

Université de Montréal

**A Framework for Domain-Specific Modeling on Graph
Databases**

par

Nikitchyn Vitalii

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en Informatique

Orientation Génie logicielle

December 21, 2021

Université de Montréal

Faculté des arts et des sciences

Ce mémoire intitulé

**A Framework for Domain-Specific Modeling
on Graph Databases**

présenté par

Vitalii Nikitchyn

a été évalué par un jury composé des personnes suivantes:

Gena Hahn

(président-rapporteur)

Eugene Syriani

(directeur de recherche)

Michalis Famelis

(membre du jury)

Résumé

La complexité du logiciel augmente tout le temps: les systèmes deviennent plus grands et plus complexes. La modélisation est un élément central de génie logiciel pour relever les défis de la complexité. Cependant, un défi majeur auquel est confronté le développement de logiciels axés sur les modèles est l'évolutivité des outils de modélisation avec une taille croissante de modèles. Certaines initiatives ont commencé à explorer la modélisation tout en stockant des modèles dans une base de données de graphes. Dans cette thèse, nous présentons NMF, un framework pour créer et éditer des modèles dans une base de données Neo4j élevée à l'abstraction du langage de modélisation.

Mots-clé: Ingénierie dirigée par les modèles, langage dédié, éditeur dédié, base de données de graphes

Abstract

Software complexity increases all the time: systems become larger and more complex. Modeling is a central part of software engineering to tackle challenges of complexity. However, a prominent challenge model-driven software development is facing is scalability of modeling tools with a growing size of models. Some initiatives started exploring modeling while storing models in a graph database. In this thesis, we present NMF, a framework to create and edit MDE models in a Neo4j database lifted to the abstraction of the modeling language.

Keywords: Model-Driven Engineering, domain-specific language, domain-specific editor, graph database

Contents

Résumé	5
Abstract	7
List of figures	13
Liste des sigles et des abréviations.....	15
Chapter 1. Introduction.....	17
1.1. Context.....	17
1.2. Problematic.....	17
1.3. Contribution.....	18
1.4. Outline.....	18
Chapter 2. Background and Related Work	19
2.1. Model-driven engineering	19
2.1.1. MDE principles.....	19
2.1.2. Modeling language engineering.....	20
2.1.2.1. Abstract syntax	20
2.1.2.2. Concrete syntax	21
2.1.2.3. Semantics	21
2.1.3. Models	21
2.1.4. Metamodeling	22
2.1.5. Tools for model editing	24
2.2. Neo4j graph database.....	27
2.2.1. Graph structure in Neo4j database.....	27
2.2.2. Deployment.....	27
2.2.3. Cypher query language.....	28
2.2.3.1. Create query	29

2.2.3.2.	Read query	30
2.2.3.3.	Update query	30
2.2.3.4.	Delete query	31
2.2.3.5.	Unwind query	31
2.2.4.	Neo4j query planner	32
2.2.5.	Parametrized queries	32
2.2.6.	APOC library	33
2.3.	MDE in databases	34
2.3.1.	A model scalability	34
2.3.2.	MDE for relational DB	35
2.3.3.	MDE for NoSQL DB	37
2.3.3.1.	Neo4EMF	37
Chapter 3.	Neo Modeling Framework	39
3.1.	NMF modules	39
3.1.1.	NMF-loader	40
3.1.2.	NMF-editor	40
3.1.3.	NMF-generator	40
3.2.	Running example	41
3.2.1.	Graf DSL	41
3.2.2.	Storing models and metamodels in the database	41
3.2.3.	Generic manipulation of models	43
3.2.4.	Domain-specific manipulation of models	43
Chapter 4.	NMF-Loader	47
4.1.	Data Mapping	47
4.1.1.	Multilevel mapping	49
4.1.2.	Mapping example	49
4.2.	NMF-loader components	50
4.2.1.	Reader	50
4.2.1.1.	Extracting the elements from an input model	50
4.2.1.2.	Buffering	51
4.2.1.3.	Reading process	52
4.2.2.	Mapper	53

4.2.3.	Loader	54
4.2.4.	Loader example	55
Chapter 5.	NMF-Editor	57
5.1.	Model editing logic	57
5.2.	NMF-editor components	58
5.2.1.	Graph controller	58
5.2.2.	Node controller	59
5.2.3.	Controller states	61
5.3.	Partial graph editing	63
5.3.1.	Element creation	65
5.3.2.	Element updates	65
5.3.3.	Element removal	66
5.3.3.1.	Nodes removal	66
5.3.3.2.	Relationships removal	67
5.3.4.	Reading elements	68
5.4.	Editor example	69
Chapter 6.	NMF Generator	73
6.1.	Domain-specific editor components	73
6.1.1.	Domain-specific model manager	74
6.1.2.	Domain-specific object controller	74
6.1.3.	Domain-specific utilities	75
6.2.	Generation process	75
6.3.	Object controller generation	76
6.3.1.	Header generation	76
6.3.2.	Attributes setter/getter generation	77
6.3.3.	Associations getter/setter generation	78
6.3.3.1.	Reference association generation	79
6.3.3.2.	Containment association generation	81
6.4.	Inheritance handling	81
6.4.1.	Header generating	81
6.4.2.	Containment association generation	82

6.5. Model manager generation	83
6.5.1. Header generation	83
6.5.2. Contents generation	84
6.6. Domain specific editing with NMF	84
Chapter 7. Evaluation	87
7.1. Comparing features of Neo4EMF and NMF	87
7.2. Performance analysis	87
7.2.1. Experiments setup	87
7.2.2. Experiment results	88
7.2.2.1. Create operation	88
7.2.2.2. Update operation	90
7.2.2.3. Remove operation	91
7.2.2.4. Read operation	93
7.3. Discussion	93
Chapter 8. Conclusion	95
8.1. Summary	95
8.2. Outlook	95
References	97
Appendix A. NMF Manual	99
A.1. Prerequisites	99
A.2. NMF architecture	100
A.2.1. NMF-editor	100
A.2.2. NMF-loader	100
A.2.3. NMF-generator	101
Appendix B. Graf metamodel in Ecore format	103
Appendix C. Graf model instance serialized in XMI format	105

List of figures

2.1	Structure of a modeling language	20
2.2	Metamodel of the LIBRARY language expressed in UML.....	21
2.3	Abstract syntax of the model instance of LIBRARY DSL.....	22
2.4	Four-layer architecture of MDE	22
2.5	ECORE meta-metamodel	23
2.6	LIBRARY model opened in EMF tree-view	24
2.7	LIBRARY metamodel opened in EMF graphical editor.....	25
2.8	Generated API for the LIBRARY DSL	25
2.9	Metamodel of a data structure in Neo4j database	27
3.1	Dependency between modules in NMF	39
3.2	The metamodel of the GRAF DSL in UML class diagrams.....	41
3.3	The model instance of the GRAF DSL	42
3.4	The metamodel of the GRAF DSL serialized in the Neo4j database	42
4.1	NMF-loader data flow	50
4.2	The architecture of the reader component of NMF-loader	51
4.3	<i>EModel</i> traversal process.....	52
5.1	The NMF-editor layered architecture and its components	58
5.2	The structure of a node controller.....	60
5.3	Node controller states.....	62
5.4	NMF-editor cache fragmentation.....	63
6.1	Domain-specific editor architecture.....	73
6.2	Domain-specific object controller.....	75
6.3	Mapping multiple inheritance to object controller	82

6.4	Inheritance hierarchy example	83
7.1	Create model objects performance	89
7.2	Create reference associations performance	90
7.3	Update objects performance	91
7.4	Performance of removing objects in depth	92
7.5	Performance of reference associations removal	92
7.6	Performance of reading related objects	93

Liste des sigles et des abréviations

MDE	Model Driven Engineering
DSL	Domain Specific Language
DSM	Domain Specific Model
API	Application Programming Interface
EMF	Eclipse Modeling Framework
NMF	Neo Modeling Framework
CRUD	Create, Read, Update, and Delete (operations)
DBMS	Database Management System
JVM	Java Virtual Machine
UML	Unified Modeling Language

XML	Extensible Markup Language
XMI	XML Metadata Interchange
IDE	Integrated Development Environment
OOP	Object Oriented Programming
RAM	Random Access Memory (of a computer)

Chapter 1

Introduction

1.1. Context

Nowadays, software is everywhere. Software programs are ubiquitously used to solve problems from the real physical world. Most often, software depends on data. A cause of the continually increasing software complexity [1] is the increasing amount of required data. A large amount of data causes a new problem: a modern software needs means to handle, store, and manipulate such amount of data.

Software engineering is a discipline that focuses on building high-quality software. One of the quality attributes of software is scalability – the ability of software to deal with more and more data [2]. Nowadays, most software uses databases – specialized software primarily designed to handle large data. The databases provide data scale as well as data manipulation operations quite efficiently.

Model-Driven Engineering (MDE) is a software development methodology that aims to describe (software) systems, as well as a system behavior through models [3]. MDE focuses on constructing, analyzing, and transforming the models. In MDE terminology, a model is an abstraction of a software system that can highlight the desired aspect with a particular level of detail. An abstraction allows to focus more on problem details of a domain rather than on the platform-specific and implementation details. Thus, MDE serves to facilitate a software development process rising the level of abstraction through modeling.

1.2. Problematic

Models, which are primarily artifacts in MDE, should be viewed as ordinary data sets. Therefore, MDE models have common data characteristics, including scalability. The scalability challenges in software engineering are even more present in MDE: with the continuous growth of models size, MDE is still far from efficient data scaling. Modeling tools fail to handle large models in terms of time, responsiveness, and crashes. For instance, if we load a

model of size 1GB in Eclipse IDE [4, 5] (the famous MDE modeling environment) – it will crash [6]. In MDE some work has been done to resolve this problem. For example, there is an initiative to port modeling into databases (relational [7], and graph databases [8]).

In this thesis we address the problem of scalability in MDE by providing a framework to perform modeling activities directly in databases. Although the existing solutions resolve some scalability problems, they are not reusable outside of Eclipse IDE: it is not possible to integrate them with other technologies. Therefore, we aim to provide a lightweight modeling framework with minimum dependencies that is able to solve the model scalability problem.

1.3. Contribution

The goal of this thesis is to provide a seamless integration of modeling in memory or in a database to scale activities to models of arbitrary size. More specifically, we have developed a modeling framework (NMF) for the Neo4j database [9] with the following contributions:

- a framework to manipulate MDE models directly in the database
- a loader to port an existing model or metamodel from commonly used XMI model persistence format [10] into the database
- a code generator – to produce a domain-specific API for any domain (i.e., DSL). The generated code is closer to the abstraction of the problem domain rather than to the solution domain.

The whole framework is packaged under an easy to use library available at [11].

1.4. Outline

The whole thesis consists of 8 chapters. In Chapter 2, we present an overview of MDE methodology and existing approaches to interact with large models. Chapter 3 briefly describes our solution to handle MDE models – Neo Modeling Framework (NMF). Chapters 4, 5, and 6 provide detailed descriptions of the three main modules of NMF. In Chapter 7, we present the performance testing result of NMF. Finally, we conclude in Chapter 8.

Chapter 2

Background and Related Work

In this chapter, we introduce a background needed to understand the purpose of NMF. In Section 2.1, we review the concept of Model-driven engineering (MDE). In Section 2.2, we overview in detail the Neo4j database – particular storage we use for storing MDE-related data. In Section 2.3, we overview existing modeling solutions that implement the MDE concepts.

2.1. Model-driven engineering

Nowadays, software is ubiquitously used to solve problems from the real world. A software development process requires developers to map concepts of a particular domain from the real world into a virtual software world (i.e., software solution). The common way to perform this task is to build abstractions that match a real world system but are expressed in common programming technologies such as programming languages. Since software complexity constantly increases all the time, the development process requires more cognitive effort from the programmer to build those abstractions [3].

Over time, software development technologies have evolved considerably: from low-level machine-oriented practices (such as assembly language) to modern high-level programming languages (such as Java or C#). Every new generation of development technologies aims to facilitate the development process by augmenting the level of abstraction: from the computer (hardware) concept up to a concept of a particular application domain. The higher level of abstraction is, the more a programmer can focus on a solution rather than on solving typical problems imposed by concrete development tools [1, 12].

2.1.1. MDE principles

Model-driven engineering (MDE) is a software development paradigm that allows to reduce a software development complexity raising the level of abstraction [3]. MDE relies on models to describe a system design and different system workflows, abstracting from concrete development tools and technologies. MDE promotes a concept where a **model** and **model transformations** are

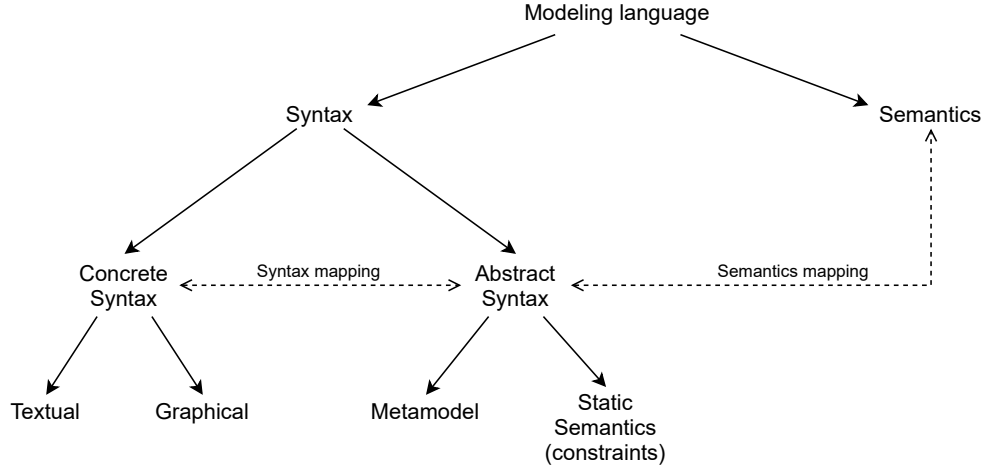


Fig. 2.1. Structure of a modeling language

considered as basic units of development. This means that a software solution can be expressed at an abstract level: in a model using concepts and notations that directly match the original domain. Afterwards, the model can be transformed into executable artifacts through model transformation.

A modeler usually operates on domain-specific models (DSM): conceptual models that represent a particular field of knowledge. DSM highlight a desired aspect of a system for a domain expert, which is familiar with the domain concepts and notations he usually works with. A model is a platform-independent artifact that can describe only relevant parts of a system, omitting implementation details. Therefore, it ensures a high level of abstraction.

Model transformations are used to express the behavior of models [13]. Model transformation is the process of converting one model into another one according to some predefined transformation rules. The resulting model may represent any software artifact, such as configuration files, data structure definitions, message schemes, or even a programming code.

2.1.2. Modeling language engineering

To define a model, a modeler needs a modeling language. A modeling language defines a set of rules to express a precise model. Usually, these rules come from a particular application domain. To distinguish between languages depending on the specific domain, the concept of domain-specific languages (DSL) was introduced. DSLs provide concepts, rules, and notations tailored to a particular field of knowledge. Therefore, a domain expert can use a DSL to define DSMs. Fig. 2.1 shows typical components of a modeling language. It consists of three components: abstract syntax, concrete syntax, and semantics.

2.1.2.1. Abstract syntax. The abstract syntax of a DSL serves to define the domain. It consists of two components: the structural part and the optional static semantics part. The structural part is usually specified by a metamodel using UML class diagrams notation [14]. It defines the allowed types, relationships between them, and attributes for each type. The static semantics is a set of additional constraints to ensure the well-formedness of DSMs. The common way to define

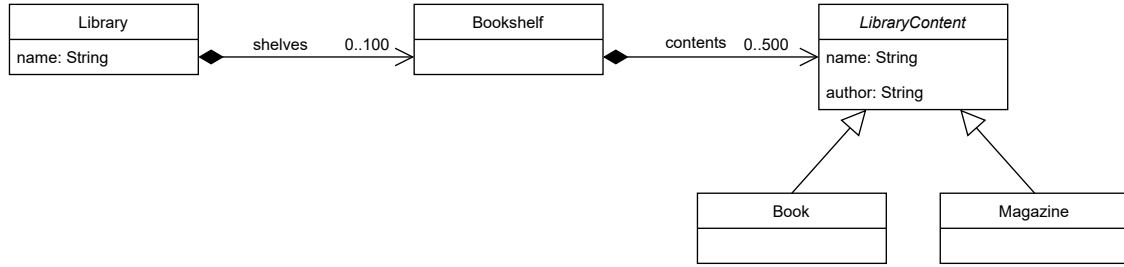


Fig. 2.2. Metamodel of the LIBRARY language expressed in UML

the constraints is to use Object Constraints Language (OCL) [15]. For example, constraints may specify that the value of the name attribute of a class must be unique across all its instances.

Fig. 2.2 shows a metamodel example of LIBRARY DSL. Here, a **Library** may contain up to one hundred **Bookshelves**, in turn containing up to five hundred **Books** or **Magazines**. The library must have a name. Also, any book and magazine are generalized in an abstract **LibraryContent** entity, which has a title and author name. Here, the abstract syntax comes with no additional constraints.

2.1.2.2. Concrete syntax. The concrete syntax of a DSL serves to define a visual representation of a model. A concrete syntax is a set of elements from which a modeler can create a model. These elements are specified by mapping each entity of a metamodel to a concrete visual representation. This means that a concrete syntax should be unambiguous. A concrete syntax can be either textual or graphical. A textual concrete syntax is defined by grammars [16], whereas a graphical concrete syntax is defined with visual shapes [17].

2.1.2.3. Semantics. The semantics of DSL defines the meaning of the language. Theoretically, the semantics is a link between the abstract syntax of the DSL and an application domain. The abstract syntax is typically defined according to the notions and rules of some domain. The semantics can "explain" the meaning of elements of the abstract syntax of a DSL according to the chosen domain. Every element of the language must have exactly one meaning.

A common task in MDE is to provide a mapping between different domains (i.e., between different semantics). The mapping is usually done using the concept of model transformations.

2.1.3. Models

A DSM is an instance of the DSL. Hence, a model has all the same properties as the DSL has: abstract syntax, concrete syntax, and semantics. The abstract syntax of a model is a physical structure that represents all elements of the model. The syntax defines how a model is represented in a memory of a computer (hence, it presents a model structure). Fig. 2.3 shows an example of an abstract syntax of **Library** model using UML object diagram. This model conforms to the metamodel defined in **Library** metamodel Fig. 2.2. In Fig. 2.3, the **Library** element has a concrete name. Also, the element has two bookshelves that contain two and three library contents (**Book** or **Magazine**) respectively. The number of library contents of each bookshelf respects the bounds of the metamodel. Since the abstract syntax of the **Library** DSL was presented using only a

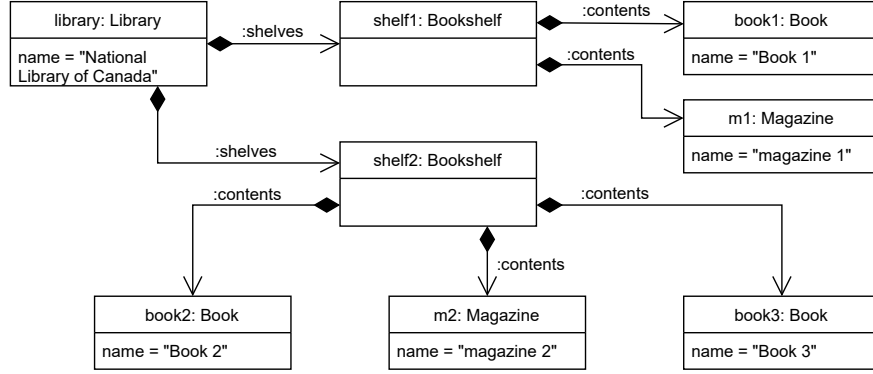


Fig. 2.3. Abstract syntax of the model instance of LIBRARY DSL

metamodel and no additional constraints were presented, the model in Fig. 2.3 is a valid instance of the Library DSL.

Fig. 2.3 represents the abstract syntax of the model. A modeler can define a concrete syntax of the model using this structure of objects. For instance, a concrete graphical colored icon per each entity. The semantics of a model (the meaning of the elements) is typically the same as the semantics of a DSL the model conforms to.

2.1.4. Metamodeling

In MDE, a DSL serves to create models, and a model must conform to its metamodel. We recall that a metamodel is a part of the abstract syntax of a DSL. Therefore, a metamodel and a model initiate two-level dependency: a DSL metamodel is an upper level which **defines** a concept, and a model is a dependent level which **uses** that concept. A metamodel is also a model that defines a possible structure (i.e., a modeling space) for its sub-models. Thus, a metamodel defines how a model can be constructed [1]. The process of development of a metamodel for a specific domain is called metamodeling.

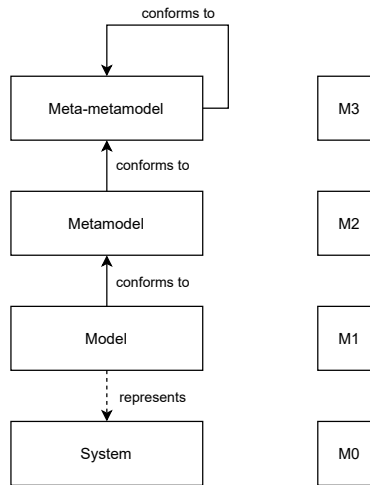


Fig. 2.4. Four-layer architecture of MDE

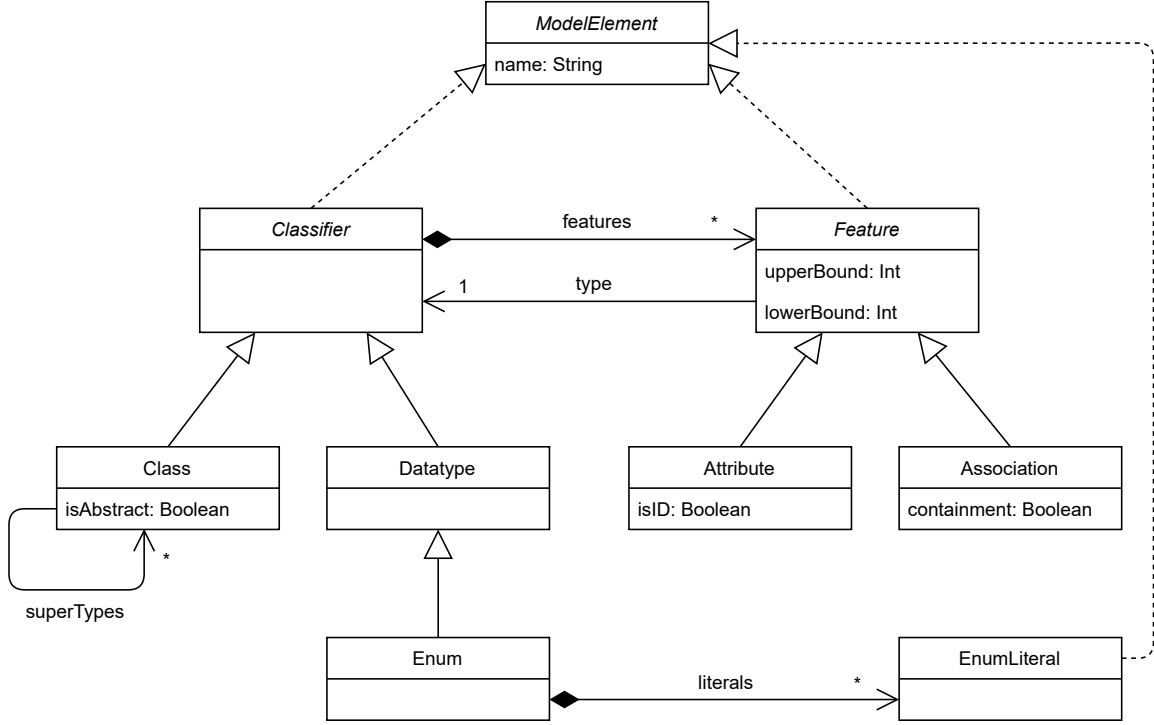


Fig. 2.5. Ecore meta-metamodel

MDE is based on the four-layer architecture defined by the Object Management Group (OMG) [1]. The architecture describes relationships between levels of abstraction that exist above the modeled system. Fig. 2.4 shows those layers. Here is a brief explanation:

- M_0 level - represents a modeled system
- M_1 level - represents a model which describes the system from M_0
- M_2 level - describes a metamodel for M_1 , a language that serves to create models at M_1 level
- M_3 layer - meta-metamodel: defines concepts for creating languages. The meta-metamodel conforms to itself, creating a cyclic dependency. This means that a meta-metamodel is defined by itself.

According to the four-layer architecture, our example of **Library** metamodel shown in Fig. 2.2 resides at the level M_2 . The model presented in Fig. 2.3 is an instance of **Library** metamodel and resides at the level M_1 . The Library metamodel is also a model, and a language is needed to define it. In our case, we use Ecore specification, which is a meta-metamodel at level M_3 [18]. We use that specification across all this thesis to express metamodels.

