomicity, Consistency, Integration, Durability

APOC Awesome Procedures On Cypher

BASE Basically Available, Soft State, Eventually Consistent

BFS Breadth-first search

CD Compact Disc

CLI Command Line Interface

DBMS Database Management System

DC Direct Current

DFS Depth-first search

GDB Graph Database

GS Generalization Set

GSQL Graph SQL

IC Integrity Constraint

JSON JavaScript Object Notation

MDD Model-driven Development

NoSQL Not only SQL

OCL Object Constraint Language

OOP Object-oriented Paradigm

RDB Relational Database

SQL Structured Query Language

UFO Unified Foundational Ontology

UML Unified Modeling Language

XML Extensible Markup Language

Appendix B

Procedure transforming OntoUML diagrams

```
1  foreach (Class c in diagram.Classes)
2  {
3      // ----- ATTRIBUTES
4      foreach (Attribute a in c.Attributes)
5      {
6          Constraints.Add(PROPERTY_MUST_BE_OF_DATATYPE,
7              c.Name,
8              a.Name,
9              a.DataType.Name
10         );
11         if (a.Multiplicity.LowerBound > 0)
12             Constraints.Add(PROPERTY_MUST_BE_PRESENT,
13                 c.Name,
14                 a.Name
15             );
16         if (a.IsID)
17             Constraints.Add(PROPERTY_MUST_BE_UNIQUE,
18                 c.Name,
19                 a.Name
20             );
21     }
22
23     // ----- STEREOTYPES
24     if (not c.Stereotype.HasIdentity)
25     {
26         relatedSortals = diagram.Classes
27             .Filter(r => r.Stereotype.HasIdentity)
28             .Filter(r => r.Superclasses
29                 .Exists(s => c.Equals(s) or c.Superclasses.Contains(s)));
30         intransitivelyRelatedSortals = relatedSortals
31             .Filter(i => i.Superclasses
32                 .ForAll(s => not relatedSortals.Contains(s)));
33         Constraints.Add(
34             LABEL_MUST_BE_IN_COMBINATION,
35             c.Name,
36             intransitivelyRelatedSortals.Map(i => i.Name)
37         );
38     }
39
```

```
40      if (c.Stereotype.ProvidesIdentity)
41      {
42          otherIdentityProviders = diagram.Classes
43              .Filter(p => p.Stereotype.ProvidesIdentity)
44              .OtherThan(c);
45          if(otherIdentityProviders.Count > 0)
46              Constraints.Add(
47                  LABEL_CANNOT_BE_IN_COMBINATION,
48                  c.Name,
49                  otherIdentityProviders
50                      .Map(p => p.Name)
51              );
52      }
53  }
54
55  // ----- GENERALIZATIONS
56  foreach (Generalization g in diagram.Generalizations)
57  {
58      Constraints.Add(
59          LABEL_MUST_BE_IN_COMBINATION,
60          g.Subclass.Name,
61          { g.Superclass.Name }
62      );
63  }
64
65  // ----- GENERALIZATION SETS
66  foreach (GeneralizationSet gs in diagram.GeneralizationSets)
67  {
68      if (gs.IsDisjoint)
69          foreach (Class c in gs.Subclasses)
70              Constraints.Add(
71                  LABEL_CANNOT_BE_IN_COMBINATION,
72                  c.Name,
73                  gs.Subclasses
74                      .OtherThan(c)
75                      .Map(s => s.Name)
76              );
77      if (gs.IsCovering)
78          Constraints.Add(
79              LABEL_MUST_BE_IN_COMBINATION,
80              gs.Superclass.Name,
81              gs.Subclasses
82                  .Map(s => s.Name)
83          );
84  }
85
86  // ----- ASSOCIATIONS
87  foreach (BinaryAssociation a in diagram.Associations)
88  {
89      if(a.FirstMultiplicity.LowerBound > 0)
90          constraints.Add(RELATIONSHIP_MUST_BE_PRESENT,
91              a.SecondClass.Name
92              a.FirstClass.Name,
93          );
94      if (a.SecondMultiplicity.LowerBound > 0)
95          constraints.Add(RELATIONSHIP_MUST_BE_PRESENT,
96              a.FirstClass.Name
97              a.SecondClass.Name,
98          );
99  }
```