Fig. 2.5 shows an Ecore metamodel at M_3 . To avoid complex figures, only the relevant parts of the model are presented. Here, a **Class** instance is used to represent some entity which can be abstract. For example, a **LibraryContent** model object in Fig. 2.2 is an instance of the abstract class. Inheritance may be defined between classes (at the model instance level), which means that the inherited class has all the properties its parent has. The inheritance is specified by a **superTypes** association. The instance of the class can have **Attributes** of primitive types (integer,

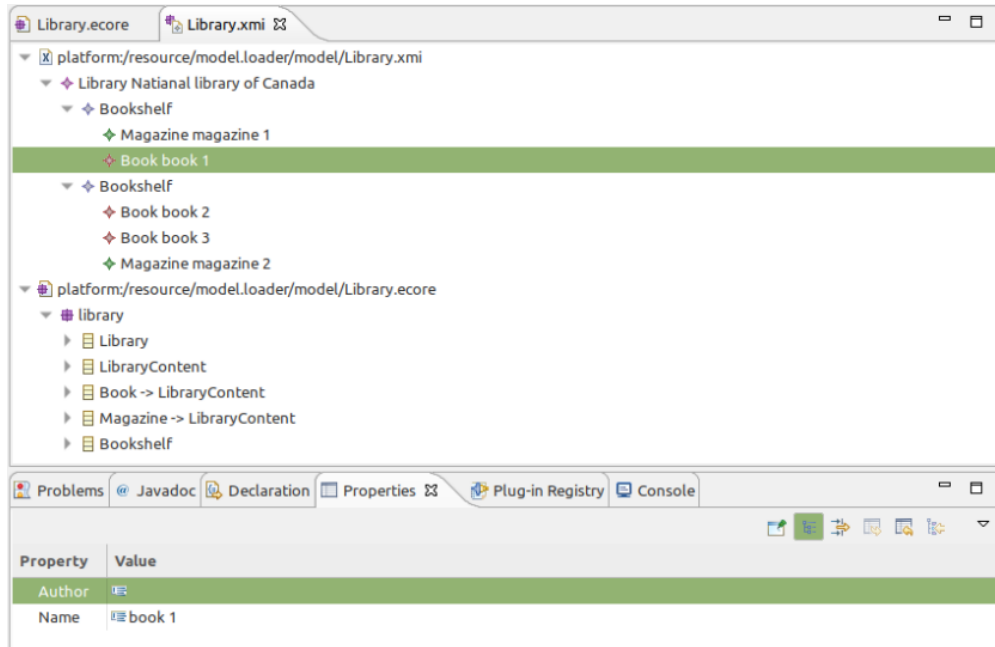


Fig. 2.6. LIBRARY model opened in EMF tree-view

string, boolean, etc.). Also, the instance can have **Associations** that point to other entities. Both attributes and associations must have **lower** and **upper** bounds – so-called cardinality. An upper bound greater than 1 means that the feature is a collection. **Association** which points to some other entity has a containment attribute. A **containment** is an equivalent of a composition pattern in UML [14]. According to the ECORE specification, any element except of a root element must have a parent, i.e., must be contained by one other element. The containment pattern has a particular impact at the model edition: removing an element leads to recursively removing all its contained elements. The **shelves** is a containment association in the LIBRARY example. This means that **Bookshelves** are contained by the **Library**. A non-containment association is called a **reference association**. It is the equivalent of a simple association in UML.

The model in Fig. 2.5 is a metamodel for itself. Each element of the model is an instance of some element of the same model. For instance, **Class**, **Attribute**, and **Association** are instances of **Class**.

2.1.5. Tools for model editing

The Eclipse Modeling Framework (EMF) is a set of tools that provide support for modeling and code generation [5]. EMF implements various generally adopted modeling specifications, so it has been accepted as a standard for modeling tools. EMF allows modeling at levels M_1 and M_2 (according to architecture presented in Fig. 2.4). At level M_3 EMF uses ECORE specification presented in Fig. 2.5. EMF is intended for use within the Eclipse IDE as a plugin. So, it relies on runtime dependencies of the Eclipse platform. The core components of EMF are: model editor (including metamodel editor), code generator, and model persistence.

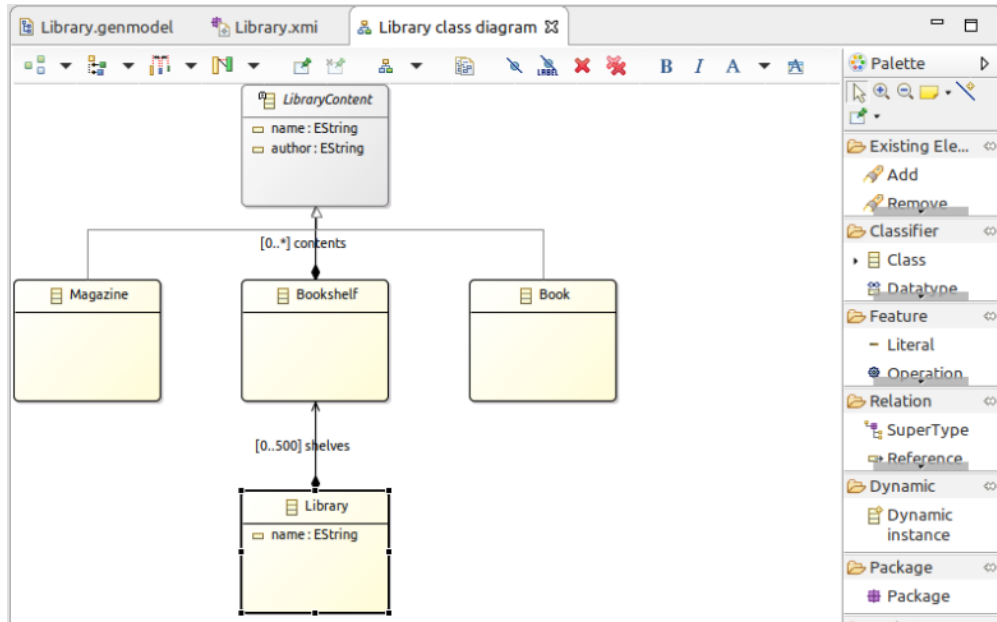


Fig. 2.7. LIBRARY metamodel opened in EMF graphical editor

The model editor component provides the ability to compose, modify, and visualize domain-specific models. The default editor provides a tree-based view. It allows to visualize a model structure (entities, attributes, and references of that entities) in terms of an abstract syntax. Fig. 2.6 shows a model from Fig. 2.3 opened in EMF tree-based view. Within the editor a modeler can modify the model elements. In addition, EMF provides graphical facilities to edit metamodels using of UML class diagram [17]. Fig. 2.7 presents an example of the LIBRARY metamodel from Fig. 2.2 opened in the EMF graphical editor. The modeler can choose the elements from the palette to construct a desired structure.

The code generator component of EMF provides support for producing a set of Java classes for a domain-specific model. The produced code represents a domain-specific API that can be used for

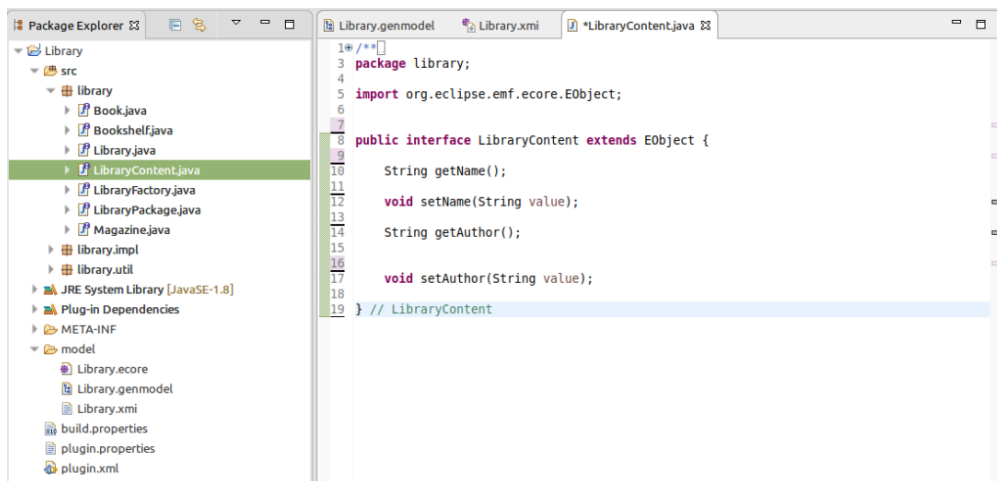


Fig. 2.8. Generated API for the LIBRARY DSL

command-based editing of the model at runtime. To produce an API, the code generator requires an abstract syntax as input. Fig. 2.8 presents the generated Java code for the LIBRARY domain.

The model persistence component provides serialization/deserialization support for models. All the created models can be exported and saved to a file in XML Metadata Interchange (XMI) format. A model can be imported later from XMI back to the EMF editor.

XMI is a model persistence format standardized by the OMG [10]. XMI is based on Extensible Markup Language (XML). It consists of a specific set of tags with a specific syntax aimed at describing a model. Listing 2.1 presents the LIBRARY model from Fig. 2.3 serialized into XMI format. The tag names for the markup come from a particular domain. Therefore, the XMI model must have a link to its metamodel (line 7) to be valid and open in the EMF editor. The containment feature is represented through XML tags embedding: **contents** (which is a name of a specific association of the metamodel) are embedded into bookshelves (lines 10–16, 19–27). All these elements are embedded into a **Library** element which is the root element of the model (line 2).

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <mylibrary:Library
3   xmi:version="2.0"
4   xmlns:xmi="http://www.omg.org/XMI"
5   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6   xmlns:mylibrary="lib"
7   xsi:schemaLocation="lib Library.ecore"
8   name="National library of Canada">
9   <shelves>
10    <contents
11      xsi:type="mylibrary:Magazine"
12      name="magazine 1" />
13    <contents
14      xsi:type="mylibrary:Book"
15      name="book 1" />
16  </shelves>
17  <shelves>
18    <contents
19      xsi:type="mylibrary:Book"
20      name="book 2" />
21    <contents
22      xsi:type="mylibrary:Book"
23      name="book 3" />
24    <contents
25      xsi:type="mylibrary:Magazine"
26      name="magazine 2" />
27  </shelves>
28 </mylibrary:Library>
```

Listing 2.1. XMI representation of LIBRARY model instance

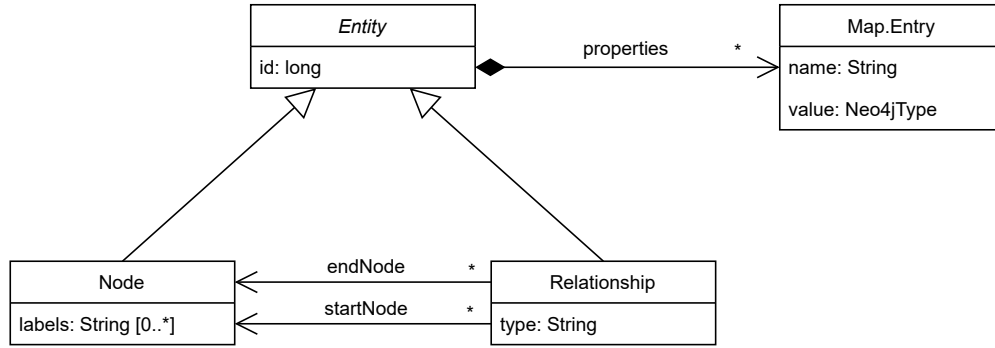


Fig. 2.9. Metamodel of a data structure in Neo4j database

All the components of EMF rely on the EMF's API. This is a core API provided in the Java programming language (since the entire EMF is written in Java). The API provides generic model manipulation operations. Internally, EMF represents a model as a set of runtime Java objects. EMF API allows CRUD the elements of that model. In addition, EMF uses this API for internal interactions between its components. For instance, EMF persistence module deserializes a model from XMI format into a set of runtime Java objects using this API.

2.2. Neo4j graph database

In this Section, we present an in-depth overview of the Neo4j database.

2.2.1. Graph structure in Neo4j database

Neo4j is a NoSQL database primarily designed to store data in graph structures [9]. Fig. 2.9 presents a metamodel of a data structure in Neo4j database. Neo4j has two types of entities: a **Node** and a **Relationship**. A Node may have zero to many **labels**. Labels are used to group nodes into sets where all nodes with a certain label belong to the same set. A relationship connects two nodes and must have exactly one **type** – the counterpart of the label for relationships. Both nodes and relationships may have **Properties**. Properties are stored as an unordered collection of key-value pairs. In properties, the key has a string type and the value may have one of following Neo4j types: Integer, Float, String, Boolean, Point – a spatial type, temporal types: Date, Time, Null, List – a heterogeneous, ordered collection of values and a Map – a heterogeneous collection of key-value pairs where the key is a String, and the value may have any type mentioned above. In Neo4j database, a Null value is handled in a special way: setting a value to null leads to removing the property from the property list.

2.2.2. Deployment

Neo4j developers offer two ways to deploy the database: as an independent remote server or as an embedded library in a Java application.

Neo4j DBMS is a software implemented in the Java programming language, so it can be included (embedded) as a dependency to a Java project. In this case, the application and the database run

on the same JVM, providing better performance. The embedded database is completely integrated into the client application and cannot be used outside it. To access the embedded database and perform read/write operations over the elements, the client must use a low-level serializing API.

The remote form of deployment requires a separate running JVM instance but has better system scalability. The remote form initiates a web-based architecture and requires a client application to manage the database. The remotely deployed database can serve multiple client applications and does not depend on them. Also, the web-based architecture gives the freedom to implement the client applications in different programming languages. To access the remotely deployed Neo4j database and perform read/write operations, the client (application) must use Cypher query language.

2.2.3. Cypher query language

Cypher is a query language with a textual concrete syntax that was designed to interact with a remote Neo4j database server. In this section, we shortly overview basic Cypher syntax.

The Cypher syntax consists of two main components: element patterns and keywords. Referring to the Fig. 2.9, we recall that Neo4j has three types of elements: a node, a relationship, and a property. Cypher provides specific textual patterns to denote these elements. We present the list of basic patterns in Listing 2.2.

```

1 { name: "someName", age: 20 } // map notation
2 [1, 2, 3] // list
3 () // a node
4 (n) // a node with an alias
5 (:Label) // a node with a label
6 (:Label1:Label2) // a node with two labels
7 (n:Label {name: "myProperty"}) // a node with an alias, label, and a property
8 ()-->() // a relationship between two nodes
9 ()-[:refType]->() // relationship with a type
10 ()-[r:refType {name: "myRef"} ]->() // a relationship with an alias, a type, and a property
11 (n)-[:refType]->(n) // a relationship pointing to the same node it starts from

```

Listing 2.2. Patterns of the Cypher query language

We start from a property pattern. To specify properties, Cypher provides a curly brackets notation `{}`. Properties are key-value pairs delimited by a comma and specified inside the curly brackets (line 1). At the same time, curly brackets denotes a map value. Properties can be presented as a part of a node pattern or a relationship pattern.

To specify a node, Cypher provides round brackets notation `()`. This is a basic node pattern that can be extended to provide additional information about the node. We can provide three parameters inside the brackets, respecting the order: an alias, labels, and properties (line 7). All of these components are optional and can be omitted, as shown on line 3. To specify an alias, we can use any literal as a first parameter (line 4). Usually, the query consists of several commands.

The alias is used to refer to the element in subsequent commands. The alias is valid only within the same query.

To specify a label, we can use any literal led by a semicolon delimiter. A semicolon is not a part of a label (line 5). A label can be specified right after an alias (if present). A node can have multiple labels (line 6). To specify properties, we can use the property pattern (described above) as the last parameter.

To specify a relationship, Cypher provides the following notation: `-[]->`. Basically, this is a directed arrow split by square brackets. Inside the brackets, we can specify three parameters: an alias, a type, and properties (line 9). The logic for these parameters is exactly the same as for nodes. The only difference is that a relationship can have only one type, while a node can have multiple labels. If we do not need to specify any additional parameter for a relationship, we can omit the square brackets and use only a directed arrow notation `->`. A relationship in Neo4j connects two nodes, so usually, we specify the node pattern at the start and the end of the relationship pattern (lines 8–11).

2.2.3.1. Create query. To create elements, Cypher provides the **CREATE** keyword. We can use this keyword in conjunction with the node pattern or the relationship pattern to specify what element to create.

```
1 CREATE ()
2 CREATE (n)
3 CREATE (:Book)
4 CREATE (b:Book {name: "myBook"})
5 CREATE (n)-[:ref]->()
```

Listing 2.3. Example of create operations of Cypher query language

Listing 2.3 demonstrates an example of creating elements with Cypher query language. It is a consistent query that contains multiple commands. If we execute this query, the database will have five nodes and one relationship. The first and second nodes are empty with no labels and properties (lines 1–2). The third node has a label **Book**. The fourth node has a label **Book** and a property **name**.

To create a node, we combine the **CREATE** keyword with a node pattern (lines 1–4). Line 2 demonstrates how to create a node with an alias **n**. The alias is valid only within the same query and can be used to refer to exactly this node (line 5). In contrast, the node created on line 1 is not accessible in this query because we did not specify an alias for it. To access exactly that node (even in the same query), we should explicitly retrieve it from the database. Line 3 demonstrates how to create a new node with a label. Line 4 illustrates how we can create a node with an alias, a label, and a property. To create a relationship, we combine **CREATE** keyword with a relationship pattern (line 5). The relationship starts from the node, created on line 2. For the target node of the relationship, we use the empty brackets pattern. This means that the target node will be created as well.

2.2.3.2. Read query. To find an existing element in the database, Cypher provides the **MATCH** keyword. We can use this keyword in conjunction with the node pattern or the relationship pattern to specify the element to find. Additionally, Cypher provides the **WHERE** keyword to filter the found elements. With this keyword we can specify more information that cannot be presented as a part of the element pattern. For example, we can filter a node by ID. To retrieve data from the database and send it back to the client, Cypher provides the **RETURN** keyword.

```
1 MATCH (n:Library)
2 WHERE ID(n) = 10
3 MATCH (n)-->(b:Book)
4 RETURN b
```

Listing 2.4. Read query

Listing 2.4 demonstrates an example of retrieving data from the database. It is a single (consistent) query that contains four commands. For the first command, we combine the **MATCH** keyword with a node pattern, providing an alias and a label (line 1). After the first command, the alias **n** represents all existing nodes with the label **Library** in the database. For the second command, we apply a filter to the nodes using the **WHERE** keyword (line 2). After the second command, the alias **n** represents only a node with a specific ID (or null if that node does not exist in the database). For the third command, we combine the **MATCH** keyword with a relationship pattern. We search for all nodes with a label **Book** that have an incoming relationship from the specific node the alias **n** represents (line 3). Also, we assign a new alias **b** to those nodes. Finally, we send back the found **Book** nodes to the client application, using the **RETURN** keyword (line 4).

2.2.3.3. Update query. To update node/relationship properties, Cypher provides the **SET** keyword. The update operation in Cypher has two syntax variants: one for updating a single property, another for updating multiple properties of an element at once. Both of these variants require an alias of the element to update. The alias in Cypher can represent a single element as well as a group of elements. Accordingly, the **SET** command updates all the elements the alias represents.

```
1 MATCH (n) WHERE ID(n) = 10
2 SET n.name = "myBook" // update single property
3 SET n.age = null      // remove property 'age'
4 MATCH (lib:Library)
5 SET lib += { capacity: 1000, test: true } // update specific properties
```

Listing 2.5. Update query

Listing 2.5 demonstrates an example of updating nodes in the database. We get a node by ID, providing an alias **n** for that node (line 1). Then, we can update a single property of the node (line 2). If the property does not exist – it will be created; otherwise, only the value will be updated. Also, we can remove a property by setting its value to **null** (line 3). Next, we get all nodes with a label **Library** (line 4). We do not apply any filter, so the alias **lib** represents multiple nodes (or null if they do not exist in the database). Finally, we update a list of properties for all nodes that alias **lib** represents (line 5). We use **+=** operator to update only the properties specified in the

curly brackets and leave unchanged other existing properties. Alternatively, we can use `=` operator to replaces all the properties with the new ones specified in the curly brackets.

2.2.3.4. Delete query. To delete elements from the database, Cypher provides the `DELETE` keyword. We can use this keyword in conjunction with an alias of an element.

```
1 MATCH (n)
2 WHERE ID(n) = 10
3 DETACH DELETE n
```

Listing 2.6. Remove node example

Listing 2.6 demonstrates an example of node removal. We search for a node with a specific ID (lines 1–2). After the line 1 an alias `n` represents all existing nodes in the database. After line 2 the alias represents only a single node (or null if it does not exist in the database). We can delete the nodes the alias `n` represents using the `DELETE n` command. But if one of the nodes we want to delete has incoming/outgoing relationships, the database will throw an exception because we did not explicitly specify the command to remove those relationships. To prevent this exception, Cypher provides the `DETACH` keyword. We use it to remove also all existing adjacent relationships of the node (line 3).

Listing 2.7 demonstrates an example of relationship removal. We search for a relationship with a specific ID (lines 1–2). After the line 1 an alias `r` represents all existing relationships in the database. After line 2 the alias represents a single relationship (or null if it does not exist in the database). We delete relationships the alias `r` represents using `DELETE` keyword (line 3).

```
1 MATCH ()-[r]->()
2 WHERE ID(r) = 10
3 DELETE r
```

Listing 2.7. Remove relationship example

2.2.3.5. Unwind query. The `UNWIND` keyword is used to transform a list data to a individual rows data. Listing 2.8 demonstrates the usage of that operation. In the presented query, we use the `UNWIND` keyword to expand a list of string literals into individual rows (line 1). Also, we provide the `row` alias to the transformed data using `AS` keyword.

```
1 UNWIND ['a', 'b', 'c'] AS row
2 RETURN row
```

Listing 2.8. Unwind query

All the remaining commands of the query will be applied for each list element independently, like inside a four-loop. Listing 2.9 demonstrate the output of the query, presented in Listing 2.8. The input list as transformed to individual list of records.

```
1 row
2 ----
3 a
```

```
4 b
5 c
```

Listing 2.9. Unwind query result

2.2.4. Neo4j query planner

Neo4j database is a highly optimized storage medium. Upon receiving a query, the Neo4j database decomposes it into operators, each of which performs a specific piece of work. Then, the operators are combined into a tree-like structure called an execution plan. Decomposing and understanding a query is a complex task. Therefore, the database keeps in a cache executions plans for the latest queries. Next time when the same query is received, the database simply finds its execution plan from the cache and omits the decomposing query process. Reusing the execution plans leads to faster query execution time. By default, the database can “memoize” the 1000 last queries.

```
1 MATCH (n:Library)
2 SET n.name = "lib1"
```

Listing 2.10. Example 1 of nodes updating

```
1 MATCH (n:Library)
2 SET n.name = "lib2"
```

Listing 2.11. Example 2 of nodes updating

Listing 2.10 demonstrates a common example of nodes updating. We search for all the nodes that have a label `Library` (line 1), and assign a new `name` property for all of them (line 2). Listing 2.11 presents exactly the same logic.

Suppose we run a query presented in Listing 2.11 after the query presented in Listing 2.10. The query planner will not reuse the execution plan in the second case because the queries are not identical. In the second query the value of the `name` property is `"lib2"`, that is different from the value in the first query `"lib1"`. Thus, Neo4j will rebuild the execution plan for the second query. We must reuse exactly the same query string to reuse the execution plan. We can achieve this by using a parameterized query.

2.2.5. Parametrized queries

We recall that the remotely deployed database instance can be accessed from a client application. Neo4j offers multiple database drivers for different programming languages. Those drivers can be integrated into a client application to access the database. In particular, the driver can send queries and receive a response from the database.

Neo4j drivers offer an important feature: they support parametrized Cypher queries. A parametrized query is a query that consists of two components: a query string and runtime parameters.

Hence, we can reuse the same query string but provide different parameters in runtime. For instance, we can optimize queries from Listings 2.10 and 2.11 replacing them by a single parametrized query (presented in Listing 2.12).

```
1 MATCH (n:Library)
2 SET n.name = $name
```

Listing 2.12. Example of a parametrized query

In Cypher, the placeholder for a parameter starts with \$ sign. We use a placeholder `$name` instead of a constant string literals (line 2). The parameter then must be supplied alongside the query string via the database driver. Listing 2.13 presents an example of sending a parameterized query in Kotlin programming language using a Neo4j Java driver [19].

```
1 val driver = GraphDatabase.driver(uri, user, password)
2 driver.session().writeTransaction { tx ->
3     tx.run(Query("...", parameters( "name", StringValue("lib1"))))
4 }
5 driver.close()
```

Listing 2.13. A call of a parametrized query from Kotlin function

After initializing the driver instance and establishing a database connection (line 1), we start a write transaction (line 2). In the transaction, we provide a query string (omitted for brevity. Assuming using the query string from Listing ??), and a runtime parameter `name` with value `"lib1"` (line 3). After the execution, we close the database connection (line 5).

The execution plan takes into account and caches only the query string. The runtime parameters do not affect the execution plan building. Thus, parameterized queries make it possible to reuse a query string and, therefore, reuse an execution plan for the same queries.

2.2.6. APOC library

APOC library is a set of common procedures and functions for the Neo4j database. This library aims to provide functionality that cannot be (easily) expressed in Cypher itself. We can invoke it directly from Cypher using the `CALL` keyword.

```
1 CALL apoc.create.node(["MyLabel"], {}) YIELD node
2 CALL apoc.create.relationship(node, node, "rerType", {}) YIELD rel
```

Listing 2.14. create elements with APOC library

Listing 2.14 demonstrates an example of creating elements with APOC library. We create a node using `apoc.create.node` function (line 1). This function takes two parameters: a list of labels (square brackets notation `[]`) and a map of properties (inside curly brackets notation `{ }`). To return the result from an APOC function we use the `YIELD` keyword. The `node` alias then is available in the remaining Cypher query. We use the same logic to create a relationship (line 2).

Both `apoc.create` function and Cypher `CREATE` keyword provide the same functionality: they create new elements in the database. But, there is an important difference. Using the Cypher

`CREATE` keyword requires to specify the node labels and relationship types (i.e., element types) as constant string literals that cannot be passed as runtime parameters (Cypher limitation). Thus, this approach imposes the creation of a new query string for each new element. In contrast, the `apoc.create` function allows to specify element types as a parameter value. Thus, we can use APOC functions to create a parameterized query and supply element types as parameters. This leads to reusing the execution plans, so leads to faster query execution time. We use this approach for batch element creation and describe it in detail in Chapter 5.

2.3. MDE in databases

We now discuss related work in MDE to store and manipulate models in databases to ensure scalability with respect to model size.

2.3.1. A model scalability

Over time, MDE models grow larger. Modeling tools have faced a new problem: there is a need to handle very large models (i.e., with more than 10^5 elements) and provide an efficient way to manipulate them. The majority of modeling tools use XMI format to store and import/export models. Unless XMI representation provides human readability, it is not an efficient way to handle huge models: the whole model must be loaded into memory and parsed before any model element can be accessed. Thus, using the most widespread XMI representation format for interchanging such big models may cause the available memory footprint to exceed. To solve the model scalability problem, the MDE community started to investigate other persistence types to store models such as databases [20]. A database management system (DBMS) is a software to organize a large amount of data and provide quick access to it. DBMS aims to efficiently support all the possible operations (i.e., create, read, update, and delete – CRUD) over the data. With these capabilities, databases may be used to store models and manipulate them in place. In this Section, we overview several optimization approaches across different persistence types to handle large MDE models. Also, we discuss some features of those approaches in relation with NMF.

First, we overview a partial XMI-based model loading mechanism [21]. The essence of that approach is to load only the needed elements but not the entire model. The loading process involves parsing the XMI markup. For that, a new XMI parser was developed since existing XMI parsers do not allow partial loading. To know in advance what exactly elements of a model should be fetched, a program that consumes the model may perform static analysis. For instance, by analyzing a metamodel (which is usually much smaller than the model instance), it is possible to find out a possible structure of a model itself. The overall presented parsing algorithm works as follows: given an element, the parser tries to find it in the model. The disadvantage of this approach is that it can only be used for read-only operations on models, since it has limitations to propagate changes to partial models. Modifying XMI data requires a strict knowledge of elements order and, hence, the overall model. Nevertheless, this approach is an interesting extension to NMF-loader (described in Sections 3.1.1 and 4.2), where we import models from XMI to the database. In this case, if the XMI model is too large, their technique can solve the scalability problem while reading the entire model.

However, we currently assume that large models reside in the database and are not serialized in XMI.

The second approach to optimize an I/O latency for a large amount of data is to use prefetching and caching techniques for the elements. Caching is the process of “remembering” an element after an application used it. Prefetching is the process of getting an element in advance before it is actually requested. These two techniques allow to reduce an access count to the actual storage and keep the desired data elements in memory with quick access. PrefetchML is a framework that successfully implements prefetching/caching techniques for MDE models [22]. The framework allows to specify the rules for prefetching and caching. Rules are triggered when a certain condition occurs. For instance, when a certain element is loaded or modified, the framework can prefetch some related elements. PrefetchML supports several persistence types for models: XMI and CDO [23], through the EMF infrastructure, and graph persistence through the Neo4EMF infrastructure [8]. In the case of using XMI, the model must be entirely loaded in memory at least for the first time before any caching, since PrefetchML relies on EMF infrastructure. This framework does not provide the automatic creation of the rules. Thus, the user should manually specify the rules according to his needs. In NMF, we use only a caching mechanism to optimize access to model elements at runtime. All the loaded elements are cached in memory by default, and a user can unload them after use.

Third, we overview a model indexing approach. Indexing is a process of assigning unique keys for a collection of data. The key can be used to rapidly access a specific element from that collection. Hence, there is no need to iterate through the entire collection and search for an element every time it is accessed. The Hawk framework successfully adapted the indexing technique for MDE models [24, 25] to provide efficient and scalable model querying. This framework was primarily designed to provide a version control system (VCS) for MDE models. Hawk keeps track of a collection of models in file-based formats (such as XMI) and maintains an index for those models into a graph database. All the tracked models must conform to the same metamodel. Hence, the index is a complex graph data structure that represents the latest version of tracked models and their metamodel. The Hawk index actually merges and replicates the contents of the tracked models into a graph database. The index can be used to efficiently query the tracked models without the need to access the actual file-based representations. Hawk updates the graph index if any tracked file-based model has been changed. Hawk handles model scalability in a special way: since the tracked models are merged into a single index, Hawk considers all the observed models as fragments of a single model. Thus, a large model can be split into multiple fragments and presented in cross-referenced XMI files. This would be another solution of the model scalability problem. In NMF, we do not support model indexing yet.

2.3.2. MDE for relational DB

A relational database stores data in a structure of related tables. Structured Query Language (SQL) is a standardized declarative programming language used to manage relational databases (i.e., tables) and perform CRUD operations on the data. SQL serves to construct queries – a set of

textual commands to manage the database contents. Object Relational Mapping (ORM) is a logical layer between a database and a concrete object-oriented programming language. ORM allows to express data management operations in terms of manipulating objects of a programming language rather than in plain-text queries. Internally, any ORM translates object commands to appropriate queries for the underlying database. Therefore, ORM augments the database management process on a higher level of abstraction since it seamlessly integrates the SQL interface into an application design and does not require a developer to deal with SQL. Nevertheless, SQL can still be used alongside ORM to express more complex queries that ORM does not support. Hibernate is an ORM for Java programming language [26]. This framework maps an object-oriented domain model presented in Java objects to a relational database and back from the database to objects. Thus, a Java object represents a specific entity (i.e., table row) in the database. A programmer can use that object to modify the property of the underlying database entity (i.e., column value) to create or search for related entities.

Next, we overview Connected Data Objects (CDO), a relational-database-based persistence framework for storing and manipulating MDE models [23]. CDO is part of EMF. Thus, it works natively with models conforming to the Ecore specification (presented in Fig. 2.5). It is possible to access a model in CDO storage directly from EMF editors. The CDO framework follows a client-server architecture where the server manages a database. Thus, the model can be stored remotely. On the client-side, CDO offers a Java API to manipulate models in a command-based way. EMF editors internally rely on this API to access the model. Also, the API can be used directly from an application code to present models as sets of CDO (Java) objects. CDO objects accessed by the client remain logically attached to their pendant in the database. Hence, the CDO API seamlessly implements the ORM technique. Since CDO uses database storage, it is possible to query certain model elements on demand. Only those elements that are loaded from the storage need to be maintained in the client application. After the elements are no longer needed, they can be unloaded to free the memory resource. This feature allows working with models larger than the memory would otherwise allow. This is the main benefit in comparison to XMI-based storage, where a model must be processed entirely before use. CDO also supports model versioning and branching. These features allow multiple modelers to collaborate on a model simultaneously.

Finally, we overview the design of relational database schema suitable for storing MDE models. The general approach, described in [7] looks as follows:

- One table per metamodel class. For instance, our Library model instance would require tables: `Library`, `Bookshelf`, `Book`, and `Magazine`.
- One-to-one and one-to-many associations are mapped using foreign keys, which requires an additional column in a table.
- Many-to-many associations are mapped to separate table in the database with two foreign keys

Unless relational-databases-based tools (such as CDO) solve the MDE model scalability problem, relational databases are not the best fit for storing the models. First, the model mapping imposes a relational database to keep additional structures, such as additional columns in a table and foreign keys for one-to-many associations, or even separate tables for many-to-many associations. Hence,

that merged data structure in the database does not provide a natural representation of MDE models.

Second, to add the data to the relational database, a table with a set of columns must be predefined. Hence, relational databases have a good vertical scaling (i.e., easily allow to add more rows to tables), but have poor horizontal scaling (i.e., require additional structure modification before adding a new column in a table or creation a new table).

Finally, relational databases data traversing and navigating through the tables ubiquitously requires a join operation. A join operation combines the output from exactly two row sources, such as tables or views, and returns a combined row source. In the case with more than two join tables, the result of the first join must be used joined with the third. Internally, the database performs a for-loop of two tables to match rows by a foreign key. In some cases, the database can optimize the lookup (for instance, to not repeat the iteration over a table). Thus, in general, the join is a relatively resource-intensive operation.

2.3.3. MDE for NoSQL DB

NoSQL, which stands for “Not Only SQL”, however is a non-relational database management system. NoSQL includes a wide range of database, such map-based, document-based (MongoDB), and graph-based (Neo4j) depending on the underlying data representation. In this thesis, we focus on graph databases.

Graph databases have several benefits over relational ones in terms of MDE needs. First, a structure of MDE models can easily be described in terms of graph concepts (i.e., in terms of vertices, associations, and attributes). Therefore, graph databases suit better for representing such kind of data. Unlike a relational database, a graph database does not require additional data structure like foreign keys to store models. This natural model structure translation is the main motivation that led us to choose a native graph database instead of another NoSQL database. Especially, we choose the Neo4j database.

Second, graph databases provide better flexibility since they do not require a predefined schema to add the data.

Finally, Neo4j graph database has no alternative of the join operation. Instead, graph nodes are linked to its neighbors directly in the storage. This type of referencing is called index-free adjacency. When Neo4j performs a graph query for retrieving a sub-graph, it simply has to follow the links in the storage – no index lookups are needed. The cost of each node-to-node traversal is $O(1)$ [9]. Therefore, NoSQL databases provide better flexibility and performance as they are not limited by the traditional relational approach to data storage [27].

2.3.3.1. Neo4EMF. Neo4EMF is an MDE model persistence framework that uses NoSQL databases as an underlying storage [8, 28]. Neo4EMF project is built on top of EMF (described in Section 2.1.5) and is available in Eclipse IDE as a plugin. Neo4EMF completes EMF project with different persistence types support, including Neo4j database that we primarily focus on.

Neo4EMF consists of two main components: model persistence and code generator. The model persistence component provides access to different databases storage. It contains connectors to multiple NoSQL databases (the full list of supported storages is enumerated in Section 7.1). Neo4EMF extends standard EMF API to make it compatible with all the supported storage.

Neo4EMF supports only embedded Neo4j deployment type. Internally, Neo4EMF relies on the Neo4j-OGM tool to interact with the database. Neo4j-OGM tool adapts the ORM technique (described in Section 2.3.2) for the Neo4j graph database. This tool uses Java objects to represent underlying graph elements in the database [29]. While working with embedded Neo4j, Neo4j-OGM does not use Cypher queries. Instead, it serializes Java objects directly to the storage. Since the embedded database is always deployed locally, Neo4j-OGM has access to the data through the API.

The code generator component of Neo4EMF provides domain-specific editing operations in the same way as EMF does.

Neo4EMF uses caching technique to optimize I/O performance. This framework does not support other optimization techniques such as data indexing or prefetching at the moment of publication of the thesis.

Neo4EMF has proven better model traversal performance in comparison to CDO (with relational database storage) and pure EMF (with XMI storage) [8]. NoSQL databases provide better scalability and performance than relational databases due to the interconnected nature of models.

Neo4EMF stores a model together with its entire metamodel in the database, explicitly connecting each model element with by a relationship with its meta-element.

Chapter 3

Neo Modeling Framework

In this chapter, we present Neo Modeling Framework (NMF) – our solution for efficient modeling in the Neo4j database. In Section 3.1 we present the overall framework architecture and components. In Section 3.2 we provide a running example.

3.1. NMF modules

Neo Modeling Framework (NMF) is a set of tools built on top of the Neo4j database. NMF is designed to achieve the following goals:

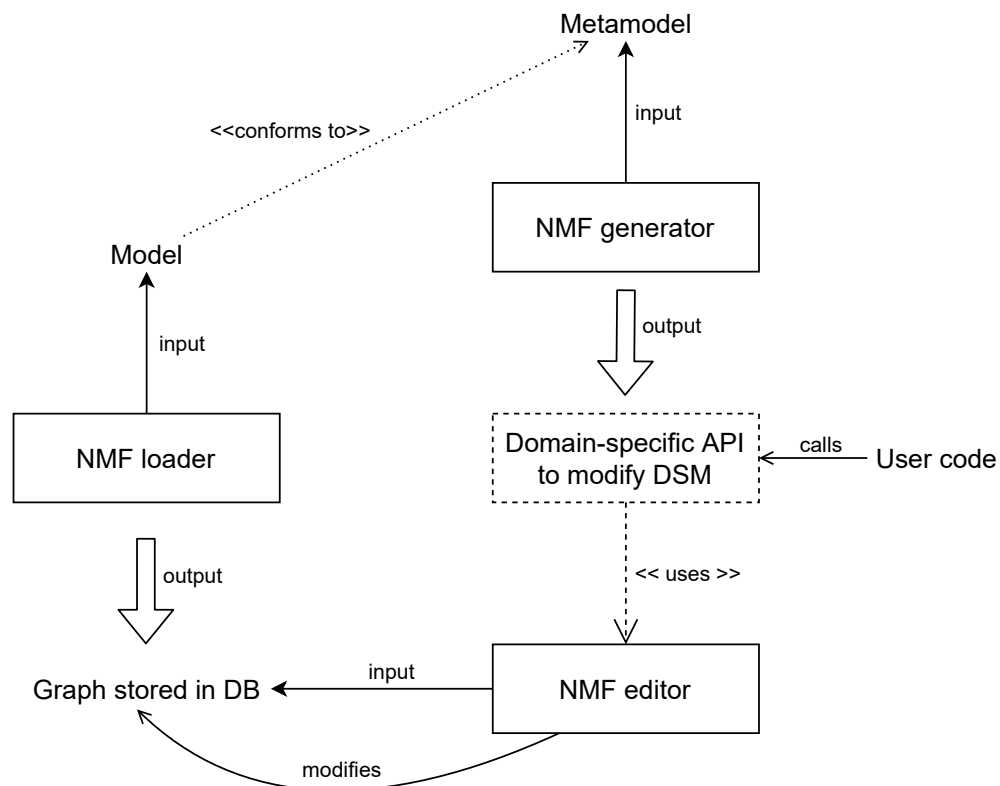


Fig. 3.1. Dependency between modules in NMF

- store a model in the Neo4j database
- manipulate (edit) models directly in the database
- provide a domain-specific API to edit any domain-specific model of the DSL

Fig. 3.1 presents the overall architecture of the Neo Modeling Framework. NMF has three main modules: NMF-loader, NMF-editor and NMF-generator. Also, NMF has a domain-specific API module that is generated from the metamodel of a given DSL.

3.1.1. NMF-loader

The purpose of NMF-loader is to store an existing model in the graph database. It takes as input a domain-specific model in Ecore format. NMF-loader also requires a metamodel for that input model to use specific types of elements and store the input model in a domain-specific manner. The metamodel must also be provided in Ecore format. NMF-loader outputs the appropriate queries to store the input model in the Neo4j database.

As an input for NMF-loader, we consider a metamodel as a model as well. Referring to the architecture of MDE shown in Fig. 2.4, the loader can have as input a model of any M_i level. If the input is a metamodel (i.e., a model at the M_2 level), it is considered an instance of the meta-metamodel of Ecore (i.e., a model at the M_3 level). In the same way, an Ecore meta-metamodel itself can be passed as an input and stored as such.

3.1.2. NMF-editor

The purpose of NMF-editor is to provide a generic API to interact with a graph directly in the database. This approach does not require the system to keep entire models in memory and offers additional scalability. The generic API is independent of any metamodel and provides general CRUD operations that do not tailor to a specific domain. NMF-editor considers all the graph elements as instances of the metamodel presented in Fig. 2.9.

The API of NMF-editor can create a model from scratch or operate on a model previously stored by NMF-loader.

3.1.3. NMF-generator

NMF-generator takes a metamodel of a DSL as an input and synthesizes a domain-specific API using the model-to-text transformation principle. The input metamodel must be in Ecore format. The produced API can manipulate any DSM conforming to the input metamodel and stored in the Neo4j database. The generated API relies on NMF-editor to interact with a graph in the database. The API aims to provide specific operations tailored to the metamodel of the DSL. This keeps the semantics of a model unchanged and allows a user to continue working in a familiar domain. For instance, for our example of the **Library** metamodel presented in Fig. 2.2, to create a new **Book** element, the generated domain-specific API will have the `createBook` method. Instead, using a generic API would imply calling the `createNode` method from NMF-editor and pass the type **Book** as parameter.

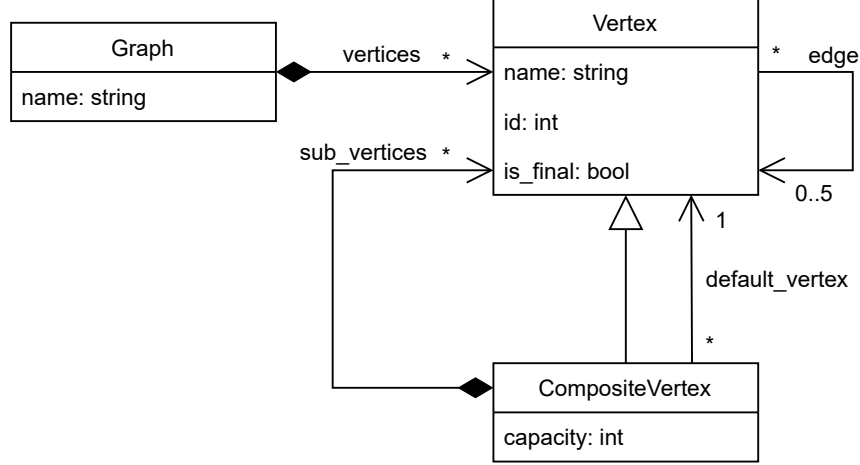


Fig. 3.2. The metamodel of the GRAF DSL in UML class diagrams

3.2. Running example

In this section, we present the GRAF DSL as a running example that will be used in the following chapters to present the functionality of NMF modules.

3.2.1. Graf DSL

Fig. 3.2 shows a metamodel of our GRAF DSL. This metamodel covers all possible spectrum of elements needed in a metamodel, such as classes, attributes, compositions, associations, inheritance, and cardinality constraints. Here we have a **Graph** element, which is the root object of any GRAF model. The **Graph** can consist of multiple **Vertex** elements. A vertex has a name, a globally unique identifier, and can be marked as final. A special kind of vertex is **CompositeVertex** that represents hierarchical vertices containing other vertices. Vertices can have multiple **edges** pointing to other vertices. A **CompositeVertex** has a mandatory reference to one of its subvertices to mark it as **default_vertex**. Also, the **CompositeVertex** has a **capacity** attribute that limits the maximum number of subvertices it can contain.

Fig. 3.3a shows a possible example of a model instance of the GRAF DSL. The model instance has a container **Graph** element with a name G_1 . Graph element has one **CompositeVertex** element V_0 . **CompositeVertex** contains two **Vertices** V_1 and V_2 . V_1 is a default vertex for the **CompositeVertex** element. Also, the vertex V_1 has one outgoing edge pointing to the vertex V_2 .

3.2.2. Storing models and metamodels in the database

Suppose we want to store the model shown in Fig. 3.3a in the database. The input model is an instance of our GRAF DSL. For the NMF-loader, we provide that input model in Ecore format. Fig. 3.3b shows how it is represented after NMF-loader stores it in the database. The figure is output from the Neo4j-browser, a standard tool that is usually installed alongside the Neo4j database. This tool offers the most convenient way for a user to inspect and visualize the contents of the database.

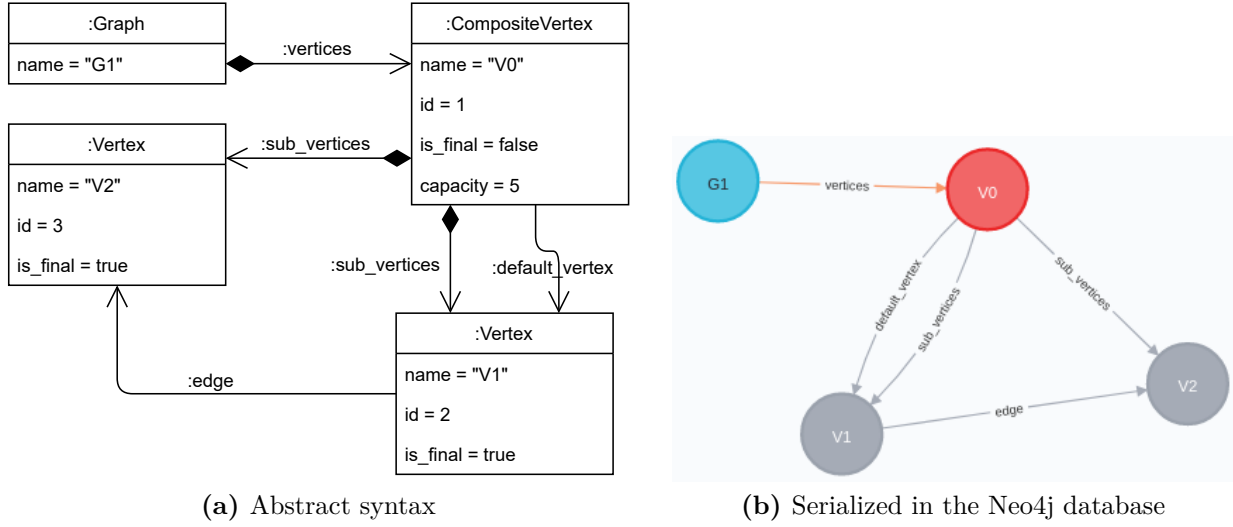


Fig. 3.3. The model instance of the GRAF DSL



Fig. 3.4. The metamodel of the GRAF DSL serialized in the Neo4j database

Now suppose we want to store the metamodel itself of our GRAF DSL shown in Fig. 3.2 in the database. For NMF-loader, we provide the input metamodel in Ecore format as well. Fig. 3.4 shows how this second model is stored with NMF-loader. The metamodel is stored as a model which conforms to the Ecore meta-metamodel presented in Fig. 2.5.

In fact, the model in Fig. 3.2 and the model in Fig. 3.4 are equivalents. The difference is that Fig. 3.2 shows the model using the UML class diagram, and Fig. 3.4 represents that model serialized in the Neo4j database. The same thing for model instances of our GRAF DSL: Fig. 3.3b represents the serialized version of the model presented in Fig. 3.3a. Both of these models have the same abstract syntax, presented in Fig. 3.3a.

3.2.3. Generic manipulation of models

Instead of loading it, suppose we want to create from scratch the model presented in Fig. 3.3a using the generic NMF-editor API. Listing 3.1 demonstrates our running example. We have implemented NMF in the Kotlin programming language, so the API is provided for that language.

Listing 3.1 presents a user code that uses the generic NMF-editor API. After establishing a connection with a database (line 1), we create a `graphController` object, that gives us access to the (empty) graph. As we can see on line 2, we can create a new node. We can then use that node to create new subnodes (line 3–5) or outgoing references (lines 7–8). Also, we can modify the properties (i.e., attribute values) of that node (line 6). After all the manipulations, we commit the changes to the database (line 9) and close the connection (line 10). The `saveChanges` method propagates accumulated changes to the database and can be called at any time. The resulting graph in the database is exactly the same as if we used NMF-loader, presented in Fig. 3.3b.