Appendix C

Input C# representation of the example model

```
1  // ----- CLASSES & ATTRIBUTES
2  Class ED = new Class("ElectricalDevice", ClassStereotype.Kind);
3  ED.Attributes.Add(new ClassAttribute("id", DataType.Integer, true));
4  ED.Attributes.Add(new ClassAttribute("nominalCurrent", DataType.Float));
5  ED.Attributes.Add(new ClassAttribute("nominalVoltage", DataType.Float));
6  Class DCPS = new Class("DCPowerSupply", ClassStereotype.Kind);
7  DCPS.Attributes.Add(new ClassAttribute("id", DataType.Integer, true));
8  DCPS.Attributes.Add(new ClassAttribute("inputVoltage", DataType.Float));
9  DCPS.Attributes.Add(new ClassAttribute("maximalCurrent", DataType.Float));
10 Class BA = new Class("Battery", ClassStereotype.Kind);
11 BA.Attributes.Add(new ClassAttribute("id", DataType.Integer, true));
12 BA.Attributes.Add(new ClassAttribute("maximalVoltage", DataType.Float));
13 BA.Attributes.Add(new ClassAttribute("minimalVoltage", DataType.Float));
14
15 Class RBA = new Class("Rechargeable", ClassStereotype.Subkind);
16 RBA.Attributes.Add(new ClassAttribute("cyclelife", DataType.Integer));
17 Class NRBA = new Class("Nonrechargeable", ClassStereotype.Subkind);
18 NRBA.Attributes.Add(new ClassAttribute("shelflife", DataType.Integer));
19
20 Class ES = new Class("EnergyStorage", ClassStereotype.Category);
21 ES.Attributes.Add(new ClassAttribute("capacity", DataType.Float));
22 Class DCVS = new Class("DCVoltageSource", ClassStereotype.Category);
23 DCVS.Attributes.Add(new ClassAttribute("actualVoltage", DataType.Float));
24 DCVS.Attributes.Add(new ClassAttribute("nominalVoltage", DataType.Float));
25
26 Class CHBA = new Class("Charged", ClassStereotype.Phase);
27 CHBA.Attributes.Add(new ClassAttribute("chargeDate", DataType.Date));
28 Class UBA = new Class("Used", ClassStereotype.Phase);
29 UBA.Attributes.Add(new ClassAttribute("lastUseDate", DataType.Date));
30 Class DBA = new Class("Discharged", ClassStereotype.Phase);
31 DBA.Attributes.Add(new ClassAttribute("dischargeDate", DataType.Date));
32
33 Class CED = new Class("ConnectedElectricalDevice", ClassStereotype.Role);
34 Class EC = new Class("ElectricalConnection", ClassStereotype.Relator);
35 EC.Attributes.Add(new ClassAttribute("id", DataType.Integer, true));
36 EC.Attributes.Add(new ClassAttribute("resistance", DataType.Float));
37 Class CDCVS = new Class("ConnectedDCVoltageSource", ClassStereotype.RoleMixin);
38 CDCVS.Attributes.Add(new ClassAttribute("drawnCurrent", DataType.Float));
39 Class CBA = new Class("ConnectedBattery", ClassStereotype.Role);
```

```
40  Class CDCPS = new Class("ConnectedDCPowerSupply", ClassStereotype.Role);
41
42  // ----- GENERALIZATIONS
43  Generalization BARBA = new Generalization(BA, RBA);
44  Generalization BANRBA = new Generalization(BA, NRBA);
45
46  Generalization ESBA = new Generalization(ES, BA);
47  Generalization DCVSDCPS = new Generalization(DCVS, DCPS);
48  Generalization DCVSBA = new Generalization(DCVS, BA);
49
50  Generalization BACBA = new Generalization(BA, CHBA);
51  Generalization BAUBA = new Generalization(BA, UBA);
52  Generalization BADBA = new Generalization(BA, DBA);
53
54  Generalization EDCED = new Generalization(ED, CED);
55  Generalization CDCVSCDCPS = new Generalization(CDCVS, CDCPS);
56  Generalization CDCVSCBA = new Generalization(CDCVS, CBA);
57  Generalization DCPSCDCPS = new Generalization(DCPS, CDCPS);
58  Generalization BACBA = new Generalization(BA, CBA);
59
60  // ----- GENERALIZATION SETS
61  GeneralizationSet BASubkindSet = new GeneralizationSet(new() { BARBA, BANRBA }, true,
    ↪  true);
62  GeneralizationSet DCVSKindSet = new GeneralizationSet(new() { DCVSDCPS, DCVSBA },
    ↪  false, true);
63  GeneralizationSet BAPhaseSet = new GeneralizationSet(new() { BACBA, BAUBA, BADBA },
    ↪  true, true);
64  GeneralizationSet CDCVSRoleSet = new GeneralizationSet(new() { CDCVSCDCPS, CDCVSCBA },
    ↪  false, true);
65
66  // ----- ASSOCIATIONS
67  BinaryAssociation ECCED = new BinaryAssociation(EC, CED, Multiplicity.OneToMany,
    ↪  Multiplicity.OneToMany);
68  BinaryAssociation CDCVSEC = new BinaryAssociation(CDCVS, EC, Multiplicity.OneToMany,
    ↪  Multiplicity.OneToMany);
```