```
1  val graphController = GraphController(dbUri, username, password) //init a connection to the DB
2  val graph = graphController.createNode("Graph")
3  val v0 = graph.createChild("vertices", "CompositeVertex") // hard-code the type of object to
    create
4  val v1 = v0.createChild("sub_vertices", "Vertex")
5  val v2 = v0.createChild("sub_vertices", "Vertex")
6  graph.putProperty("name", "G1")
7  v1.createOutRef("edge", v2) // hard-code the type of reference to create
8  v0.createOutRef("default_vertex", v1)
9  graphController.saveChanges() // commit changes to the database
10 graphController.close() // close the connection to the DB
```

Listing 3.1. Using the generic model editor

Since the generic API is domain-independent, it implies providing all the types and element names explicitly. They should be hardcoded in strings manually by a user. Furthermore, there are no constraint checks performed by NMF-editor with respect to the DSL. For example, it is possible to also create a `default_vertex` from `v0` to `v1`, which contradicts the static semantics of the GRAF DSL. To overcome these issues, we use NMF-generator.

3.2.4. Domain-specific manipulation of models

First, we generate a domain-specific API for the GRAF DSL with NMF-generator. We use the metamodel presented in Fig. 3.2 as an input, providing it in Ecore format (see Appendix B). Listing 3.2 presents the generated API for the GRAF DSL. We implemented all of NMF in the Kotlin programming, which is fully compatible with JVM and can be called from Java programming language if needed.

NMF-generator generates a separate Kotlin class along with an interface for each class of the input metamodel. Listing 3.2 presents only the generated interfaces for `Graph`, `Vertex` and `CompositeVertex` entities. Each interface has getters and setters for each attribute. Setting a

value performs a write operation to the database, and getting a value performs a read operation from the database. The generated interfaces contain methods for getting, setting, and removing objects connected by a composition or an association reference. The setters and removal methods also satisfy all the cardinality constraints of references and attributes.

Finally, the generated code contains an enumeration of concrete classes (i.e., not abstract classes and not interfaces) that are part of an inheritance hierarchy (line 29). The setter method for a containment reference returns a new generalized **Vertex** object (line 6). The generated enumeration can be used as a setter parameter to specify the concrete implementation among several possible subclasses (lines 6, 27). Internally, the API will instantiate and return the appropriate concrete class.

```

1 interface Graph : INodeEntity {
2     fun setName(v: String?)
3     fun getName(): String?
4     fun removeVertices(v: Vertex)
5     fun getVertices(limit: Int = 100): List<Vertex>
6     fun addVertices(type: VertexType): Vertex
7 }
8 interface Vertex : INodeEntity {
9     fun setName(v: String?)
10    fun getName(): String?
11    fun setId(v: Int?)
12    fun getId(): Int?
13    fun setIs_initial(v: Boolean?)
14    fun getIs_initial(): Boolean?
15    fun setEdge(v: Vertex)
16    fun removeEdge(v: Vertex)
17    fun getEdge(limit: Int = 100): List<Vertex>
18 }
19 interface CompositeVertex : Vertex {
20    fun setCapacity(v: Int?)
21    fun getCapacity(): Int?
22    fun setDefault_vertex(v: Vertex)
23    fun removeDefault_vertex(v: Vertex)
24    fun getDefault_vertex(): Vertex?
25    fun removeSub_vertices(v: Vertex)
26    fun getSub_vertices(limit: Int = 100): List<Vertex>
27    fun addSub_vertices(type: VertexType): Vertex
28 }
29 enum class VertexType { CompositeVertex, Vertex }

```

Listing 3.2. Genarated API for the GRAF DSL

The generated API offers a convenient way to edit models directly in the database. Unlike in the generic API, there is no need to specify names and types for each node explicitly. The generated code conforms to the static semantics of a the DSL.

Let us now recreate from scratch the model presented in Fig. 3.3a using the generated domain-specific API presented in Listing 3.2.

Listing 3.3 creates the same model as Listing 3.1. After establishing a connection with a database (line 1), we have a domain-specific model `manager` object that gives us access to the graph. We can create concrete domain classes such as `Graph` using that manager (line 2). We assign a name to the `Graph` object using the appropriate `setName` setter method (line 3). Line 4 demonstrates how we can create a new `CompositeVertex` as a containment of the `Graph` object. We specify the `VertexType` parameter for the method `addVertices` of the `Graph` object. The method `addVertices` returns a generalized `Vertex` object, so we instantly cast it to the `CompositeVertex` as it is. Next, we create two containments for the `CompositeVertex` (lines 5–6). Also, we create a `default_vertex` association (line 7). Finally, we commit all pending changes to the database (line 8) and close the connection to the database (line 9).

```
1  val manager = GrafModelManager(dbUri, username, password)
2  val graph = manager.createGraph()
3  graph.setName("G1")
4  val v0 = graph.addVertices(VertexType.CompositeVertex) as CompositeVertex
5  val v1 = v0.addSub_vertices(VertexType.Vertex)
6  val v2 = v0.addSub_vertices(VertexType.Vertex)
7  v0.setDefault_vertex(v1)
8  manager.saveChanges()
9  manager.close()
```

Listing 3.3. Usage of the generated API for the Graf DSL

In this chapter, we have presented the architecture of NMF and the basic functionality of the framework modules. In the following chapters, we present the internal implementation of each module in detail.

Chapter 4

NMF-Loader

This chapter describes NMF-loader module of NMF. In Section 4.1 we present the data mapping mechanism used by NMF-loader. Then, in Section 4.2 we present the architecture of the module.

4.1. Data Mapping

Unlike a relational database, the Neo4j graph database is a storage medium that does not require a strict data schema. However, to efficiently reason and manipulate models in a graph database, we shall provide a consistent way to encode models in Neo4j. In this section, we describe the formal mapping from an Ecore model to a Neo4j graph.

To describe the mapping from Ecore model to Neo4j graph, we formally represent each domain. We consider a Neo4j graph as a structure $NGraph = \langle N, R, l, t, prop_N, prop_R \rangle$, where N are nodes and $R \subseteq N \times N$ are relationships. Nodes are labeled by a string $l : N \rightarrow \Sigma^*$ with an alphabet Σ . Similarly, relationships are typed with the function $t : R \rightarrow \Sigma^*$. Nodes can hold properties $prop_N : N \rightarrow \mathcal{P}(\Sigma^* \times V)$ in the form of key/value pairs, where V is a set of values of any primitive type (string, integer, boolean, etc.). Similarly, $prop_R : R \rightarrow \mathcal{P}(\Sigma^* \times V)$ defines the properties of relationships. This abstraction of $NGraph$ corresponds to the representation of graphs in a Neo4j as presented in Fig. 2.9.

We consider an Ecore model as a structure $EModel = \langle O, attr, ref, cont, t_o, t_r, t_c \rangle$, where O is the set of objects in the model. Objects can hold a set of attributes $attr : O \rightarrow \mathcal{P}(\Sigma^* \times V)$ in the form of key/value pairs. Objects can be related by associations, which can be either containments or references $cont, ref \subseteq O \times O$. Objects, containment references, and association references are typed by a string with the functions $t_o : O \rightarrow \Sigma^*$ and $t_c, t_r : O \times O \rightarrow \Sigma^*$. This abstraction of $EModel$ corresponds to the representation of models in Ecore as presented in Fig. 2.5.

Given an Ecore model $EModel$, we construct a Neo4j graph $NGraph$ by a mapping $M : EModel \rightarrow NGraph$ as follows:

$$N = \{M(o), \forall o \in O\} \quad (4.1.1)$$

$$\forall n \in N, l(n) = t_o(o) \text{ where } n = M(o) \quad (4.1.2)$$

$$\forall n \in N, prop_N(M(o)) = M(attr(o)) \text{ where } n = M(o) \quad (4.1.3)$$

$$R = \{(M(o_1), M(o_2)) \mid o_1, o_2 \in O \wedge ((o_1, o_2) \in ref \vee (o_1, o_2) \in cont)\} \quad (4.1.4)$$

$$\forall r \in R, prop_R(r) = \begin{cases} (\text{"containment"}, true) & \text{if } (o_1, o_2) \in cont \\ & \text{where } (M(o_1), M(o_2)) \in r \\ (\text{"containment"}, false) & \text{if } (o_1, o_2) \in ref \\ & \text{where } (M(o_1), M(o_2)) \in r \end{cases} \quad (4.1.5)$$

$$\forall r \in R, t(r) = \begin{cases} t_c((o_1, o_2)) & \text{if } (o_1, o_2) \in cont \text{ where } (M(o_1), M(o_2)) \in r \\ t_r((o_1, o_2)) & \text{if } (o_1, o_2) \in ref \text{ where } (M(o_1), M(o_2)) \in r \end{cases} \quad (4.1.6)$$

In this transformation, we represent objects as nodes in Equation (4.1.1). The label of a node represents the type of its object as defined in Equation (4.1.2). In Equation (4.1.3), we represent the attributes of an object as properties of the nodes. Containments and references are represented as relationships in Equation (4.1.4). To distinguish between them, we define a property “containment” in all relationships as defined in Equation (4.1.5). Finally, the type of the containment or reference is represented as the type of the relationship Equation (4.1.6).

We have chosen this particular representation of an MDE model in the database because we focus on providing the best read/write operations performance. With our representation, we can easily query the graph using the Cypher language. Moreover, the presented mapping approach provides a natural representation of MDE models. Unlike in relational databases, it does not require any additional structures to ensure the semantics of MDE models.

In general, our mapping approach is based on the model instantiation principle. Referring to Fig. 2.4, a model M_i is an instance of a model M_{i+1} . In our approach, a graph in the database represents a model at level M_i , where each element of the graph has a logical link with its meta-element (i.e., with a corresponding element at level M_{i+1}). Indeed, we need to keep a logical link between a model instance and its metamodel to use the stored model in the MDE terms. To provide the link, we set a label of a node and a type of a relationship to a name of a meta-class of the element. Therefore, in $NGraph$, there is exactly one label per node.

In NMF, we have a different model mapping approach compared to Neo4EMF. The latter keeps an entire metamodel alongside a model instance in the database, explicitly connecting each element of the model by a relationship with its meta-element [8]. Therefore, when the number of model elements grows, the number of the relationships to the meta-elements also grows. In NMF, we do not overcharge the database, storing only relevant model elements there. At the same time, we keep the MDE logic outside of the database – on a client-side in a generated API. To check the conformance of a model or the type of a specific model element, in NMF we need a “join”, but

in Neo4EMF we have it directly. So Neo4EMF optimizes conformance checking and the storage schema while NMF optimizes model manipulations.

4.1.1. Multilevel mapping

We do not distinguish the mapping approach for a model and a metamodel. Referring to Fig. 2.4, we use the same mapping for a model at any meta-level. The only essential requirement of our mapping approach is that the input model must come with its metamodel. We use the metamodel to determine the types of elements presented in the input model correctly.

An input model of level M_2 conforms to the model of level M_3 , presented in Fig. 2.5. For the input models of level M_2 , we rely on the metamodel presented in Fig. 2.5 to perform the mapping. In case when the input model is a model of level M_3 , we use the same model as its metamodel because it conforms to itself.

An important thing to note is that according to our mapping approach, attributes and references of models of M_2 and M_3 levels are represented as separate nodes because they are defined as individual classes in the metamodel. Fig. 3.4 illustrates a serialized version of a model at level M_2 , presented in Fig. 3.2. Here, the green nodes represent attributes, and the blue nodes represent associations of the model.

4.1.2. Mapping example

Fig. 3.3a demonstrates the representation of the GRAF DSL model instance in the Neo4j database after applying our mapping approach. This figure also shows an enhanced abstract syntax of the model. The only difference between our model representation and the model's abstract syntax is that the associations in the pure abstract syntax cannot have any attributes. In our case, we use the power of Neo4j to set an additional property for each relationship to distinguish containments and non-containments. That is what we call a natural representation of an MDE model in a graph structure: a serialized model looks exactly like its abstract syntax.

Fig. 3.3a shows a model with four objects: a **Graph**, a **CompositeVertex**, and two **Vertex** objects. They are represented as separate nodes in the database. Each node has a label: the name of the meta-class of the object. A label appears at the top of each object representation after the semicolon (semicolon is not a part of the label). The **Graph** object has a **name** attribute that handles a string value **G1**. After the mapping, the **Graph** node has a property with the same name and the corresponding value.

The **CompositeVertex** object contains two **Vertex** objects. Since **Vertex** objects are already presented as individual nodes, we map the containment associations of the **CompositeVertex** object to outgoing relationships of the **CompositeVertex** node, each pointing to the corresponding **Vertex** node in the database. Both relationships have a type **sub_vertices** that corresponds to the name of the meta-association in Fig. 3.2. Also, both relationships have a boolean property **containment** set to true, highlighting that this is a containment association. Similarly, an outgoing association of type **default_vertex** of the **CompositeVertex** is mapped to the relationship with the same type and has a property **containment** set to false.

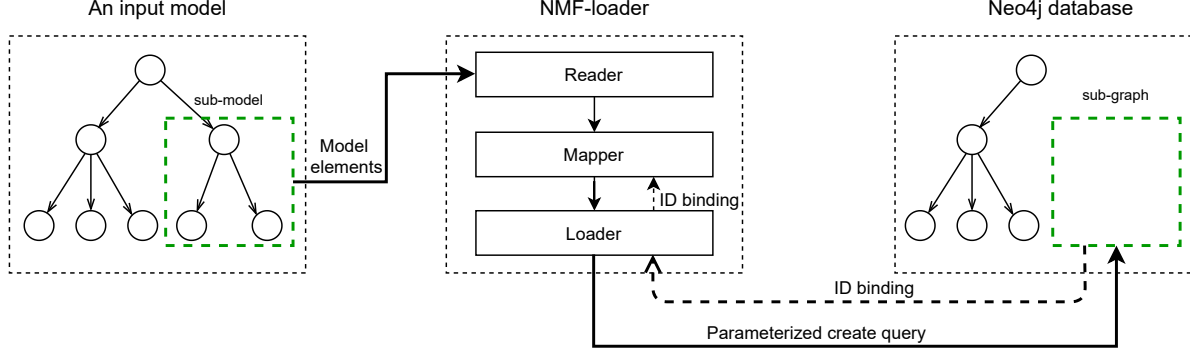


Fig. 4.1. NMF-loader data flow

4.2. NMF-loader components

NMF-loader is a tool for importing existing MDE models into the Neo4j database. NMF-loader offers the following features:

- store in the database any Ecore model at any M_i meta-level
- provide appropriate data mapping, as described in Section 4.1
- interact with a remote database instance
- process large input models by splitting them into smaller partial models
 - gradually upload partial models to the database in a transactional way;
 - keep consistency between partial models.

To provide these features, we have designed a pipeline architecture, presented in the center of the Fig. 4.1. NMF-loader consists of three components: reader, mapper, and loader. These components form a chain of handlers, which consequently process the elements of an input model.

4.2.1. Reader

The reader is the first component that traverses the input model and prepares data for the other components in the processing chain of NMF-loader. Fig. 4.2 presents the structure of the reader component. The reader consists of two sub-components: element extractor and a buffer.

4.2.1.1. Extracting the elements from an input model. The extractor sub-component is responsible for iterating and extracting the elements of an input model. The extracting process implies selecting and deserialization of the elements.

From a user point of view, NMF-loader treats the input models of any M_i level in the same way. But internally, we use different extraction strategies for models of level M_1 from those at levels M_2 and M_3 .

We use a reflective element extractor for input models of M_1 level. The key concept of this extractor is that it selects all the model elements in a reflective way. This makes the reader domain-independent meaning that a user can pass any model instance of any DSL as the input. The reflective extractor automatically inspects the meta-types of the model elements to determine

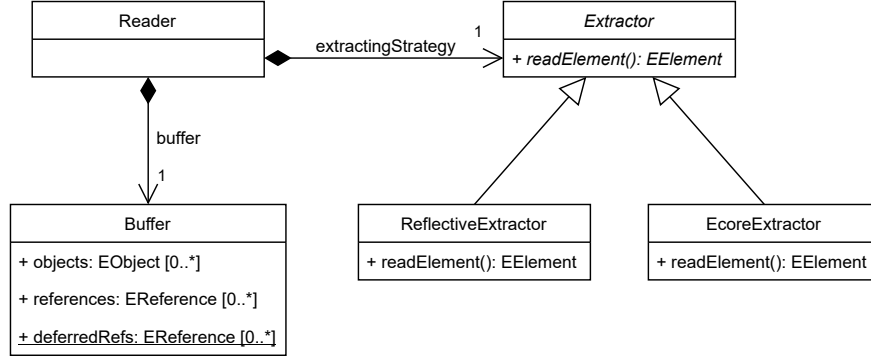


Fig. 4.2. The architecture of the reader component of NMF-loader

the type of each object and association. This strategy works independently from the metamodel as it dynamically retrieves the meta-types.

For metamodels (i.e., for input models of levels M_2 or higher) we use an Ecore-specific element extractor. This extractor selects only the relevant part of an input model. In fact, the metamodel provided in Ecore format contains the model itself and a piece of serialization information. If we used the reflective extractor to traverse a metamodel, the output graph would have many unneeded details, relevant to the specific Ecore implementation. The Ecore extractor selects only the essential part of the model that we are interested in, omitting unnecessary elements.

To choose the correct extractor, the reader component performs a preprocessing step: it detects a type (i.e., a level) of an input model. *EModels* have a structure of a tree with a single root element. The preprocessor analyses the type of that root element. Metamodels in Ecore format have a root element of type `EPackage`. For an input model with that root we use Ecore extractor; otherwise, NMF-loader uses reflective extractor.

4.2.1.2. Buffering. The reader puts to the buffer the elements selected by the extractor sub-component. The buffer accumulates the elements and organizes them into batches. The buffer has a fixed size, and once it is filled, the buffered elements are transmitted to the mapper. The reader refills the buffer as long as the input model has the remaining elements to process.

With this approach, we can handle large input models, splitting them into sub-models and processing them successively in multiple iterations. Fig. 4.1 demonstrates one iteration of the loading process. The dashed rectangle on the left side of the figure represents a sub-model stored in the buffer. The green rectangle on the right side of the Figure represents the processed sub-model. The other components of NMF-loader (mapper and loader) process not an entire input model, but a sub-model from the buffer. This is extremely important for the loader component, where the connection to a database accepts a limited amount of data in one transaction. In the special case when the input model can entirely fit within the buffer, NMF-loader processes it in one iteration.

In the loader component, the loading of the elements to the database consists of two steps: the first one is to load nodes, the second one – to load relationships. This distinction exists because of the following fact: to create a relationship between nodes in the Neo4j database, we must provide the source and target node for the relationship. Therefore, we create a relationship only between

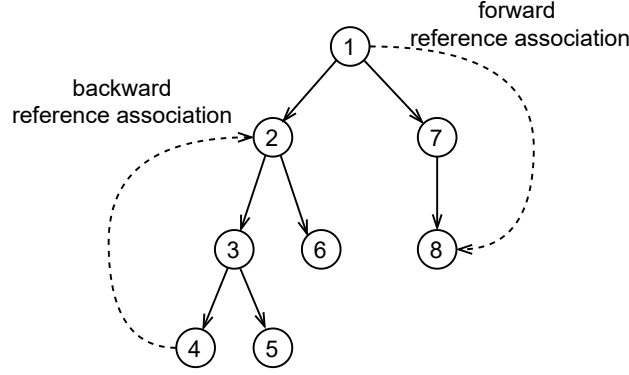


Fig. 4.3. *EModel* traversal process

existing nodes in the database. For each sub-model, the loader component uploads nodes and relationships in two different queries, where nodes come before relationships.

In the buffer, we start to form correct data structures for the other components of the processing chain. Thus, we split associations and objects of the sub-model and hold them separately in the buffer, so the loader component could use them independently. As shown in Fig. 4.2, the buffer has separate containers to hold objects, associations, and so-called deferred associations. Each of these containers have a size. If one of the containers is filled, we consider the entire buffer as filled and passes the buffered element to the mapper component.

The associations container must contain only those associations whose objects are in the objects container of the buffer, or are already in the database. This filtering helps to avoid inconsistency for loading associations to the *NGraph*, and ensures that the adjacent nodes of relationships are already presented in the database. The other associations, that have unprocessed source or target object, are added to the deferred associations container of the buffer.

4.2.1.3. Reading process. The reader iterates over an input model, selects the elements within the extractor, and puts them into the buffer. While reading the input model we must correctly fill the elements into the buffer, respecting its three containers: for nodes, for associations, and for deferred associations.

The structure of any *EModel* has some particular features:

- *EModels* have a tree structure with a single root object
- each object of *EModel*, except the root object, has exactly one incoming containment association

We use these features to efficiently iterate over an input model. Fig. 4.3 demonstrates a common structure of *EModel*. In this figure, the circle specifies an object, the solid arrow specifies a containment association, and a dashed arrow denotes a reference association. Also, we have specified the reading order for each object in the figure. We use a depth-first traversal to iterate over the model elements.

We start to traverse *EModel* from its root object. The reader immediately puts the root object to the objects container of the buffer. To move between objects we use only a containment

association. For each next object of the model (except of the root object) we perform the following procedure:

- (1) put the object (including its type and attributes) to the nodes container of the buffer
- (2) put the incoming containment association to the associations container of the buffer (skip this step for the root object)
- (3) check if either the objects or the associations container of the buffer is filled. If so – pass the buffered elements to the mapper component
- (4) for each outgoing reference association:
 - (a) check if the buffer contains the end object (an object the current association points to). If so, put the reference association to the association container of the buffer, otherwise put it to the deferred associations container of the buffer.
 - (b) check if the associations container of the buffer is filled. If so, pass the buffered elements to the mapper component
- (5) check if the deferred associations container has associations, that points to the current object. If it has, get them and perform for each:
 - (a) transfer the reference association from the deferred associations to the associations container of the buffer
 - (b) check if the association container of the buffer is filled. If so, pass the buffered elements to the mapper component
- (6) for each outgoing containment association:
 - (a) get the contained object and repeat for it all operations starting from the step 1. (This operation ensures the depth-first traversal)

The model presented in Fig. 4.3 has two reference associations. The forward reference association (from *object*₁ to *object*₈) is an association, that points to the object, that is not yet reached by the reader. Therefore, the reader puts that association to the deferred associations container. A backward reference association (from *object*₄ to *object*₂) points to the object that was already processed by the reader. Thus, the reader always puts that association to associations container of the buffer. This is a benefit of a depth-first traversal: we are able to include the backward associations directly to the associations container of the buffer. The deferred associations container of the buffer acts as a temporary holder for reference associations. On the last iteration, that container becomes empty according to the step 5 of the reading procedure.

We process containment associations as an incoming association of each object (assuming that it always has exactly one incoming containment association). Thus, the object and its incoming containment association are passed to the buffer, that provides the best sub-model consistency.

4.2.2. Mapper

The mapper component provides a two-way data mapping. First, the mapper takes the sub-model from the buffer and maps its elements according to the mapping rules defined in Section 4.1. The mapper preserves the input data structure by passing separate lists of nodes and relationships

to the loader. Second, the mapper component binds the IDs from the database to objects in the input model.

The loader component creates the nodes and relationships in two different transactions. The first transaction is to create nodes and the second to create relationships. To create a relationship, the loader must use existing nodes. To use an existing node in the second transaction, we must first match it. Considering that we work with a remote database instance, there are two ways to match a previously persisted node:

- add a custom parameter for each node. In this case, a client application manages these parameters and uses them for further node search;
- use internal database IDs that are automatically assigned to each node on creating. To rely on these internal IDs, we must take them back to the client application from the database.

We use a second approach relying on internal IDs because searching a node by ID is the most efficient way in Neo4j. For that, the loader component gets the IDs of nodes and passes them back to the mapper component, as shown in Fig. 4.1.

We may achieve the situation when the source and target objects of the association are processed in different iterations (sub-models). To correctly create the relationship in the database, we must know the IDs of both nodes. To be able to get the node ID from past iterations, the mapper component holds an additional cache map. This map holds an *EModel* object as a key and a corresponding ID from the database as a value. We put in this map the objects that have unprocessed outgoing associations, and we hold them there until the last outgoing association is stored in the database (so until the loader component does not need it). Managing this map (adding, getting, and removing IDs to it) is called ID binding.

4.2.3. Loader

The loader component provides the interaction with a remote database instance. It takes the elements from the mapper component and uploads them to the database.

In the loader component, we use parametrized Cypher queries. We use only two different queries: one to load nodes, the second to load relationships only. This approach takes into account the query planner of Neo4j database, described in Section 2.2.4. For that, we have different containers for nodes and relationships starting from the buffer component.

To create the nodes in the database, we use a Cypher query, presented in Listing 4.1. This is a parameterized query that takes a list of nodes to create as a parameter. This list comes from the mapper component. Each element in the list has the following structure:

- **alias** – a custom parameter that serves to identify the node after we get its ID from the database. The **alias** helps to perform ID binding
- **label** – string literal
- **properties** – a map of properties where the key is a property name and the value is one of supported Neo4j types

```
1 UNWIND $batch AS row
2 CALL apoc.create.node([row.label], row.properties) YIELD node
```

```
3 RETURN row.alias AS alias, ID(node) AS id
```

Listing 4.1. Batch create nodes query

Listing 4.1 consists of three commands. First, we transform the input list of parameters into individual rows using the `UNWIND` keyword (line 1). After that, each following command will be executed as many times as the size of the input list is. The logic of `UNWIND` command is very similar to a simple `for-loop`. The `row` variable denotes an individual concrete parameter on each iteration. Next, we use the `apoc.create.node` function of the APOC library to create a node from the parameters (line 2). Here, we cannot use the simple `CREATE ()` command because it does not support parameters. We should explicitly define labels and properties in the `CREATE` command. Finally, we return back to the loader component the IDs of newly created nodes, mapped to the `alias` parameter (line 3). This parameter allows to determine on the loader side which ID corresponds to which node. The loader then passes the response of that query back to the mapper component to let him perform the ID bindings for nodes.

To create the relationships in the database we use a Cypher query, presented in Listing 4.2. This is a parametrized query that takes a list of relationships to create as a parameter. Each element in the list has the following structure:

- `type` – string literal
- `from` – the ID of the source node
- `to` – the ID of the target node
- `isContainment` – a boolean true or false

```
1 UNWIND $batch AS row
2 MATCH (from) WHERE ID(from) = row.from
3 MATCH (to) WHERE ID(to) = row.to
4 CALL apoc.create.relationship(from, row.type, {containment:row.isContainment}, to) YIELD rel
```

Listing 4.2. Batch create relationships query

Listing 4.2 consists of four commands. We use the `UNWIND` keyword to transform the input list of parameters into individual rows (line 1). For each parameter, we search for a source node (line 2) and an target node (line 3) by ID. That is why the nodes must be already stored in the database. We create a relationship using the `apoc.create.relationship` function from the APOC library (line 4).

4.2.4. Loader example

We now present the loading process on a concrete example, using the model shown in Fig. 3.3a as the input for NMF-loader. Suppose we have a buffer size set to two elements for both objects and associations. The input model is a model of M_1 level, so the reader uses the reflective extractor for it. The reader starts the traversal from the root object `Graph`. The reader places it in the object container of the buffer. `Graph` only contains the CompositeVertex `V0` (via containment). Then, the reader puts `V0` with its `vertices` incoming containment association to the corresponding containers of the buffer. At this point, the objects container has two elements, the associations

container has one element, and the deferred associations container is empty. The objects container is filled, so the buffered elements are passed to the mapper. The mapper converts them into *NGraph* parameters and passes them to the loader component. The loader commits the two nodes to the database, sending a single query presented in Listing 4.1. As a response, the loader gets the IDs of newly created nodes and passes them back to the mapper component to let it perform ID bindings. Next, the loader commits the buffered relationships to the database, sending a query, presented in Listing 4.2. To store relationships, the loader uses the node IDs from the mapper. At this point, the first iteration is complete. The buffer is cleared and the reader continues to iterate over the model.

The reader starts the second iteration by putting the **default_vertex** association of the **V0** in the deferred associations container (because **V1** is not yet reached). Next, it will visit the vertex objects contained in **V0**. Among two existing **Vertex** objects, the reader will choose the first one depending on the serialization order. Suppose the reader passes to the object **V1**. It puts the object and its **sub_vertex** incoming containment association to the buffer. Also, the reader picks up the incoming **default_vertex** reference association from the deferred associations container and puts it to the associations container. At this point, the associations container is filled, so the reader passes the buffered elements to the mapper. Then the loader and the mapper component repeat the same operations as they performed on the first iteration. After the loader commits the data, the second iteration is complete.

Next, the reader continues to iterate through the model from **V1**. This object has one unprocessed **edge** association. The reader puts it to the deferred associations. Then, the reader passes to the **Vertex** object **V2** as the second containment from **V0**. The reader puts the current object **V2** as well as its incoming containment association to the buffer. Also, the **edge** association is transferred from deferred associations to the normal associations container. At this point, the associations container is filled. After the mapping, the loader component commits the data to the database. The third iteration is complete. The input model has no remaining element to process, so NMF-loader terminates the loading process.

In this chapter, we have presented the implementation details of NMF-loader module – a component of NMF to export existing MDE models from XMI format to Neo4j database.

Chapter 5

NMF-Editor

In this chapter we present NMF-editor module. In Section 5.1 we discuss the editing logic of models in MDE. In Section 5.2 we describe the architecture of NMF-editor. In Section 5.3 we describe our partial model editing mechanism. Finally, in Section 5.4 we present a usage example of NMF-editor.

5.1. Model editing logic

In MDE, there are generally four primitive operations that can be applied to the elements of a model: create, read, update, delete (the so-called CRUD operations). To maintain the well-formedness of a model, general pragmatic rules govern how to edit models. Given an *EModel* (defined in Section 4.1), the editing rules are as follows:

- (1) An *EModel* always consists of a unique root object with no incoming containment association. For example, **G1** is the root of the model in Fig. 3.3a
- (2) All non-root objects must be “contained” in another object. Thus, objects and containment associations always form a tree rooted at the root object. In the example, **G1** contains **V0**, which in turn contains **V1** and **V2**.
- (3) A non-root object can only be created by containment. For example, when creating the vertex **V1**, we must create the **sub_vertices** containment association from **V0**.
- (4) Deleting an object recursively deletes all objects it contains. Thus, deleting **V0** will also delete **V1** and **V2**. To avoid dangling references, all their adjacent associations are also deleted.
- (5) An object can only access to other objects directly related to it. For instance, in Fig. 3.3a, the graph **G1** can access the composite vertex **V0**, but not vertices **V1** and **V2**.
- (6) All attribute values must conform to the type specified in the metamodel of the *EModel* when they are updated.
- (7) All references must conform to the upper/lower bounds defined in the metamodel of the *EModel*.

Consequently editing an NGraph that represents an EModel in the database must abide to these rules. Therefore, the API that NMF-editor provides follows the same rules.

5.2. NMF-editor components

NMF-editor is a core module of NMF. It provides an API to edit graphs in a Neo4j database. Our implementation provides an API developed in Kotlin so that a user can use it from any JVM-based programming language and is thus compatible and easily integrable with the Eclipse Modeling Framework.

NMF-editor offers the following features:

- It provides editing operations for graph elements that comply with the editing logic of *EModels*, described in Section 5.1
- It relies on the *NGraph* data serialization schema, described in Section 4.1/
- It can interact with a remote Neo4j database instance.
- It supports on-demand loading to allow working with a sub-graphs and query from the database only needed elements.
- It supports buffering to accumulate locally a batch of updates in a cache before applying (i.e., committing) them on the database.
- To reduce communication overloading, all edits are cached locally. A user can apply CRUD operations to the buffered elements, thus modifying elements right in the cache.
- All elements remain accessible after the operations are committed to the database.

To provide these features, we have designed an architecture, presented in Fig. 5.1. NMF-editor has six components combined into two architectural layers: controller components and CRUD operation components. We present the detailed description of these components in the following sections.

5.2.1. Graph controller

Graph controller is the first component of NMF-editor available for a user. A short usage example of this component was presented in Listing 3.1 (lines 1–2, 9–10). The graph controller has four goals.

First, it manages a connection to the database. This component establishes a connection to various resources (e.g., database, buffer) and gives access to the graph. Usually, a user needs to

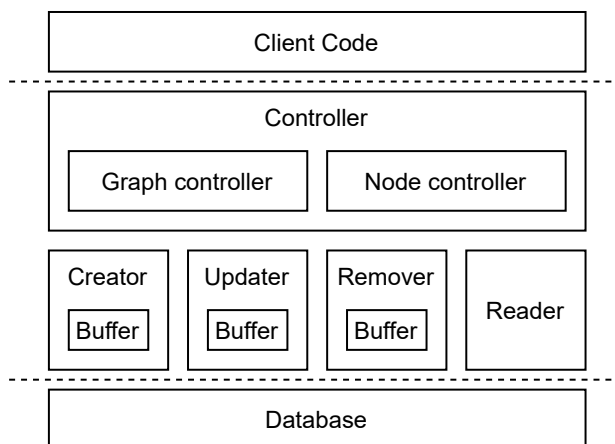


Fig. 5.1. The NMF-editor layered architecture and its components

keep one graph controller per database instance. At the end of the interaction, the graph controller must close the connection to release the resources.

Second, the graph controller spawns node controllers using the factory pattern [30]. A node controller represents a particular node and provides editing operations over it. The graph controller provides an API to create a node controller for either a new node or an existing one in the database.

Third, the graph controller provides an API to manage a cache of node controllers. We use the cache to have access to the elements after a commit. By default, creating a new node controller leads to putting it in the cache. A user has complete control over it and can decide which controller to remove from there. Internally, the cache is organized as an adjacency matrix that also keeps connections between node controllers. With it, we can query related node controllers from the cache.

Finally, the graph controller is responsible for initiating a commit of the current changes in the cache to the database. While editing, NMF-editor accumulates the write operation (i.e., elements to create, update, and remove) in independent buffers. These buffers form the cache. Node controllers “control” the elements in these buffers. With a node controller, we can edit elements directly in the cache, before committing, or in the database, after the commit. Thus, we use controllers to change the appropriate content of the buffers. The graph controller provides the `savechanges` function that flushes the buffers and applies the buffered operations to the database.

5.2.2. Node controller

A node controller gives access to a particular node. This component provides a specific set of operations and acts as a proxy for a node it represents. Whether the node is in the database or (and) in one of the buffers, inside `Creator`, `Updater`, or `Remover` components, the node controller generalizes the access to it. NMF-editor can have multiple node controllers at the same time.

Fig. 5.2 presents the main functionalities of a node controller. We split this component into two layers: `INodeEntity` represents an element, `NodeController` together with `PropertyAccessor` represent a logical layer over that element. This layer exposes to a user a specific functionality that conforms to the *EModel* editing logic, described in Section 5.1. We split the node controller to provide domain-specific editing operations support. As such, a domain-specific implementation can extend the `INodeEntity` interface providing a different functionality. We describe that approach in more detail in Chapter 6.

The `PropertyAccesor` class serves to modify node properties. The class provides common CRUD operations. The `putProperty` is a setter method that accepts a value of any JVM type (i.e., `Object`), including collections. Also, the `PropertyAccessor` provides `putUniqueProperty` method that ensures the property value uniqueness for nodes with the same label (i.e., objects of the same type in the *EModel*). Before putting the property value, we look for it in both the database and buffers of NMF-editor. If the uniqueness property is violated, the `PropertyAccesor` aborts the operation and throws a `UniqueValueException` if the value already exists.

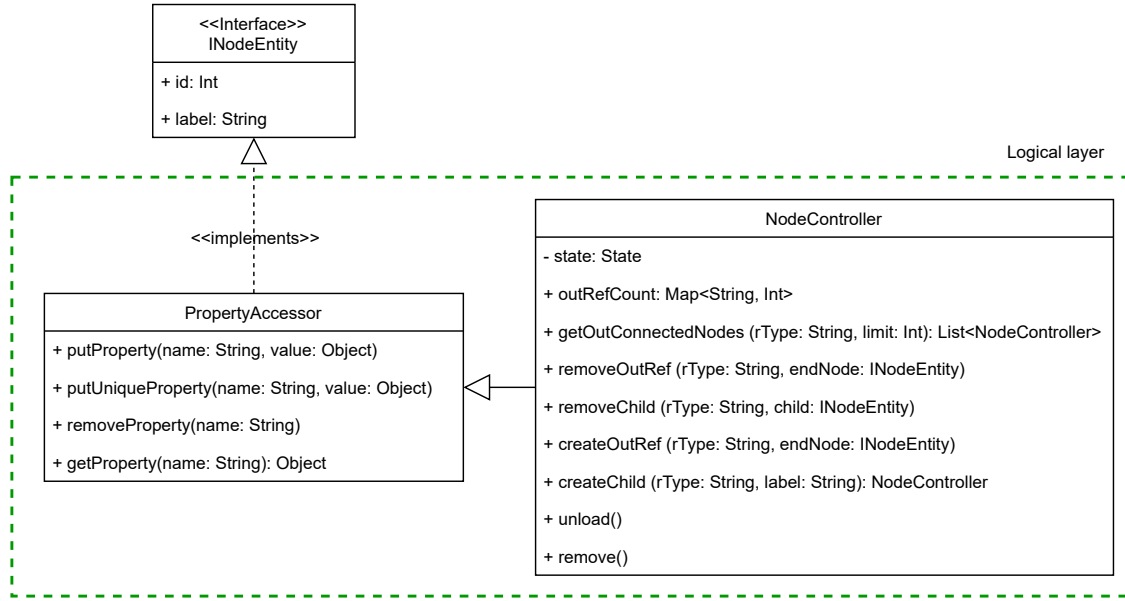


Fig. 5.2. The structure of a node controller

The **NodeController** class provides specific editing operations to CRUD related elements. The **createChild** method creates a new node together with a new incoming “containment” relationship from a node the current node controller represents. The relationship has a “containment” property set to true. In terms of the *EModel*, this means that it creates an object together with its incoming containment association. The **createChild** method takes two parameters: a type for the relationship and a label for the node. The newly created elements are passed to the buffer of the **Creator** component. The **createChild** method also creates and returns a new node controller that represents that newly created (target) node. That new node controller is then automatically saved to the adjacency matrix of the graph controller.

In terms of *EModel*, the **removeChild** method removes a contained object. It takes a relationship type and a node controller object as parameters. The node controller parameter represents a target node to remove. The type parameter helps us ensure whether the source and target nodes are actually connected by a specific relationship (because a source node can have multiple outgoing relationships of different types). Theoretically, a user can even pass as parameter a node controller that does not relate to the current one. In this case, the target node will not be removed.

Internally, the **removeChild** method analyzes the passed node controller object. If it represents a node in the database, we put it in the buffer of the **Remover** component. The node will be removed from the database next time a user invokes the **saveChanges** method of the graph controller. If the node controller represents a node in a buffer of the **Creator** component, we remove it from there, so it will not be persisted while invoking **saveChanges**. When the node is in the buffer of the **Updater** component, we do both operations: we remove the node from the buffer of the **Updater** component and put it to the buffer of the **Remover** component. In all the cases, we also remove a corresponding node controller from the adjacency matrix of the graph controller. Also, the **removeChild** method initiates a recursive deletion of all the sub-nodes contained in the target node. To find them in the

cache, we use the adjacency matrix of the graph controller. To remove them from the database, we use the power of Cypher query language. We describe that approach in more detail in Section 5.3.3. While removing a node, we also remove all its incoming/outgoing relationships in both database and cache (including buffers and adjacency matrix).

The `getOutConnectedNodes` method gets nodes connected by outgoing relationships. This method takes two parameters: a relationship type and a limit – the maximum allowed number of nodes to return. First, this method looks for corresponding node controllers inside the adjacency matrix of the graph controller. If the limit number is not satisfied, the function makes a request to the database using the reader component. For each returned node, we create a new node controller and put it in the adjacency matrix. The `getOutConnectedNodes` method combines the node controllers from the cache and the database into a single list and returns it to a user. We do not use a buffer for reading operations. Instead, we return the result to the user immediately.

The `createOutRef` and `removeOutRef` methods operate on outgoing relationships of a node the current node controller represents. In terms of *EModel*, these functions represent operations on reference associations. Both methods require a relationship type and a node controller of a target node as parameters. While creating the relationship, we assign the boolean “containment” property to false. Then we put the relationship to the buffer of the `Creator` component and the graph controller’s adjacency matrix. While removing a relationship, we analyze the target node controller. If it represents a new node, the node itself and its relationships are added to the buffer of the `Creator` component.

A node controller tracks an actual number of outgoing relationships. The `outRefCount` map groups a relationship count by the relationship type (a key is a relationship type, and a value is an actual integer number of the outgoing relationships). While invoking the `createChild`, `createOutRef`, `removeChild`, `removeOutRef` methods, we appropriately increment/decrement the actual number of outgoing relationships in the map. The tracking of outgoing relationships count allows us to provide a mechanism of checking upper/lower bounds. We describe it in more detail in Section 6.3.3.

The `remove` method of the graph controller removes the node the current node controller represents. Internally, this method acts in the same manner as the `removeChild` method, respecting a recursive removing of all the containment sub-nodes. However, unlike the `removeChild`, the `remove` method starts removing from the current node.

The `unload` method removes the node controller itself from the cache (more precisely, from the adjacency matrix of the graph controller). After unloading, the node can still exist in the database or in one of the buffers of NMF-editor. If the node is in a buffer, it will be uploaded to the database the next time a user invokes the `saveChanges` method.

5.2.3. Controller states

The node controller uses the state pattern [30] to handle the node transitions between different components and the database. The node controller acts differently depending on which of the following states it is in: `New`, `Persisted`, `Modified`, `Removed`, and `Unloaded`. Fig. 5.3 illustrates

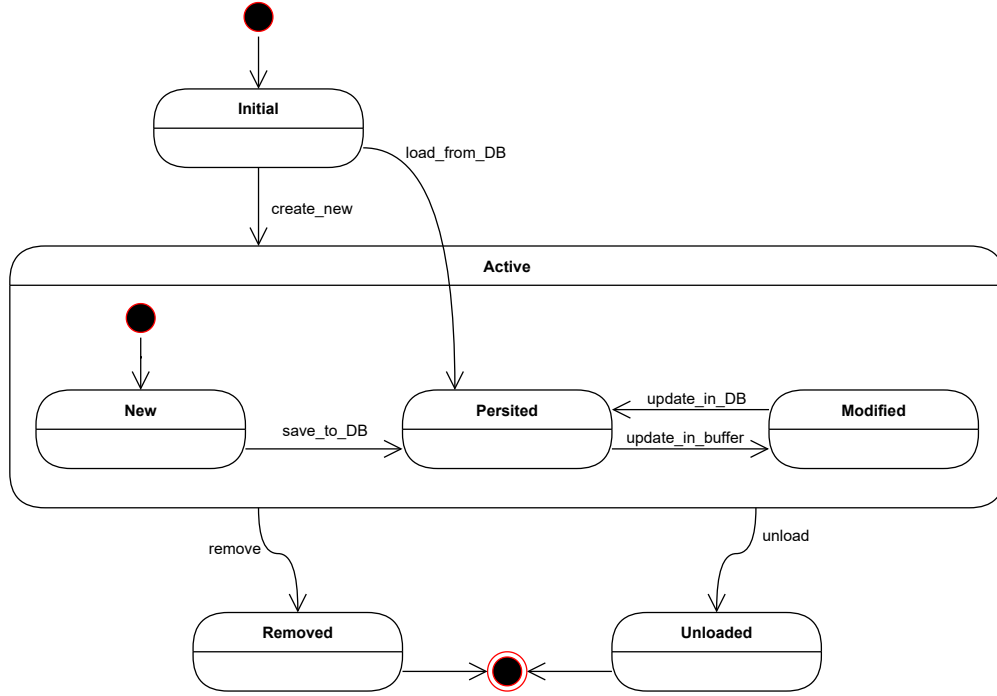


Fig. 5.3. Node controller states

the transitions between them. Initially, a user can create a new node controller either for an existing node in the database (while reading it) or for a new node. The node controller in state **New** represents a new node located in the buffer of the **Creator** component. State **Persisted** indicates that the node controller represents a node from the database. The **saveChanges** method of the graph controller triggers the transition from **New** to **Persisted**.

State **Modified** indicates that the node controller represents a node in the database, but the properties of the node have been modified and not yet committed to the database. The updated properties are located in the buffer of the **Updater** component. The node controller goes from **Persisted** to **Modified** when the user invokes **putProperty** or **putUniqueProperty**. It becomes **Persisted** when the changes are committed to the database with the **saveChanges** function.

State **Unloaded** indicates that the node controller itself was removed from the cache. This state serves to stop tracking the node and prevent any editing operations over it. We can switch to this state on invoking the **unload** function of the node controller. The **Unloaded** state is final.

State **Removed** indicates that the node controller represents a node that was removed and does not physically exist neither in the database nor in the buffer. Simultaneously, the node controller itself is removed from the cache and cannot be used for any interaction. We can pass to this state from any active state (i.e., **New**, **Persisted**, **Modified**). **Removed** is a final state for a node controller.

5.3. Partial graph editing

To start interacting with a node, a user must have a node controller for it. We hold node controllers in the cache (adjacency matrix of the graph controller) of NMF-editor. Because of the “on-demand loading”, we do not hold the entire database content in the system. Instead, a user can load in the cache only the node controllers needed. Also, we provide a mechanism to unload unnecessary controllers (that represent the nodes a user no longer needs to interact with) from the cache using the `unload` function. This kind of manipulation may lead to cache fragmentation. This means that the cache can contain partial graphs, as shown in Fig. 5.4. In the lower part of the figure, we can see an example of a consistent graph in the database. The upper part of the figure contains an example of node controllers representing particular nodes in the database. We mark each controller with the appropriate number that corresponds to a number of a node it represents. Partial graphs in the cache of NMF-editor do not have to be connected, even if the complete graph in the database is. Therefore, we should provide CRUD operations that correspond to *EModel* editing logic on partial graphs, keeping data consistency in the database.

Let us consider all the CRUD editing operations for partial graphs, following the example presented in Fig. 5.4. Suppose a user creates a new node #12 that is contained in node #11. Upon creation, the user immediately gets a node controller with state `New` for that node. Although the node is in the buffer and has not yet been stored to the database, he can operate on it as it is a part of the graph. This means that a user can rely on that controller to create new contained nodes and relationships.

We use the same logic for the update operation. If a user modifies the properties of a node, the node controller represents the node with the updated properties, despite the modifications have not yet been stored in the database.

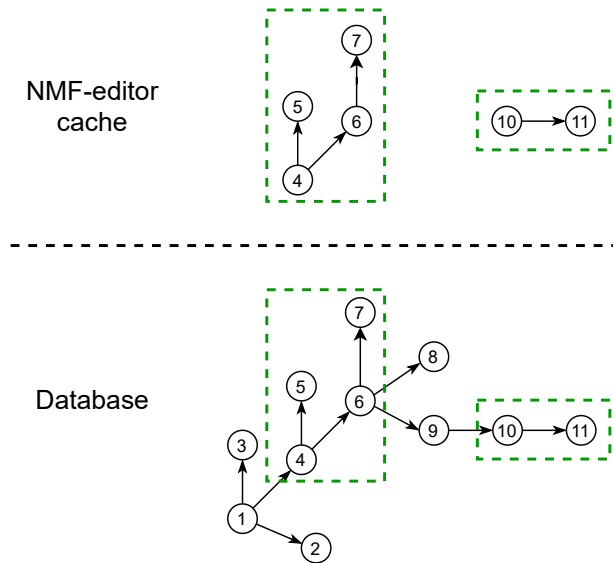


Fig. 5.4. NMF-editor cache fragmentation

We do not use the buffer for the read operation. The node controller is immediately created with state **Persisted** and stored in the cache.

The delete operation acts according to the *EModel* editing rules: it removes the node and all its containment sub-nodes in-depth recursively. We use the buffer for the delete operation as well as for the create and update operations. This means that the actual deletion of elements in the database will not be performed until a user invokes **saveChanges** function.

Because of the “on-demand loading”, node controllers that are subject to be removed may be in different partial graphs of the cache. To keep consistency between partial graphs in the cache, we split the delete operation into four steps that are described below.

- (1) Put to the buffer of the **Remover** component of the target node to remove. This node represents a root element of a partial graph that should be removed.
- (2) Analyze the cache of node controllers using the adjacency matrix. We recursively find all the available controllers in the cache that correspond to the contained nodes of the target node to remove. We remove those controllers from the cache and change their status to **Removed**. This state prevents any further editing operations over these elements.
- (3) Commit the buffer of nodes to remove (when a user invokes the **saveChanges** function) to the database. We rely on the database to find and remove all the contained nodes of the target node.
- (4) Return from the database the IDs of all removed nodes. Then, we again analyze the cache and look there for node controllers with that IDs going back to step (2). The process terminates when no more IDs are returned.

Suppose a user removes node #6 in the lower part of Fig. 5.4. We consider all relationships in the figure as containment associations. This means that while removing node #6 we also need to remove all its contained nodes 7–11. After an invocation of the remove function of node controller #6, we put a node #6 to the buffer of the **Remover** component. Also, we remove node controllers #6 and #7 from the cache and switch their state to state **Removed**. However, node controllers #10 and #11 are still in an active state. Suppose the user then uses node controller #11 and creates for it a new contained node #12. Although this action is not allowed according to *EModel* editing rules, he can still interact with node controllers #10 and #11. If the user then invokes the **saveChanges** function, NMF-editor will proceed with the remove operation first. We remove nodes 6–11 in the database and return to the NMF-editor their IDs. Then, we use these IDs to remove the node controllers #10, #11, and newly created #12. Additionally, we remove node #12 from the buffer of the **Creator** component to not allow committing of the unnecessary elements to the database. Similarly, we perform the same operation for the buffer of the **Updater** component if needed (if the user modifies a node that should be removed). Next, the remove operation is done, so we pass to the create and update operations. In our case, the two corresponding buffers are empty, so the entire commit is done. As a result, the database will contain only nodes 1–5 after this commit.

The response we get from the database is to keep consistency between partial graphs in the cache. The **saveChanges** function of the graph controller initiates a commit. This commit consists of three operations: create, delete, and update. These operations are triggered in a specific order: the delete operation comes first before the create and update operations.

5.3.1. Element creation

The **Creator** component has two goals: managing a buffer of new elements and uploading them to the database. It uses similar logic for uploading elements as the loader component of NMF-loader, described in Section 4.2.3. To upload elements, we reuse two parametrized queries of NMF-loader, that were presented in Listings 4.1 and 4.2. The first one to upload nodes and the second one to upload relationships. Also, we split the buffer to hold the nodes and relationships in separate containers of the buffer and use them as two independent sets of parameters for the queries. The parameters of nodes and relationships have the same structure as in the loader component. After uploading the nodes, the **Creator** component takes as a response the IDs of newly created nodes (because we rely on internal database IDs). Then we use these IDs to update the corresponding field of active node controllers.

There are only two differences between the **Creator** component and the loader component. First, the **Creator** component can manage his buffer (add and remove elements), while NMF-loader only adds. In NMF-editor, the user can create a new element, then remove it before the commit. In this case, the manipulation is done in the buffer: the node is removed from it and will not be uploaded to the database.

The second difference is that the buffer of the **Creator** component does not have a fixed size. A user can add as many elements as he wants before invoking the **saveChanges** function that triggers the commit. Instead, while uploading the elements, we split the buffer (for both nodes and relationships) into chunks. For instance, if the container of the relationship of the buffer exceeds the chunk size, the **Creator** component will resubmit the query with a new portion of parameters as long as the container has remaining elements to process. This approach allows to handle large commits and upload them successively.

5.3.2. Element updates

Updating an element means to modify its properties. In NMF-editor, we only update nodes to follow the *EModel* editing rules. The user can initiate a property modification of a node by using the functionality of the **PropertyAccessor** class.

In NMF-editor, the updating operation has two different variants. At any point in time, a node can be either in the database or in the buffer of the **Creator** component. For a new node, we simply update a map of **properties** for a particular node in the buffer of the **Creator** component. We then upload the new nodes together with their properties using the query in Listing 4.1. Before uploading, all the manipulations with properties are performed in the buffer of the **Creator** component.

To update the properties of an existing node in the database, we use the **Updater** component of NMF-loader. This component has two goals: manage a buffer of modified properties and upload them to the database. We accumulate the updated properties in the buffer until the user invokes the **saveChanges** function. This function initiates a commit that contains the update operation (alongside the create and delete operations).

To update node properties in the database we use a query presented in Listing 5.1. This is a parameterized query that takes a list of parameters. We use the buffer to form a set of parameters for the query. Each element of the buffer has the following structure:

- **id** – the node id
- **properties** – a map of new properties for the node (key is a property name, a value is one of the allowed Neo4j value types)

Thus, the entire buffer has a structure of a map of maps.