Appendix D

Output constraint names of the instantiated model

```
1  // ----- ATTRIBUTES
2  PROPERTY_MUST_BE_OF_DATATYPE(ElectricalDevice, id, INTEGER)
3  PROPERTY_MUST_BE_OF_DATATYPE(ElectricalDevice, nominalCurrent, FLOAT)
4  PROPERTY_MUST_BE_OF_DATATYPE(ElectricalDevice, nominalVoltage, FLOAT)
5  PROPERTY_MUST_BE_OF_DATATYPE(DCPowerSupply, id, INTEGER)
6  PROPERTY_MUST_BE_OF_DATATYPE(DCPowerSupply, inputVoltage, FLOAT)
7  PROPERTY_MUST_BE_OF_DATATYPE(DCPowerSupply, maximalCurrent, FLOAT)
8  PROPERTY_MUST_BE_OF_DATATYPE(Battery, id, INTEGER)
9  PROPERTY_MUST_BE_OF_DATATYPE(Battery, maximalVoltage, FLOAT)
10 PROPERTY_MUST_BE_OF_DATATYPE(Battery, minimalVoltage, FLOAT)
11 PROPERTY_MUST_BE_OF_DATATYPE(Rechargeable, cyclelife, INTEGER)
12 PROPERTY_MUST_BE_OF_DATATYPE(Nonrechargeable, shelflife, INTEGER)
13 PROPERTY_MUST_BE_OF_DATATYPE(EnergyStorage, capacity, FLOAT)
14 PROPERTY_MUST_BE_OF_DATATYPE(DCVoltageSource, actualVoltage, FLOAT)
15 PROPERTY_MUST_BE_OF_DATATYPE(DCVoltageSource, nominalVoltage, FLOAT)
16 PROPERTY_MUST_BE_OF_DATATYPE(Charged, chargeDate, DATE)
17 PROPERTY_MUST_BE_OF_DATATYPE(Used, lastUseDate, DATE)
18 PROPERTY_MUST_BE_OF_DATATYPE(Discharged, dischargeDate, DATE)
19 PROPERTY_MUST_BE_OF_DATATYPE(ElectricalConnection, id, INTEGER)
20 PROPERTY_MUST_BE_OF_DATATYPE(ElectricalConnection, resistance, FLOAT)
21 PROPERTY_MUST_BE_OF_DATATYPE(ConnectedDCVoltageSource, drawnCurrent, FLOAT)
22
23 PROPERTY_MUST_BE_PRESENT(ElectricalDevice, id)
24 PROPERTY_MUST_BE_PRESENT(ElectricalDevice, nominalCurrent)
25 PROPERTY_MUST_BE_PRESENT(ElectricalDevice, nominalVoltage)
26 PROPERTY_MUST_BE_PRESENT(DCPowerSupply, id)
27 PROPERTY_MUST_BE_PRESENT(DCPowerSupply, inputVoltage)
28 PROPERTY_MUST_BE_PRESENT(DCPowerSupply, maximalCurrent)
29 PROPERTY_MUST_BE_PRESENT(Battery, id)
30 PROPERTY_MUST_BE_PRESENT(Battery, maximalVoltage)
31 PROPERTY_MUST_BE_PRESENT(Battery, minimalVoltage)
32 PROPERTY_MUST_BE_PRESENT(Rechargeable, cyclelife)
33 PROPERTY_MUST_BE_PRESENT(Nonrechargeable, shelflife)
34 PROPERTY_MUST_BE_PRESENT(EnergyStorage, capacity)
35 PROPERTY_MUST_BE_PRESENT(DCVoltageSource, actualVoltage)
36 PROPERTY_MUST_BE_PRESENT(DCVoltageSource, nominalVoltage)
37 PROPERTY_MUST_BE_PRESENT(Charged, chargeDate)
38 PROPERTY_MUST_BE_PRESENT(Used, lastUseDate)
39 PROPERTY_MUST_BE_PRESENT(Discharged, dischargeDate)
```

```
40 PROPERTY_MUST_BE_PRESENT(ElectricalConnection, id)
41 PROPERTY_MUST_BE_PRESENT(ElectricalConnection, resistance)
42 PROPERTY_MUST_BE_PRESENT(ConnectedDCVoltageSource, drawnCurrent)
43
44 PROPERTY_MUST_BE_UNIQUE(ElectricalDevice, id)
45 PROPERTY_MUST_BE_UNIQUE(DCPowerSupply, id)
46 PROPERTY_MUST_BE_UNIQUE(Battery, id)
47 PROPERTY_MUST_BE_UNIQUE(ElectricalConnection, id)