```
1 UNWIND $batch AS row
2 MATCH (node) WHERE ID(node) = row.id
3 SET node += row.properties
```

Listing 5.1. Cypher query to update nodes in the database

In Listing 5.1, we use the **UNWIND** command to convert the input list of parameters into individual rows (line 1). The **row** variable represents each independent parameter (as counter inside a for-loop). We look for a node with a specific ID (line 2) and assign properties using **SET** command with **+=** operation (line 3). This operation acts as follow:

- any property in the **properties** map that are not in the node will be added
- any property not in the map that are in the node will be left as is.
- any property that are in both the map and the node or relationship will be replaced in the node

5.3.3. Element removal

The delete operation has two different variants depending on the actual element location. To remove a new node or relationship that is in the buffer of the **Creator** component, we pop the appropriate parameter from there, excluding it from the commit. To remove an existing element in the database, we use the **Remover** component. This component has two goals: accumulate elements to remove in the buffer and upload them to the database. We accumulate the elements in the buffer until a user invokes the **saveChanges** function. The buffer has two separate containers: for nodes and relationships. We use these containers to form two sets of parameters for parameterized queries.

5.3.3.1. Nodes removal. To remove nodes in database we use a query, presented in Listing 5.2. This query takes a list of node IDs as a parameter. Each ID serves to find a node that should be removed. We remove a node together with all its contained nodes in-depth recursively according to the *EModel* editing rules. Each parameter in the buffer denotes a root element of a “containment” sub-graph to be removed. For each node, we discover the sub-graph in the database and remove all its elements. The buffer may contain multiple elements, so we are able to remove multiple sub-graphs within a single commit.

```
1 UNWIND $batch AS row
2 MATCH (node) WHERE ID(node) = row.id
3 WITH node
```

```

4 MATCH (node)-[*0..{containment:true}]->(d)
5 WITH d, ID(d) AS removedIDs
6 DETACH DELETE d
7 RETURN removedIDs

```

Listing 5.2. Cypher query to remove containment nodes

In the query, we look for a node with the specified ID (line 2). To refer to this element later in the same query, we assign it an alias `node`. The `WITH` keyword serves to restrict variable use inside the query. This command forms a new scope where only the `node` variable is available (line 3). Variables `row` and `$batch` are not visible after line 3.

Next, we look for all the nodes contained in the root `node`. Inside the relationship notation (square brackets), we specify a filter to find only containment relationships. The notation `*0..` (from 0 to infinity) specifies an arbitrary number of hops between nodes. Thus, the alias `d` will represent all the contained nodes along with `node`. We form a new scope of variables in the query, propagating variables `d` and `removedIDs` (line 5).

Then, we remove the nodes the alias `d` represents together with all the relationships related with them (line 6) (because a relationship cannot exist without a source and target node). Finally, we return the IDs of all the removed nodes. We then use those IDs to set to the `Removed` state the node controllers representing nodes in the removed sub-graph and were not affected before the commit.

5.3.3.2. Relationships removal. To remove relationships from the database we use a query presented in Listing 5.3. Because we do not use relationship controller in NMF-editor, we cannot track relationships and rely on their IDs. Thus, to find and remove a relationship in the database we rely on four parameters:

- source node ID
- target node ID
- relationship type
- limit – a number of relationships with a certain type to remove

Each element of the relationship container of the buffer has exactly the same structure. The last parameter needs detailed explanation. Suppose we have five relationships between two nodes: three labeled `type1` and two labeled `type2`. Suppose the user wants to remove two relationships of type `type1`. To perform this operation in one query and to not rely on IDs of relationships, we provide the following parameters: source/target node IDs, relationship type: `type1`, limit: 2.

```

1 UNWIND $batch as row
2 MATCH (start) WHERE ID(start)=row.startID
3 CALL apoc.cypher.doIt("
4   MATCH (start)-[r]->(end)
5   WHERE type(r)=rType AND ID(end)=endID
6   WITH r LIMIT $limit
7   DELETE r",
8   {start:start, rType:row.rType, endID:row.endID, limit:row.limit})

```

```

9 ) YIELD value
10 RETURN value

```

Listing 5.3. Cypher query to remove relationships by source/target node IDs

In the query, we use the `UNWIND` keyword to transform a list of parameters into individual rows (line 1). Then we look for a source node of the relationship (line 2). We use `start` alias to represent this node. Next, we start a sub-query using `doIt` function of APOC library (lines 3–9). We need a sub-query to not break the `UNWIND` process since each subsequent command acts as inside a for-loop. Hence, a sub-query initiates an “inner loop” with its own parameters scope. Inside the sub-query, the alias `end` represents the target node. The alias `r` represents all the relationships connecting the source and target nodes. Then, we limit the number of relationships (respecting the case) the alias `r` represents (line 6). Finally, we remove them from the database (line 7).

5.3.4. Reading elements

NMF-editor provides two different reading cases. The function `readNodeByID` of the graph controller triggers the reading query, presented in Listing 5.4. This query takes a single `id` parameter. It returns the node found in the database (or null if a node with the ID does not exist). Based on the returned result, the `Reader` component then creates a new node controller with the state `Persisted` and puts it to the cache.

```

1 MATCH (n)
2 WHERE ID(n) = $id
3 RETURN n

```

Listing 5.4. Cypher query to read a node by ID

The function `getOutConnectedNodes` of the node controller gets related nodes from the database. It triggers the reading query presented in Listing 5.5.

```

1 MATCH (start) WHERE ID(start) = $startID
2 CALL apoc.path.subgraphNodes(
3   start, {relationshipFilter: $refType, labelFilter: $label, minLevel:1, maxLevel:1}
4 ) YIELD node
5 WITH node LIMIT $limit
6 OPTIONAL MATCH (node)-[r]->()
7 WITH ID(node) AS id, labels(node)[0] AS label, type(r) AS rType, count(r) AS count
8 RETURN id, label, apoc.map.fromPairs(collect([rType, count])) AS count

```

Listing 5.5. Cypher query to read related nodes

We look for the source node by ID in database (line 1). To represent this node in the query we use the `start` alias. Then, we use a `subgraphNodes` function from the APOC library to find related nodes of the source node (lines 2–4). This function takes five parameters:

- `start` – a source node to find related nodes for

- `relationshipFilter` – a type of an outgoing relationship that connects the source node to the target node (a source node can have multiple outgoing relationships of different types)
- `labelFilter` – a label of the target node
- `maxLevel` – the maximum number of hops from the source node
- `minLevel` – the minimum number of hops in the traversal from the source node

In our case, we explicitly bound the min/max hops in the range $[1, 1]$ to get only the nearest related nodes to the source node. All the variables in the query that start with `$` sign are the runtime query parameters that come from the `Reader` component according to the user needs. The alias `node` represents all the found related nodes that correspond to the provided filters (line 4). Then, we limit the number of the returned nodes (line 5).

For each `node` we look for outgoing relationships and group them into the map by a relationship type (lines 6–7). This map help us to respect the upper/lower bounds and to construct a valid node controller object (to initialize `outRefBound` variable in the `NodeController` class). Finally, we return from the database the related node ID, label, and the map of outgoing relationships to properly initialize node controller objects (line 8).

5.4. Editor example

We now present an editing example in Listing 5.6. In this example we use a graph structure, presented in the lower part of Fig. 5.4. For that graph we use the concept of the GRAF DSL, presented in Fig. 3.2.

```

1  val graphController = GraphController(dbUri, username, password)
2  val v1 = graphController.createNode("Graph")
3  val v2 = v1.createChild("sub_vertices", "CompositeVertex")
4  val v3 = v1.createChild("sub_vertices", "CompositeVertex")
5  val v4 = v1.createChild("sub_vertices", "CompositeVertex")
6  val v5 = v4.createChild("sub_vertices", "CompositeVertex")
7  val v6 = v4.createChild("sub_vertices", "CompositeVertex")
8  val v7 = v6.createChild("sub_vertices", "CompositeVertex")
9  val v8 = v6.createChild("sub_vertices", "CompositeVertex")
10 val v9 = v6.createChild("sub_vertices", "CompositeVertex")
11 val v10 = v9.createChild("sub_vertices", "CompositeVertex")
12 val v11 = v10.createChild("sub_vertices", "CompositeVertex")
13 graphController.saveChanges() // commit #1
14 v9.putProperty("name", "V9")
15 graphController.saveChanges() // commit #2
16 graphController.unload(v2)
17 graphController.unload(v3)
18 graphController.unload(v9)
19 println(v9.getProperty("name")) // throws exception -- controller v9 is in state Unloaded
20 val vertices = v1.getOutConnectedRef("sub_vertices", 3) // read operation
21 for (v in vertices) {
22     println(v.id)

```

```

23 }
24 v6.remove()
25 v7.createChild("sub_vertices", "Vertex") // throws exception -- controller v7 is in state
    Removed
26 val v12 = v11.createChild("sub_vertices", "Vertex")
27 graphController.saveChanges() // commit #3
28 graphController.close()

```

Listing 5.6. NMF-editor example

We create a `graphController` object that establishes a connection to the database (line 1). Then, we recreate a graph structure presented in the lower part of the Fig. 5.4 (lines 2–12). Each variable `vX`, where `X` is a number that corresponds to the node in the figure, represents a node controller with state `New`. These node controllers are added to the cache (adjacency matrix of the graph controller), and the corresponding parameters are added to the buffer of the `Creator` component. Then, we invoke the `saveChanges` function and start a the commit to the database (line 13). At this point, the buffers of the `Remover` and `Updater` components are empty, so the commit contains only the create operation. After the commit, the buffer of the `Creator` component is flushed, node controllers `v1`–`v11` switch their state from `New` to `Persisted`.

Next, we demonstrate the update operation by assigning a property `name` to the node `#9` (line 14). The node controller changes its state from `Persisted` to `Modified`. The buffer of the `Updater` component now contains that property. We upload the modifications to the database in a new commit (line 15).

Next, we demonstrate the on-demand loading feature of NMF-loader. We unload the node controllers `v2`, `v3` and `v9` from the cache (lines 16–18). After this action, appropriate node controllers change their states to `Unloaded`. Any interaction with an unloaded node controller throws an exception (line 19). Internally, we also remove these controllers from the cache that leads to cache fragmentation. At this point, controllers `v10`–`v11` are detached from other controllers in the cache.

Now we demonstrate the reading operation. We use node controller `v1` to load from the database related nodes, connected by the `sub_vertices` relationship (line 20). The function `getOutConnectedNodes` returns a list of node controllers with state `Persisted`. We have specified the desired number of nodes to read (3) as a parameter for the function. At this point, the cache contains only node controller `v4`. Also, the function initiates a read operation to the database and constructs new controllers for nodes `#2` and `#3`. These new controllers update the cache and make available editing operations on appropriate nodes. So, the list `vertices` contains controllers for nodes `#2`, `#3` and `#4`. To demonstrate the usage of these controllers, we output the IDs of nodes they represent (lines 21–23).

Next, we demonstrate the delete operation. We remove node `#6` (line 23). We have already explained the overall removal process of node `#6` in Section 5.3. Node controllers `v6`–`v7` (because `v7` represents a contained node) change their state to state `Removed`. Any further editing operations with that node controller `v7` are not allowed (line 24). However, controllers `v10`–`v11` are not

reachable in the cache (because we have previously unloaded the controller `v9` on line 18), so their states are unchanged at this moment. A user can still interact with those controllers (line 25), but that interaction will not affect the graph in the database. Finally, we perform the third commit (line 26). At this point, the buffer of the **Updater** component is empty, the buffer of the **Creator** component has one element (node `#12`), and the buffer of the **Remover** component also has one element (node `#6`). Recall that the `saveChanges` function performs the delete operation first. Nodes 6–11 are removed from the database, and NMF-editor gets their IDs as a response. Then, in the cache, we find node controllers `v10`, `v11`, `v12` and change their state to **Removed**. The removal of the node controller `v12` leads to the popping of the node `#12` from the buffer of the **Creator** component. So, the buffer of the creator component becomes empty, and the entire commit is done. The database now contains only nodes 1–5.

NMF-editor acts like a continuously running service. In the presented example, the variables of inactive node controllers remain available for a user. Indeed, the JVM garbage collector will destroy them once the end of scope is reached.

Chapter 6

NMF Generator

In this chapter we present NMF-generator. It produces an abstraction layer on top of NMF-editor closer to the domain of the metamodel. In Section 6.1 we present the architecture of the generated abstraction layer. In Section 6.2 we describe the generation process of the abstraction layer. In Section 6.3 we provide a usage example of the generated API.

6.1. Domain-specific editor components

NMF-generator serves to create a domain-specific API to edit graphs in the Neo4j database. Like NMF-editor, the generated API abides to the *EModel* editing logic, presented in Section 5.1. Also, the generated domain-specific API provides editing operations tailored to a specific domain. Thus, a domain expert can use the API to edit a domain-specific *EModel*, even if the model is stored as a graph. Therefore, we consider the generated API as a domain-specific model editor.

Fig. 6.1 presents the architecture of a domain-specific (generated) editor. It has three components: the domain-specific object controller, the domain-specific model manager, and the domain-specific utilities. These components completely rely on NMF-editor. Therefore, the domain-specific editor provides the same features, as enumerated in Section 5.2. We provide a detailed description of each component in what follows.

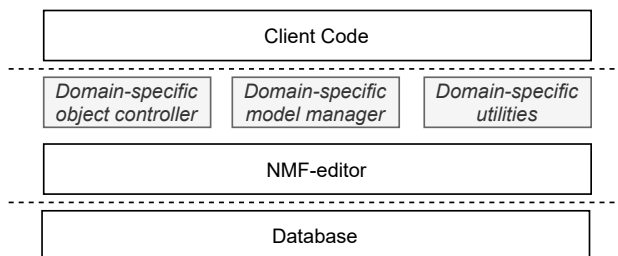


Fig. 6.1. Domain-specific editor architecture

6.1.1. Domain-specific model manager

The domain-specific model manager is a domain-specific wrapper of the graph controller component (presented in Section 5.2.1). This component gives access to a model stored as a graph. Internally, the model manager is implemented using the adapter design pattern [30]. The model manager encapsulates the graph controller object (see Section 5.2.1) and propagates function invocations to it. This approach allows to hide the generic API of NMF-editor and expose to the user domain-specific functions with appropriate naming and restrictions. Thus, the model manager has the same four goals as the graph controller has.

First, it gives the initial access to the model. It manages a connection with database using the underlying graph controller object.

Second, the model manager spawns domain-specific object controllers. For each non-abstract class of the domain (i.e., concrete class of the metamodel) the model manager provides two functions: `createX` and `getXByID` where `X` is a name of the corresponding class. For example, the model manager class generated for the GRAF DSL (presented in Fig. 3.2) provides `createVertex` and `getVertexByID` functions for the `Vertex` class of the metamodel. These two functions return domain-specific object controller of type `Vertex`. Inside the `createVertex` function, we invoke `createNode("Vertex")` of the graph controller with the predefined parameter. Recall that the function of the graph controller creates and returns a node controller. Hence, for each domain-specific object controller, a new node controller with an appropriate state is created.

Third, the model manager provides the `unload` function for each non abstract class of the metamodel to manage the cache of the underlying graph controller. For instance, the model manager provides `unload(v: Vertex)` function for the `Vertex` class of the metamodel.

Finally, the domain-specific model manager component is responsible for initiating a commit: it exposes to the user the `saveChanges` function. Internally, this function invokes the same function of the underlying graph controller.

6.1.2. Domain-specific object controller

This component is the domain-specific counterpart of the node controller component (presented in Section 5.2.2). It gives access to a particular object of *EModel*.

Fig. 6.2 presents the design of the domain-specific object controller. This component consists of two artifacts: an interface (corresponding to a `DomainClass`) and a class that implements that interface (corresponding to a `DomainClassImpl`). This approach decouples the abstraction from its implementation, following the Dependency Inversion Principle [31]. Hence, the user can rely only on abstractions (interfaces), not on concrete implementations to operate on model elements, to favor modularity.

A `DomainClassImpl` (i.e., a domain-specific object implementation class) is implemented using the adapter pattern [14] (in the same manner as the domain-specific model manager). The domain-specific controller encapsulates a node controller and propagates function invocations to it using predefined parameters. Hence, the implementation class adapts a node controller to domain specific object controller. For instance, to update the `name` attribute of a `Vertex` object, the object controller

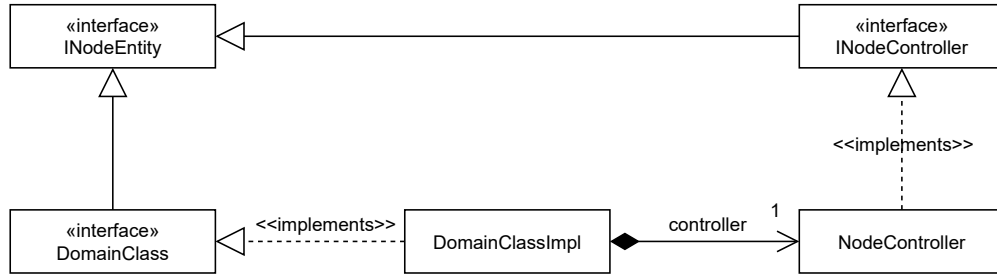


Fig. 6.2. Domain-specific object controller

provides the `setName` function. Inside this function, we invoke `putProperty("name", "value")` on the underlying node controller. Thus, a domain-specific object controller entirely relies on the underlying node controller.

6.1.3. Domain-specific utilities

This component generalizes various generated helpers for model editing. They are used to correctly express the editing logic of *EModel* in a generated API. The domain-specific utilities consist of two types of helpers:

- Additional data types that are defined in a metamodel (such as enumerations).
- Type-specifiers to correctly handle inheritance while creating object controllers. With the type-specifier a user can explicitly specify a concrete type of an object controller to create a hierarchy of several inherited types.

We provide a more detailed description of this component in Section 6.2.

6.2. Generation process

NMF-generator automatically generates the domain-specific API so the client code can seamlessly use the API to edit models. The resulting API (i.e., the output of NMF-generator) is a code in the Kotlin programming language.

The generation is mainly an iterative process which looks as follows. NMF-generator gets a metamodel in Ecore format and iterates over its contents, translating elements into Kotlin code artifacts (i.e., classes, interfaces, and enumerations) according to predefined textual templates. An input metamodel must conform to the M_3 meta-metamodel presented in Fig. 2.5. Therefore, we can extract from it concrete **Classes**, **Attributes**, **Associations**, and **Enumerations** including their related information (such as **name** or **upperBound** of an association).

NMF-generator produces Kotlin artifacts corresponding to the domain specific editor components described in Section 6.1. The generator starts to iterate over the metamodel from **Classifier** elements (that can be either a class or an enumeration). For each **Enumeration** of the metamodel, NMF-generator produces a corresponding Kotlin enumeration class that includes all the **EnumLiterals**. This generated artifact is a part of the domain-specific utilities component (described in Section 6.1.3).

For each **Class** of the input metamodel, NMF-generator produces an interface and a class implementing that interface. These two artifacts together represent a domain-specific object controller (described in Section 6.1.2). Next, for each metamodel class, we discover attributes and associations to generate Kotlin interface/class members.

Also, while iterating over the metamodel classes, NMF-generator simultaneously fills members of a model manager Kotlin class (described in Section 6.1.1). This class is completely produced after the overall iteration is completed because it requires information about all the metamodel classes.

Finally, NMF-generator produces a helper Kotlin enumeration class for each inheritance hierarchy present in the metamodel. Classes of a metamodel that directly or indirectly extend the current class are part of a hierarchy. The generated artifact enumerates all the non-abstract classes of the hierarchy. For instance, in our GRAF DSL the **CompositeVertex** extends **Vertex**. Therefore, NMF-generator produces the **VertexType** Kotlin enumeration (listed in Listing 3.2 line 29). The generated enumeration contains both **Vertex** and **CompositeVertex**. We discuss in detail the usage of this artifact in Section 6.4.

We implemented this code generator using Kotlin templates. We explain in details each component generation in the following sections.

6.3. Object controller generation

An object controller consists of an interface and its implementation class (Fig. 6.2). NMF-generator produces code artifacts considering Kotlin syntax. Therefore, to assemble a single object controller NMF-generator relies on the classes of the metamodel to produce Kotlin interface/class header, and the attributes and associations of the class to generate methods of the interface/class body.

6.3.1. Header generation

NMF-generator synthesizes a **Class** of a metamodel into headers of the class and interface of an object controller. To declare the header of an interface, we use the following template:

```
1 interface $className : $superTypes {
```

Listing 6.1. Template of the interface header of an object controller

The variable **className** denotes a capitalized name of the current class of the input metamodel. The variable **superTypes** is a string of names of direct supertypes separated by commas. If the metamodel class has no supertypes, the **superTypes** variable denotes a string **INodeEntity**. We extend the **INodeEntity** interface of NMF-editor to have access to specific methods of the underlying node controller. For instance, the **Vertex** class in GRAPH DSL (defined in Fig. 3.2) does not have any supertype. Thus, the **Vertex** interface in Listing 3.2 extends **INodeEntity** interface (line 8). In contrast, the **Vertex** class is a supertype of the **CompositeVertex**. Therefore, the **CompositeVertex** interface in Listing 3.2 extends **Vertex** interface, implicitly extending **INodeEntity** (line 19).

We use the following template to declare the header of an implementation class:

```
1 $abstract class ${className}Impl(private val controller: INodeController) : $className {
```

Listing 6.2. Implementation class header template of a domain-specific object controller

If a metamodel class is abstract, we put the corresponding **abstract** Kotlin keyword before the class name. If the class of the metamodel is an interface, we do not generate an implementation class for it. In this special case the object controller consists only of a Kotlin interface. The class name contains an **Impl** suffix to distinguish between the interface and the implementation class. Also, we generate a constructor that takes a node controller object. Finally, we extend the interface defined in Listing 6.1 using the **className** variable.

6.3.2. Attributes setter/getter generation

NMF-generator synthesizes an **Attribute** of the metamodel class into Kotlin getter and setter methods for both the interface and the implementation class of the object controller. All the templates for attribute generation rely on the **eAttr** variable, that represents the **Attribute** object of a metamodel. We use two different templates depending on **upperBound** of the attribute. An **upperBound** cardinality of 1 specifies single-valued attribute. Otherwise, for multi-valued attributes (**upperBound** > 1), the attribute specifies a collection.

We present a template for single-valued attribute in Listing 6.3. This code is generated into the body of the interface of an object controller.