48
49 // ----- STEREOTYPES
50 LABEL_MUST_BE_IN_COMBINATION(EnergyStorage, {Battery})
51 LABEL_MUST_BE_IN_COMBINATION(DCVoltageSource, {DCPowerSupply, Battery})
52 LABEL_MUST_BE_IN_COMBINATION(ConnectedDCVoltageSource, {ConnectedBattery,
   ↪ ConnectedDCPowerSupply})
53
54 LABEL_CANNOT_BE_IN_COMBINATION(ElectricalDevice, {DCPowerSupply, Battery,
   ↪ ElectricalConnection})
55 LABEL_CANNOT_BE_IN_COMBINATION(DCPowerSupply, {ElectricalDevice, Battery,
   ↪ ElectricalConnection})
56 LABEL_CANNOT_BE_IN_COMBINATION(Battery, {ElectricalDevice, DCPowerSupply,
   ↪ ElectricalConnection})
57 LABEL_CANNOT_BE_IN_COMBINATION(ElectricalConnection, {ElectricalDevice, DCPowerSupply,
   ↪ Battery})
58
59 // ----- GENERALIZATIONS
60 LABEL_MUST_BE_IN_COMBINATION(Rechargeable, {Battery})
61 LABEL_MUST_BE_IN_COMBINATION(Nonrechargeable, {Battery})
62 LABEL_MUST_BE_IN_COMBINATION(Battery, {EnergyStorage})
63 LABEL_MUST_BE_IN_COMBINATION(DCPowerSupply, {DCVoltageSource})
64 LABEL_MUST_BE_IN_COMBINATION(Battery, {DCVoltageSource})
65 LABEL_MUST_BE_IN_COMBINATION(Charged, {Battery})
66 LABEL_MUST_BE_IN_COMBINATION(Used, {Battery})
67 LABEL_MUST_BE_IN_COMBINATION(Discharged, {Battery})
68 LABEL_MUST_BE_IN_COMBINATION(ConnectedElectricalDevice, {ElectricalDevice})
69 LABEL_MUST_BE_IN_COMBINATION(ConnectedDCPowerSupply, {ConnectedDCVoltageSource})
70 LABEL_MUST_BE_IN_COMBINATION(ConnectedBattery, {ConnectedDCVoltageSource})
71 LABEL_MUST_BE_IN_COMBINATION(ConnectedDCPowerSupply, {DCPowerSupply})
72 LABEL_MUST_BE_IN_COMBINATION(ConnectedBattery, {Battery})
73
74 // ----- GENERALIZATION SETS
75 LABEL_MUST_BE_IN_COMBINATION(Battery, {Rechargeable, Nonrechargeable})
76 LABEL_MUST_BE_IN_COMBINATION(Battery, {Charged, Used, Discharged})
77
78 LABEL_CANNOT_BE_IN_COMBINATION(Rechargeable, {Nonrechargeable})
79 LABEL_CANNOT_BE_IN_COMBINATION(Nonrechargeable, {Rechargeable})
80 LABEL_CANNOT_BE_IN_COMBINATION(Charged, {Used, Discharged})
81 LABEL_CANNOT_BE_IN_COMBINATION(Used, {Charged, Discharged})
82 LABEL_CANNOT_BE_IN_COMBINATION(Discharged, {Charged, Used})
83 LABEL_CANNOT_BE_IN_COMBINATION(DCPowerSupply, {Battery})
84 LABEL_CANNOT_BE_IN_COMBINATION(Battery, {DCPowerSupply})
85 LABEL_CANNOT_BE_IN_COMBINATION(ConnectedDCPowerSupply, {ConnectedBattery})
86 LABEL_CANNOT_BE_IN_COMBINATION(ConnectedBattery, {ConnectedDCPowerSupply})
87
88 // ----- ASSOCIATIONS
89 RELATIONSHIP_MUST_BE_PRESENT(ConnectedElectricalDevice, ElectricalConnection)
90 RELATIONSHIP_MUST_BE_PRESENT(ElectricalConnection, ConnectedElectricalDevice)
91 RELATIONSHIP_MUST_BE_PRESENT(ElectricalConnection, ConnectedDCVoltageSource)
92 RELATIONSHIP_MUST_BE_PRESENT(ConnectedDCVoltageSource, ElectricalConnection)
```

Appendix E

Testing data for the instantiated model

```
// ----- CLEAR
MATCH(n) DETACH DELETE (n);

// ----- ELECTRICAL DEVICES
CREATE (e1
    :ElectricalDevice {
        id: 1,
        nominalVoltage: 1.2,
        nominalCurrent: 0.025
});

CREATE (e2
    :ElectricalDevice {
        id: 2,
        nominalVoltage: 5.0,
        nominalCurrent: 0.2
});

CREATE (e3
    :ElectricalDevice {
        id: 3,
        nominalVoltage: 12.0,
        nominalCurrent: 1.5
});

// ----- BATTERIES
CREATE (b1
    :EnergyStorage :DCVoltageSource
    :Battery :Rechargeable :Charged {
        id: 1,
        capacity: 3.0,
        actualVoltage: 1.45,
        nominalVoltage: 1.2,
        maximalVoltage: 1.45,
        minimalVoltage: 1.2,
        cyclelife: 1000,
        chargeDate: date("2022-10-28")
});

```

```
40  // ----- DC POWER SUPPLIES
41  CREATE (s1
42      :DCVoltageSource
43      :DCPowerSupply {
44          id: 1,
45          actualVoltage: 4.95,
46          nominalVoltage: 5.0,
47          inputVoltage: 230.0,
48          maximalCurrent: 1.5
49  });
50
51  // ----- ELECTRICAL CONNECTIONS
52  MATCH (b:Battery {id: 1}), (e:ElectricalDevice {id: 1})
53  SET e:ConnectedElectricalDevice
54  SET b:ConnectedBattery:ConnectedDCVoltageSource
55  SET b += {drawnCurrent: 0.025}
56  CREATE (b)-[:Mediation]->(r
57      :ElectricalConnection {id: 1, resistance: 0.05}
58  )-[:Mediation]->(e);
59
60  MATCH (s:DCPowerSupply {id: 1}), (e:ElectricalDevice {id: 2})
61  SET e:ConnectedElectricalDevice
62  SET s:ConnectedDCPowerSupply:ConnectedDCVoltageSource
63  SET s += {drawnCurrent: 0.25}
64  CREATE (s)-[:Mediation]->(r
65      :ElectricalConnection {id: 2, resistance: 0.01}
66  )-[:Mediation]->(e);
67
```

Appendix F

Content of the enclosed CD

- `readme.txt` ..... brief description of the content and deployment manual
- `data`
  - `01-procedure.txt` ..... pseudocode of the transformation procedure
  - `02-input-csharp.cs` ..... C# representation of the example model
  - `03-input-json.json` ..... JSON representation of the example model
  - `04-comments.txt` ..... constraint names of the instantiated model
  - `05-triggers.cql` ..... Neo4j constraints of the instantiated model
  - `06-data.cql` ..... valid testing data for the instantiated model
  - `07-tests.cql` ..... negative tests for the instantiated model
- `exe`
  - `DiagramInstantiator-win-x64.exe` ..... executable file for Windows
  - `DiagramInstantiator-linux-x64` ..... executable file for Linux
- `src`
  - `thesis` ..... source code of the thesis in LaTeX format
  - `utilities`
    - `DiagramInstantiator` ..... source code of DiagramInstantiator
    - `InstantiationTester` ..... source code of InstantiationTester
- `text`
  - `thesis.pdf` ..... text of the thesis in PDF format