```
1 fun set$attrName(v: $type?)
2 fun get$attrName(): $type?
```

Listing 6.3. Interface template for the single-valued attribute

In Listing 6.3 the getter/setter methods start with **get** and **set** prefix respectively. The variable **attrName** denotes a capitalized name of the **eAttr** object. The variable **type** specifies the type of the attribute. The ? (question mark) indicates a nullable type in Kotlin. Thus, a setter/getter function can accept/return a **null** value.

Listing 6.4 presents the implementation for getter and setter. We use this template to generate the code in the implementation class of the object controller.

```
1 override fun set$attrName(value: $type?) {
2     if (value == null) controller.removeProperty("${eAttr.name}")
3     else controller.putProperty("${eAttr.name}", value)
4 }
5 override fun get$attrName(): $type? {
6     return controller.getProperty("${eAttr.name}", $mapFunc)
7 }
```

Listing 6.4. Implementation template for the single-valued attribute

In Listing 6.4, the **attrName** variable denotes the capitalized name of the **Attribute** as explained earlier. In contrast, the **eAttr.name** variable specifies the actual name (not capitalized) of the attribute. If the value is set to **null**, we remove the property of the corresponding node in Neo4j

(line 2). As you can see, inside the implementation class we invoke functions of the underlying node controller with predefined parameters (lines 2, 3, and 6). To correctly transform a value type from Neo4j to Kotlin type inside the getter method, we use the `mapFunc` variable (line 6). For instance, for a metamodel attribute of type `Float` we use `AsFloat` map function.

For multi-valued attributes, we use the template presented in Listing 6.5 for interface code generation. The only difference between the template in Listing 6.3 is that we use the `List` Kotlin type to specify a collection.

```
1 fun set$attrName(v: List<$type>?)
2 fun get$attrName(): List<$type>?
```

Listing 6.5. Interface template for the multi-valued attribute

Listing 6.6 presents the template for the implementation of setter and getter for multi-valued attributes. In this case, we generate code that provides a `List` parameter type and allows the user to get/set the entire collection. In the setter method, we check if the list satisfies the allowed bounds (lines 4–5). If the bounds of the `Attribute` are not specified explicitly in the metamodel, we omit the bounds check generation. This allows to set a list of any size.

```
1 override fun set$attrName(value: List<$type>?) {
2     when {
3         value == null || v.isEmpty() -> removeProperty("${eAttr.name}")
4         value.size in ${eAttr.lowerBound}..${eAttr.upperBound} -> putProperty("${eAttr.name}", v)
5         else -> throw Exception("Bound limits: list size must be in ${eAttr.lowerBound}..${eAttr.upperBound}")
6     }
7 }
8 override fun get$attrName(): List<$type>? {
9     return controller.getProperty("${eAttr.name}", $mapFunc)
10 }
```

Listing 6.6. Implementation template for the multi-valued attribute

6.3.3. Associations getter/setter generation

NMF-generator synthesizes an `Association` of the metamodel class into Kotlin getter, setter, and remover methods. All the templates for association generation rely on the `eRef` variable, that represents a concrete `Association` object of the metamodel. From that variable, we derive two more common variables that are used in all further templates for association generation:

- `refName` is the capitalized name of the association. In contrast, a `eRef.name` denotes the actual (not capitalized) name of the association.
- `type` denotes the type of the target element of the association. Also, in terms of NMF-generator, the `type` variable denotes an interface of the node controller (see Section 6.3.1).

According to the Ecore specification, associations can be either containment or reference. Also, an association can have different cardinalities like attributes. To distinguish between these cases, we use four different templates for code generation.

6.3.3.1. Reference association generation. Like for attributes, associations can be either single-valued or multi-valued (`upperBound = 1` or `> 1` respectively). For single-valued reference associations, we generate the three following methods that we put into interface of an object controller:

```
1 fun set$refName(v: $type)
2 fun unset$refName(v: $type)
3 fun get$refName(): $type?
```

Listing 6.7. Interface template for the single-valued association reference

The `upperBound 1` means that an element can have maximum one outgoing association. Thus, adding a reference means to set a value. The user creates an association from a source element to a target element by providing an object controller that represents the target element as a parameter for the setter method (line 1). We use the same logic for association removal with the unset function (line 2). The getter method returns the target object controller of the association (line 3). If there is no association set the get method returns `null`.

According to our naming approach, the names of methods for association manipulation depends on a name of the association of the metamodel. Therefore, in case when an association has a name in plural form (such as `vertices` association of `Graph` element of our GRAF DSL) we get a naming ambiguity in the generated code, such as `removeVertices(v: Vertex)` despite that we remove only a single association. To eliminate this mismatch in future we plan to provide a lexical analyzer for those cases. Then, NMF-generator would be able to transform a plural form of a noun into a singular one.

Listing 6.8 presents the template for generating the implementation for those three methods from Listing 6.7. We put the generated code into the implementation class of the object controller. So, the generated code can rely on the available `controller` field.

```
1 override fun set$refName(v: $type) {
2     if (controller.outRefCount.getOrDefault("${eRef.name}", 0) < $upperBound) {
3         controller.createOutRef("${eRef.name}", v)
4     } else throw Exception("UpperBound exceeded")
5 }
6 override fun unset$refName(v: $type) {
7     if (controller.outRefCount.getOrDefault("default_vertex", 0) > $lowerBound) {
8         controller.removeOutRef("${eRef.name}", v)
9     } else throw Exception("LowerBound exceeded")
10 }
11 override fun get$refName(): $type? {
12     val data = getOutConnectedNodes("${eRef.name}", 1) {
13         ${type}Neo4jImpl(it)
```

```

14 }
15 return if (data.isEmpty()) null else data[0]
16 }

```

Listing 6.8. Implementation template for the single-valued association reference

The setter and remover methods (lines 1–10) rely on `createOutRef` and `removeOutRef` methods of the underlying node controller respectively (the specification of the node controller is described in Section 5.2.2). The generated setter and remover methods take a target object controller (to create/remove the association with) as a parameter. Since each object controller implements an `INodeController` interface, it is a compatible parameter for the node controller functions (lines 3, 8). Also, we wrap the invocation of the underlying node controller method with a bounds check (lines 2–4, 7–9). As shown in Listing 6.8, setting more association references than the `upperBound` or removing fewer association referenes than the `lowerBound` rises an exception. Therefore, to reset an existing association the user must first remove it and then create a new one.

The getter method (lines 11–16) relies on `getOutConnectedNodes` of the underlying node controller. According to the specification (Fig. 5.2), this method returns a list of node controllers. Inside the implementation method, we map the returned node controller to a domain-specific object controller (line 13) and return the first element of the list or null if the list is empty (line 15). To perform the mapping we simply create a runtime domain-specific object by passing a node controller as a constructor parameter.

For multi-valued reference associations, we generate the three following methods that we put into an interface of the object controller:

```

1 fun add$refName(v: $type)
2 fun unset$refName(v: $type)
3 fun get$refName(limit: Int = 100): List<$type>

```

Listing 6.9. Interface template for the multi-valued association reference

The setter method starts with `add` prefix to highlight the possibility of creating multiple associations (line 1). Implementation of the remover method is exactly the same as in Listing 6.7. The getter method returns a list of object controllers instead of a single one (line 3).

```

1 override fun get$refName(limit: Int): List<$type> {
2     return controller.getOutConnectedNodes("${eRef.name}", limit) { controller ->
3         when (controller.label) {
4             for (t in allTypes) {
5                 "$t" -> ${t}Neo4jImpl(controller)
6             }
7             else -> throw Exception("Cannot cast to INodeController)
8         }
9     }
10 }

```

Listing 6.10. Implementation template for the multi-valued association reference

6.3.3.2. Containment association generation. For single-valued containment association, we generate the following methods:

```
1 fun insert$refName(): $type
2 fun remove$refName(v: $type)
3 fun get$refName(): $type?
```

Listing 6.11. Interface template for single-valued containment association

The key differences between a generated code for a reference and a containment association is that the setter method for the containment association creates and returns a target object controller (line 1). This way, the target object is always contained, by constructing in a source object. The implementation of the getter and remover methods remain the same as in Listing 6.8.

The `insert$refName` method relies on `createChild` function of a node controller. It returns a controller for the contained object. The `remove$refName` (i.e., remover) method relies on `removeChild` of a node controller. The method `get$refName` invokes `getConnectedNodes`. We recall that the specification of the node controller was described in Section 5.2.2.

For multi-valued containment association, we generate the following methods:

```
1 override fun set$refName(): $type {
2     return controller.createChild("${eRef.name}")
3 }
4 override fun remove$refName(v: $type) {
5     controller.removeOutRef("${eRef.name}", v)
6 }
7 override fun get$refName(): $type? {
8     val data = loadOutConnectedNodes("${eRef.name}", 1) {
9         ${type}Neo4jImpl(it)
10    }
11    return if (data.isEmpty()) null else data[0]
12 }
```

Listing 6.12. Implementation template for single-valued containment association

6.4. Inheritance handling

Special case must be taken into account when generating a class inheriting the other classes. In particular, the generated header and containment association pointing to it must be revised.

6.4.1. Header generating

If a metamodel presents a case of inheritance between classes, the generated API must conform to this situation. According to Ecore specification (Fig. 2.5), a metamodel class can have multiple supertypes. A supertype can be either interfaces, abstract, or even concrete classes. In Kotlin, an interface can extend multiple interfaces, but a class can extend at most one class. Therefore, our approach of splitting an object controller into the interface and the class implementing that interface

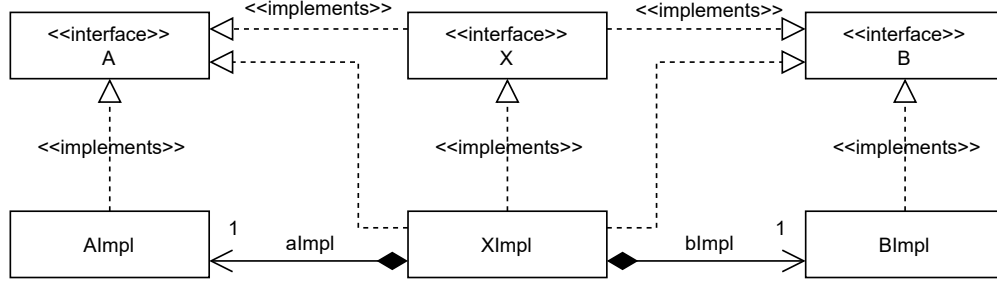


Fig. 6.3. Mapping multiple inheritance to object controller

allows us to correctly translate a case of multiple inheritance in the metamodel into Kotlin code. That separation allows us to provide an inheritance over composition using the interface segregation principle [31].

We now consider the code generation for the following example. Given a metamodel with three classes: the classes A and B are supertypes for the class X. Fig. 6.3 presents a structure of a generated code for a metamodel in UML class diagram. Here, we have three domain-specific object controllers. Interface X and class XImpl together presents an X class of the metamodel. Hence, the object controllers abide the target schema presented in Fig. 6.2. But, in Fig. 6.3 we omit the node controller related elements (such as `INodeEntity` interface and `NodeController` class) for brevity. According to the metamodel, X class extends class A and B. Therefore, interface X implements interfaces A and B. The implementation class XImpl also implements all the three interfaces. But, it delegates the functionality of A and B interfaces to the corresponding implementation classes using the composite reuse principle [31].

Hence, for generation purposes, a single inheritance is a special case of multiple inheritance. NMF-generator uses the same transformation to translate it into a Kotlin code. Therefore, following the example of the generated code for GRAF DSL presented in Listing 3.2, `CompositeVertex` interface extends `Vertex` interface (line 19). This ensures a direct semantics translation from a metamodel to a generated code. As a result, the user can pass object of either `Vertex` or `CompositeVertex` type as a parameter in the `setDefault_vertex` method (line 22) and respect the Liskov substitution principle [31].

6.4.2. Containment association generation

Inheritance in the metamodel also affects the code generation for containment associations. In Section 6.3.3 we described that a setter for an containment association also creates a target object controller. Let us consider the example of a metamodel in Fig. 6.4. The “child” of the `SourceClass` element is generalized under `TargetClass` type. The getter method `getAssociation(): TargetClass` of the `SourceClass` needs to create a concrete implementation class of an object controller which can be A, B, C, or `TargetClass`. Therefore, we provide a selection option for a user: it is possible to explicitly specify a desired type of the containment element providing an enum parameter to the function.

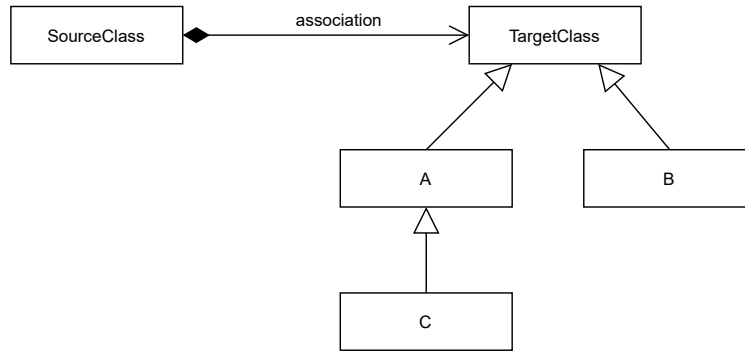


Fig. 6.4. Inheritance hierarchy example

For that, we additionally generate an enumeration that contains non-abstract classes of the inheritance chain. (in this case: TargetClass, A, B, and C). We then use this enumeration as a parameter for add function to explicitly specify what type of controller to create.

NMF-generator discover a list of subclasses for each class from the input metamodel and uses the following template to produce the enumerations:

```
1 enum class ${className}Type { ${subClasses} }
```

Listing 6.13. Template of the subtypes enumeration

6.5. Model manager generation

We generate a single Kotlin class that represents the model manager component of NMF-generator. Like other Kotlin classes, it consists of a header and a body.

6.5.1. Header generation

The header of the class consists of a constructor and methods whose naming do not depend on a concrete metamodel class. Listing 6.14 presents a template for generating a header of a domain specific model manager.

```

1 class ${modelName}ModelManager(dbUri: String, username: String, password: String) {
2     private val manager = IGraphManager.getDefaultManager(dbUri, username, password)
3     fun saveChanges() {
4         manager.saveChanges()
5     }
6     fun clearCache() {
7         manager.clearCache()
8     }
9     fun close() {
10        manager.close()
11    }

```

Listing 6.14. Template of the header of a model manager

The name of the model manager class consists of a capitalized `modelName` (that comes from the `EPackage` element of the metamodel) and a `ModelManager` suffix (line 1). For example, a model manager for our GRAF DSL has name `GraphModelManager`. We declare the class and provide a constructor that takes three parameters (line 1). Constructor parameters serve to instantiate an underlying graph controller (line 2). Functions `saveChanges`, `clearCache`, and `close` invoke their respective counterparts in the graph controller (lines 3–11).

6.5.2. Contents generation

The model manager class provides a set of methods for each non abstract class of the metamodel. We provided a detailed description for those methods in Section 6.1.2. To produce those methods, we use the following template:

```

1  fun create$className(): $className {
2      return ${className}Neo4jImpl(manager.createNode("$className"))
3  }
4  fun get${className}ById(id: Long): $className {
5      return ${className}Neo4jImpl(manager.loadNode(id, "$className"))
6  }
7  fun get${className}List(limit: Int = 100): List<$className> {
8      return manager.loadNodes("$className", limit) { ${className}Impl(it) }
9  }
10 fun unload(node: $className) {
11     manager.unload(node)
12 }
13 fun remove(node: $className) {
14     manager.remove(node)
15 }
```

Listing 6.15. Template of the contents of a model manager

The `className` is a capitalized name of the current metamodel class. The `manager` field was defined in Listing 6.14 and determines an underlying graph controller.

6.6. Domain specific editing with NMF

We now discuss the benefits and limitations of using the generated domain-specific API. In Listing 3.3 we have presented the example of using the GRAF API. This example demonstrated basic editing operations such as attribute modification (line 3), containment creation (lines 4–6), and reference association creation (line 7). In this example, we have a special case of a containment association in the metamodel: both `sub_vertices` and `vertices` containment associations point to the generalized `Vertex` class. Therefore, the content of a `Vertex` may be either `CompositeVertex` or another `Vertex`. The user must explicitly specify the concrete type of the contained object within the enum (lines 4–6). The enum was presented in Listing 3.2 (line 29).

The API calls can be easily integrated into a custom code within arbitrary workflows. For instance, the user can place the API calls into loops, **if-else** statements, or in custom functions. The API can be used in any JVM-based programming languages such as Java, Kotlin, or Scala.

The advantage of the domain-specific API is the rising level of abstraction. The user can focus more on editing the structure of his model rather than on the details provided by an editing tool. The generated API respects the domain rules within its constraints. At the moment of publication, NMF API supports only basic constraints supports (such as the lower and upper bounds). Advanced OCL constraints are not yet supported.

The essence of NMF is to provide storage and a manipulation facility for MDE models. This makes NMF compatible with the model transformations principle of MDE. Since NMF API provides CRUD operations, future model transformations could be built on top of them. Thus, the user can encode the transformations workflows using the API.

Chapter 7

Evaluation

In this Chapter, we evaluate NMF in two ways: brief feature comparison with Neo4EMF and preliminary performance analysis of NMF with respect to CRUD operations.

7.1. Comparing features of Neo4EMF and NMF

Since NMF and Neo4EMF provide similar functionality, we provide a comparison Table 7.1 to distinguish the features of these tools. The first major difference between these frameworks is the way to interact with the database. Since Neo4EMF supports multiple NoSQL databases, we focus on Neo4j to show the difference in commonly used technologies. NMF supports only remote Neo4j database via Cypher queries, while Neo4EMF works only with embedded database variant using Neo4j-ogm tool (low-level Java API) [29]. Theoretically, the remote database deployment allows configuring a distributed database (one database on multiple servers) to provide even more data scalability. Secondly, Neo4EMF is highly coupled with EMF API and Eclipse IDE. In contrast, NMF provides a modular utility with minimum dependencies. NMF is compatible with XMI model persistence format. NMF can import the models produced by EMF and Neo4EMF tools using NMF-loader.

7.2. Performance analysis

7.2.1. Experiments setup

In this Section, we test the performance of domain-specific editing operations. For that, we determine test scenarios for each of CRUD operations using the generated code of GRAF DSL (presented in Listing 3.2). Therefore, we test domain-specific API and implicitly the underlying NMF-editor module. We do not provide the testing of NMF-loader and NMF-generator modules. In this Section, we present an evaluation of a time performance and do not evaluate memory usage.

Experiments are executed on a 64 bits computer running Linux Ubuntu 18.04. Hardware elements are: Intel Xeon CPU E5-2620 v2 (2.10GHz), 64 GB RAM of DDR4 (1600 MHz). Software

	Neo4EMF	NMF
Run platform	Highly integrated in Eclipse IDE. Available to install as Eclipse plugins	Depends only on JVM
Code generation	Produces an API that transparently extends EMF's API	Produce a dedicated independent Kotlin API
Graphical modeling environment	-	-
MDE metamodel specification	Xcore	Ecore
Import models from XMI	+	+ (NMF-loader module)
Export models to XMI	+	-
OCL support	+	-
On-demand elements loading & caching	+	+
Data storage schema	Keeps a model alongside with its metamodel in the database, explicitly connecting each element of the model by a relationship with its meta-element	Keeps only a relevant model in the database
Supported back-ends	Default in-memory, MapDB, HBase, BerkeleyDB, Blueprints, Neo4j (through the Blueprints interface)	Neo4j
Neo4j deployment type	Embedded	Remote
Neo4j access interface	Neo4j-OGM tool	Cypher queries

Table 7.1. Features comparison of NMF and Neo4EMF

used: Java 11, Neo4j database 4.3, Gradle 6.4. The database, as well as NMF modules, are installed on the same computer.

7.2.2. Experiment results

For each test case, we gather the results of the experiments and present them in charts. For all the charts, the Y axis specifies the time in seconds, the X axis specifies the number of elements (specific for each chart).

7.2.2.1. Create operation. In this experiment, we evaluate the performance of the **Create** operation. We present the performance results for two cases: the creation of single (unconnected) objects and the creation of associations (connecting existing objects in the database).

We start by explaining the experiment setup for evaluating the performance of objects creation. In this experiment, we measure the time of creating a set of objects via domain-specific API and

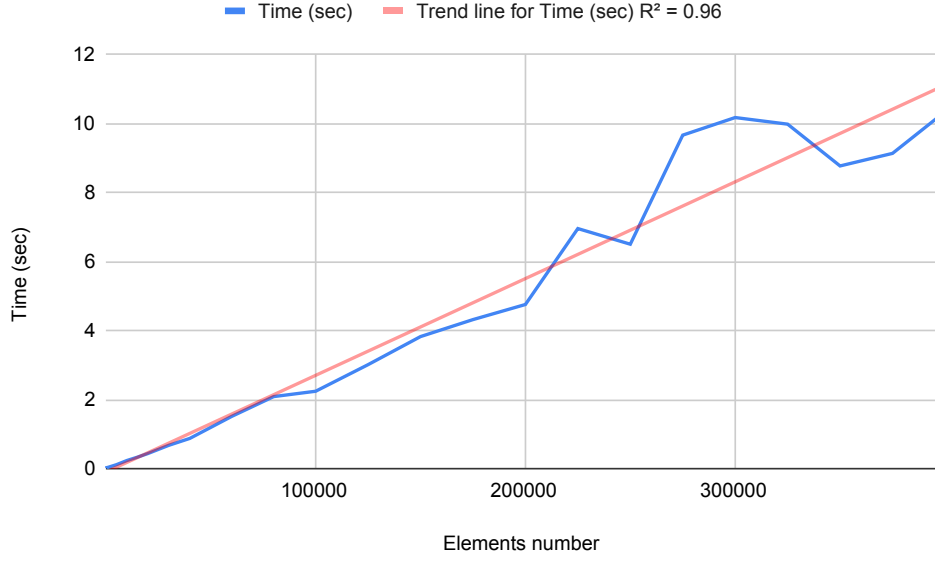


Fig. 7.1. Create model objects performance

uploading them to the database. The size of a set (i.e., count of objects to create) is the variable parameter for this experiment. We determine the first set as 1000 objects, increasing each next set by 1000 elements. The final set in this experiment is 400000 objects. We repeat the evaluation 30 times for each test set, calculate the average, and put it in the chart. This technique allows to calibrate the result.

The detailed workflow of elements creation was described in Section 5.3.1. We recall that NMF-editor keeps all the created elements in memory (in buffer) before committing them to the database. While performing the commit, NMF-editor can split all the buffered elements into chunks and upload them in several transactions to respect the bandwidth. The default size of the chunk for nodes creation is 20000 elements. Therefore, NMF-editor uploads the final set of 400000 objects in 20 transactions.

Fig. 7.1 demonstrates the performance of objects creation. We can see the linear time complexity (the blue line) that increases with the input size. We also put in the chart the trend-line to prove the trend. The chart demonstrates that NMF can create more than 300000 objects in just 10 seconds. We achieved this result by using the APOC library in conjunction with parameterized queries.

Also, we conduct the second experiment to measure the creation time of the domain-specific associations and uploading them to the database. Before the experiment, we pre-create two objects and load them into the database. Next, we create a certain number of associations between the objects using the corresponding domain-specific object controllers. As well as for objects creation testing, we form test sets from 1000 to 400000 associations (with step of 1000) and upload the sets to the database. Also, we repeat the evaluation of each test set 30 times to calibrate the result. The **Creator** component of NMF-editor provides the bandwidth parameter by default set to 10000: the

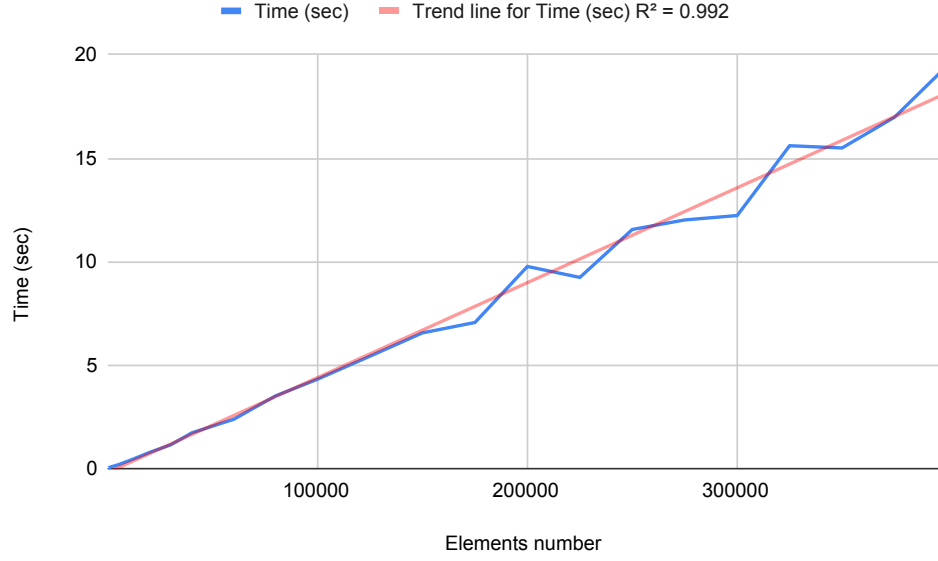


Fig. 7.2. Create reference associations performance

number of associations to create in a single transaction. Hence, the last set of 400000 associations will be uploaded in 40 transactions.

Fig. 7.2 presents a performance of association creation. The chart demonstrates linear time increasing.

7.2.2.2. Update operation. In this experiment, we measure a time of **Update** operation of attributes of the model objects in the database. Before the experiment, we create in the database 400000 objects. Also, we keep tracking the corresponding controllers for those objects on the client-side. Next, we determine sets of objects to update (the first set is 1000 up to 400000 with a step of 1000). For each object of the set, we update one attribute using the corresponding object controller and then upload the entire set of updates to the database. Therefore, the set size (i.e., the count of objects to update) is the variable parameter for this experiment.

The detailed workflow of elements update operation was described in Section 5.3.2 We recall that NMF-editor accumulates the updates in the cache before a client explicitly commits them to the database. While performing the commit, NMF-editor can split all the buffered elements into chunks and upload them in multiple transactions. This logic is the same as **Creator** component provides. The default size of a chunk is 20000 elements. Hence, the last set of 400000 elements will be uploaded in 20 transactions.

Fig. 7.3 presents the experiment result. The chart demonstrates constant time complexity (with weak increasing trend). Updating 400000 objects in the database take about 1.05 seconds. When NMF keeps tracking the object controllers of persisted nodes in the cache, this guarantees instant nodes lookup by internal database IDs. Hence, a relatively small amount of objects does not affect the updating performance in terms of time. In the future, we plan to redo this experiment with even bigger data sets (at least 10^5 + objects to update).

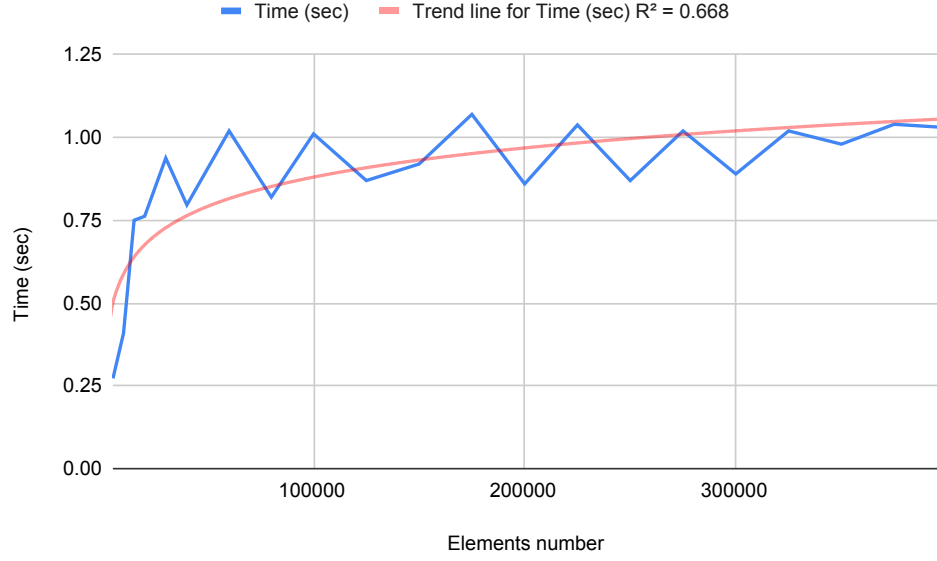


Fig. 7.3. Update objects performance

7.2.2.3. Remove operation. In this experiment, we evaluate the performance of the **Remove** operation of model objects in the database. We present the performance results for two cases: removing contained objects (i.e., in-depth cascade delete) and removing reference associations.

We start by explaining the experiment setup for objects removal. First, we create the model in the database with the following structure: a root object contains one other object, which also has one contained object, and so on. Thus, our structure has a view of a chain where all the objects are connected by a containment association. We keep only the root object controller in the NMF cache. In this experiment, we remove the root object from the NMF client, expecting the entire chain to be removed from the database. Therefore, we delegate to the database discovering and removing all the contained objects together with the containment associations. The number of objects in the chain (i.e., depth) is a variable parameter in this experiment. We form the chains of 1000 objects up to 300000 (each next chain has 1000 more objects than the previous). Since we repeat each test set 30 times to get the average and calibrate the result, we create 30 same chains in the database to evaluate a single test case. Then we successively remove the root objects, measure the removal time of each chain, and calculate the average.

Fig. 7.4 presents the experiment results. We can see the constant time complexity. This is due to our mapping schema, which allows the overall algorithm of nodes discovering and removing to be performed directly in the database. The detailed workflow of elements removal operation was described in Section 5.3.3. We recall, that internally NMF-editor passes to the database only one parameter: the ID of the root object. In this thesis, we present only in-depth objects removal, but in the future, we plan to perform also an evaluation of in-breadth placed objects removal.

Also, we conduct the second experiment to measure the time of associations removal. Before the experiment, we create in the database two objects, keeping in the cache their controllers. Next, we prepare the data to be removed: we create a certain number of reference associations between

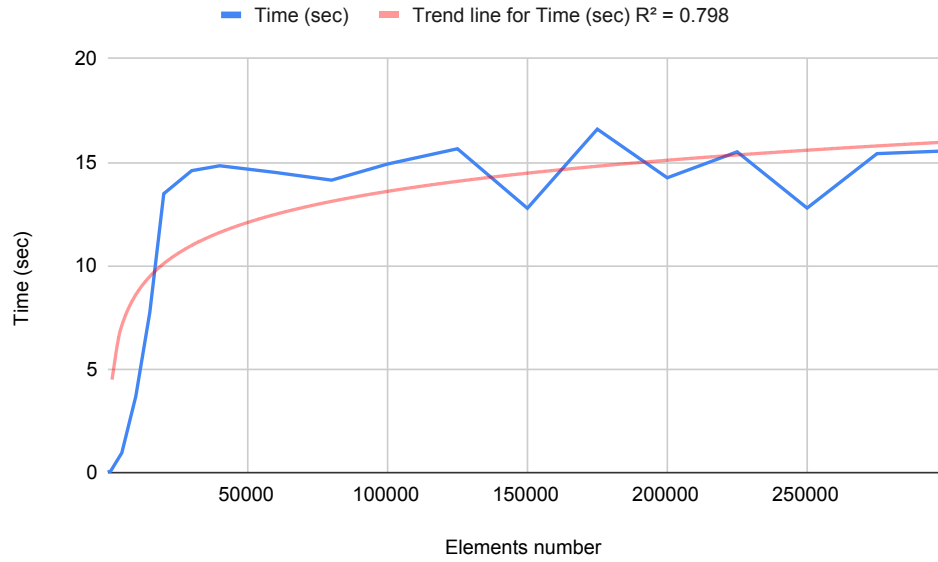


Fig. 7.4. Performance of removing objects in depth

the objects in the database. The number of associations to remove is a variable parameter in this experiment. We determine association sets as usual: the first set has 1000 associations to remove, the last one has 400000 (with the step of 1000 associations). Also, we repeat the evaluation 30 times to calculate an average. Therefore, before the first test set, we create $1000 \times 30 = 30000$ associations. Then, we perform 30 removings (each time 1000 associations), calculate the average, and put the result in the chart. Next, before the second test set of 2000 elements, we create $2000 \times 30 = 60000$

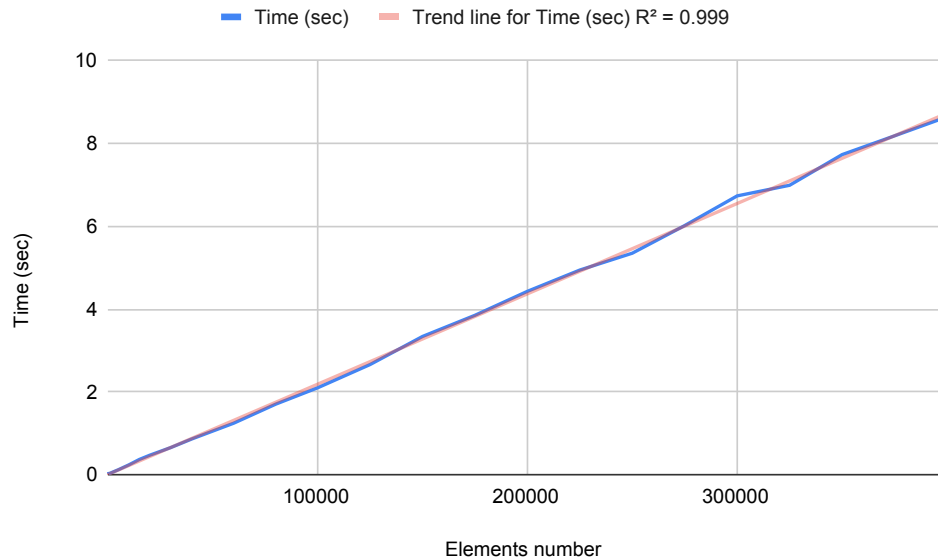


Fig. 7.5. Performance of reference associations removal

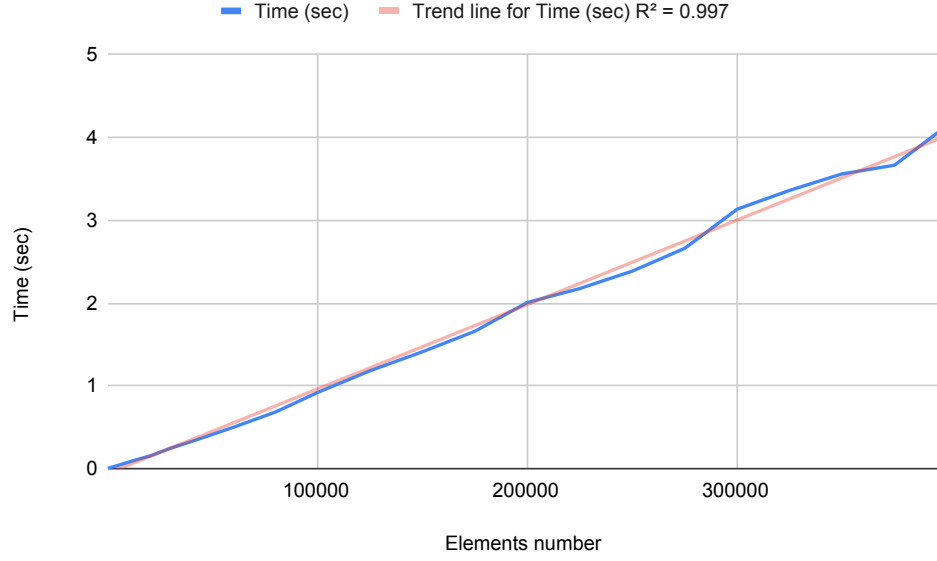


Fig. 7.6. Performance of reading related objects

elements. Before the final test set of 400000 elements, we must create $400000 * 30 = 12000000$ associations. Fig. 7.5 presents the experiment results. We can see the linear time complexity.

7.2.2.4. Read operation. In this experiment, we measure the time of **Read** operation of objects from the database. Before the experiment, we create in the database one root **Graph** object that contains 400000 **Vertex** objects. Then, we keep tracking only the root object controller on the client-side and use it to load the related objects from the database. The variable parameter in this experiment is the number of objects to load that we explicitly indicate on each iteration (from 1000 to 400000 with step pf 1000). The measured workflow includes loading a certain number of nodes from the database and mapping them to domain-specific object controllers. Also, we repeat each iteration 30 times to calibrate the result. The reading process was presented in detail in Section 5.3.4. We recall that the **Reader** component of NMF-editor does not use buffering or delaying: it returns the result instantly.

Fig. 7.6 presents the experiment result. The chart demonstrates linear time increasing. This means that the reading performance of related objects depends on the number of objects to load. In the chart, we can see that NMF is able to lookup and load 400000 objects in about 5 seconds.

7.3. Discussion

In this thesis, we have presented NMF – a modeling framework to operate on ultra-large MDE models. NMF aims to solve the MDE model scalability problem. This framework adapts the Neo4j database for modeling needs and provide for a modeler common MDE functionality. We have explained the benefits of each NMF module across the Chapters of this thesis. But, at the moment of publication, NMF has some incomplete and missing features.

- NMF-loader relies on standard EMF API to read an input model in **XMI** format. Therefore, this module does not use a partial model loading technique since the EMF API parses the whole input model before proceeding.
- NMF supports only one working client at the time. There is no possibility to collaborate for multiple clients simultaneously on a single model.
- NMF does not provide a graphical modeling environment, so the client could visualize models and results of the editing operations.
- NMF does not provide an advanced OCL support in the generated code.
- NMF does not provide a possibility to export a model into **XMI** format.

All the implemented features together with our plans allow NMF to improve the state of the art of existing tools in MDE.

Chapter 8

Conclusion

8.1. Summary

In this thesis, we have presented a new modeling framework to manipulate MDE models on graph databases. The framework includes a loader module to load existing models and metamodels in Neo4j database. Also, the framework includes a library to edit models seamlessly located locally in memory and in the database. Finally, we provide a domain-specific API generated for a given metamodel to facilitate interaction for a domain expert.

8.2. Outlook

In the future, we plan to make an in-depth performance comparison with existing frameworks such as Neo4EMF [8, 28]. Also, we would like to conduct more experiments to prove the scalability of NMF. Finally, we hope such a framework can be extensively applied in industrial cases.

References

- [1] V. García Díaz, E. Núñez Valdez, J. Espada, B. Pelayo García-Bustelo, J. Cueva Lovelle, and C. Marín, “A brief introduction to model-driven engineering,” vol. 18, pp. 127–142, 04 2014.
- [2] I. O. F. STANDARDIZATION, “Iso/iec 25010: Systems and software engineering-systems and software quality requirements and evaluation (square),” 2011.
- [3] R. France and B. Rumpe, “Model-driven development of complex software: A research roadmap,” in *Future of Software Engineering (FOSE '07)*, pp. 37–54, 2007.
- [4] Eclipse Foundation, “Eclipse modeling project.” <https://www.eclipse.org/modeling>. Last access: 2021-12-10.
- [5] Eclipse Foundation, “Eclipse modeling framework.” <https://www.eclipse.org/modeling/emf>. Last access: 2021-12-10.
- [6] V. Sousa, E. Syriani, and M. Paquin, “Feedback on how mde tools are used prior to academic collaboration,” in *Symposium On Applied Computing*, (Marrakesh), pp. 1190–1197, ACM, apr 2017.
- [7] G. Varró, K. Friedl, and D. Varro, “Implementing a graph transformation engine in relational databases,” *Software and Systems Modeling*, vol. 5, pp. 313–341, 09 2006.
- [8] A. Benelallam, A. Gómez, G. Sunyé, M. Tisi, and D. Launay, “Neo4emf, a scalable persistence layer for emf models,” in *Modelling Foundations and Applications* (J. Cabot and J. Rubin, eds.), (Cham), pp. 230–241, Springer International Publishing, 2014.
- [9] A. Vukotic, N. Watt, T. Abedrabbo, D. Fox, and J. Partner, *Neo4j in Action*. USA: Manning Publications Co., 1st ed., 2014.
- [10] OGM, “Mof 2 xmi mapping specification version 2.5.1.” <https://www.omg.org/spec/XMI/2.5.1>, 2015.
- [11] “Neo modeling framework github repository.” <https://github.com/geodes-sms/NeoModelingFramework>. Last access: 2021-12-13.
- [12] A. Kheir, H. Naja, M. C. Oussalah, and K. Tout, “From Viewpoints and Abstraction Levels in Software Engineering Towards Multi-Viewpoints/Multi-Hierarchy in Software Architecture.,” in *The Eighth International Conference on Software Engineering Advances*, (Venice, Italy), p. 478, Oct. 2013.
- [13] D. Di Ruscio, R. Eramo, and A. Pierantonio, “Model transformations,” in *Formal Methods for Model-Driven Engineering*, vol. 7320 of *LNCS*, pp. 91–136, Springer Berlin Heidelberg, 2012.
- [14] H. C. Purchase, L. Colpoys, D. Carrington, and M. McGill, *UML Class Diagrams: An Empirical Study of Comprehension*, pp. 149–178. Boston, MA: Springer US, 2003.
- [15] M. Richters and M. Gogolla, *OCLE: Syntax, Semantics, and Tools*, pp. 42–68. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002.
- [16] L. Bettini, *Implementing Domain Specific Languages with Xtext and Xtend - Second Edition*. Packt Publishing, 2nd ed., 2016.

- [17] Eclipse Foundation, “Graphical modeling framework.” <https://www.eclipse.org/gmf-tooling>. Last access: 2021-12-11.
- [18] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*. Assison-Wesley Professional, 2nd ed., 2009.
- [19] “Neo4j java driver.” <https://neo4j.com/developer/java>. Last access: 2021-12-13.
- [20] A. Mazak, S. Wolny, and M. Wimmer, *On the Need for Data-Based Model-Driven Engineering*, pp. 103–127. Cham: Springer International Publishing, 2019.
- [21] R. Wei, D. Kolovos, A. García-Domínguez, K. Barmpis, and R. Paige, “Partial loading of xmi models,” pp. 329–339, 10 2016.
- [22] G. Daniel, G. Sunyé, and J. Cabot, “Prefetchml: A framework for prefetching and caching models,” in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, MODELS ’16, (New York, NY, USA), p. 318–328, Association for Computing Machinery, 2016.
- [23] D. Seybold, J. Domaschka, A. Rossini, C. B. Hauser, F. Griesinger, and A. Tsitsipas, “Experiences of models@run-time with emf and cdo,” in *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2016, (New York, NY, USA), p. 46–56, Association for Computing Machinery, 2016.
- [24] K. Barmpis and D. Kolovos, “Hawk: Towards a scalable model indexing architecture,” in *Proceedings of the Workshop on Scalability in Model Driven Engineering*, BigMDE ’13, (New York, NY, USA), Association for Computing Machinery, 2013.
- [25] A. Garcia-Dominguez, K. Barmpis, D. S. Kolovos, M. A. A. da Silva, A. Abherve, and A. Bagnato, “Integration of a graph-based model indexer in commercial modelling tools,” in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, MODELS ’16, (New York, NY, USA), p. 340–350, Association for Computing Machinery, 2016.
- [26] “Hibernate orm.” <https://hibernate.org/orm>. Last access: 2021-12-10.
- [27] R. Wang, Z. Yang, W. Zhang, and X. Lin, “An empirical study on recent graph database systems,” in *Knowledge Science, Engineering and Management* (G. Li, H. T. Shen, Y. Yuan, X. Wang, H. Liu, and X. Zhao, eds.), (Cham), pp. 328–340, Springer International Publishing, 2020.
- [28] G. Daniel, G. Sunyé, A. Benelallam, M. Tisi, Y. Vernageau, A. Gómez, and J. Cabot, “Neoemf: A multi-database model persistence framework for very large models,” *Science of Computer Programming*, vol. 149, pp. 9–14, 2017. Special Issue on MODELS’16.
- [29] “Neo4j - ogm (object graph mapper).” <https://neo4j.com/developer/neo4j-ogm>. Last access: 2021-12-13.
- [30] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, 1994.
- [31] R. C. Martin, *Clean Architecture: A Craftsman’s Guide to Software Structure and Design*. Robert C. Martin Series, Boston, MA: Prentice Hall, 2017.

Appendix A

NMF Manual

Neo Modeling Framework (NMF) is an open-source set of tools primarily designed to manipulate ultra-large datasets in the **Neo4j database**. NMF implements Object Graph Mapping (OGM) technique that allows a remote client to operate on data directly in the graph database. Hence, a client application can delegate to the database handling of a large amount of data that exceeds the client's RAM capacity through NMF. Therefore, the client can select and load to memory only needed elements but not load an entire dataset from the storage. Also, NMF optimizes writing operations (i.e., CREATE, UPDATE, and DELETE), previously grouping the changes in the cache and performing them later in a transactional way. These features optimize I/O performance for editing large datasets in the Neo4j database providing better efficiency and scalability.

Inspired by Eclipse Modeling Framework (EMF), NMF treats any dataset in terms of *EMF model*. EMF is the famous modeling tool for Model-Driven Engineering (MDE) software development methodology. To find out more about EMF and MDE, please refer to:

- An introduction to MDE: <http://www.scielo.org.co/pdf/tecn/v18n40/v18n40a11.pdf>
- <https://www.eclipse.org/modeling/emf>
- <https://eclipsesource.com/blogs/tutorials/emf-tutorial>

NMF endows the datasets with modeling concepts of EMF models. NMF editing operations follow the editing logic of EMF models. In NMF, any model (i.e., dataset) must conform to some meta-model – the structure that describes a model itself. In particular, NMF relies on Ecore metamodel presented in Fig. 2.5.

NMF aims to resolve the EMF model scalability problem by using the graph database as an underlying storage. We use Neo4j graph database because the structure of EMF model has a nature of a graph.

A.1. Prerequisites

Before using NMF modules a user must have:

- A running Neo4j database instance (either local or remote) with **APOC** plugin installed. We recommend using **Neo4j Desktop** to set up the database environment.

- JRE (8+) installed. Run in terminal: `java -version` to check the JRE installation.
- Gradle (6+) build tool installed (only for manual building)

We recommend using IntelliJ IDE since it provides integration for Gradle and JRE.

A.2. NMF architecture

NMF consists of three modules: `NMF-editor`, `NMF-loader`, and `NMF-generator`. These modules are designed to achieve the following goals:

- `NMF-loader`: stores existing EMF models in the Neo4j database
- `NMF-editor`: performs editing operations (i.e., `CREATE`, `UPDATE`, `REMOVE`, `READ`) over the models directly in the database
- `NMF-generator`: provides a domain-specific API to edit the models

The overall NMF architecture within dependencies between the modules are presented in Fig. 3.1.

A.2.1. NMF-editor

`NMF-editor` is a core module that provides an interaction with the Neo4j database. `NMF-loader` is packaged as an executable jar file and can be used as runtime dependency. The jar can be found in the Github release section. The test sandbox can be found in `NMF-editor test` file.

```

1 val dbUri = "bolt://localhost:7687"
2 val username = "neo4j"
3 val password = "admin"
4 val graphManager = GraphManager(dbUri, username, password) // init a connection with the DB
5 val n1 = graphManager.createNode("Node1") // n1 is a node controller
6 val n2 = graphManager.createNode("Node2")
7 val n3 = n1.reateChild("ref2", "Node3")
8 graphManager.saveChanges() // commit updates to the storage
9 n1.createOutRef("ref1", n2) // controllers remain interactable after the commit
10 n1.putProperty("property", "Test property")
11 graphManager.saveChanges() // commit new changes
12 n1.remove() // remove the node 'n1' within its child 'n3' (cascade delete)
13 graphManager.saveChanges()
14 graphManager.close()

```

Listing A.1. NMF-editor API usage example

The modification operations are applied in a transactional way on `graphManager.saveChanges()` function invocation.

A.2.2. NMF-loader

This module provides a model storing facility. `NMF-loader` can export an existing EMF model provided in XMI format into the Neo4j database. The input model must be provided in a file. Only `*.xmi` and `*.ecore` file extensions are supported.

NMF-loader is packaged as an executable jar file and can be used from a command line. Download the latest [NMF-loader](https://github.com/geodes-sms/NeoModelingFramework/releases/tag/v1.0) release and run the following command in terminal: `java -jar <NMF_LOADER_PATH> -help`. The output should be as follows:

```

1 java -jar <NMF_LOADER_PATH> --help
2 -h,--host <HOST:PORT> Database host address with port used to create bolt connection. Example:
   -h 127.0.0.1:7687
3 -m,--model <PATH> path to model file to be loaded
4 -u,--user <arg> Database auth: username
5 -p,--password <arg> Database auth: password

```

Listing A.2. NMF-loader usage example

NMF-loader requires 4 parameters:

- Database host (-h) address of Neo4j database to establish a connection with. Both local and remote addresses are supported. By default, Neo4j local installation is on `http://127.0.0.1:7687`
- Database credentials: a username (-u) and a password (-p)
- Model path (-m): an actual location of the model to load

NMF-loader can proceed both model instances and metamodels (a model of any level M_i according to Fig. 2.4. To correctly process a model of level M_1 , NMF-loader requires a metamodel the model conforms to. For that, a model instance stored in XMI format must have linked its metamodel location in the xmi header as follows:

```

1 ...
2 xsi:schemaLocation="EPACKAGE_NAME PATH_TO_METAMODEL"
3 ...

```

Listing A.3. NMF-loader usage example

We list GRAF model instance with a valid `xsi:schemaLocation` attribute in Appendix C. An example of valid models as well as metamodels can be found in `Models` directory in NMF github repository [11].

A.2.3. NMF-generator

NMF-generator is a code generation facility. It produces a set of Kotlin classes (domain-specific API) for editing a specific model. Unlike a generic API of NMF-editor, the produced API is conceptually closer to the domain rather than to data. The generated API relies on NMF-editor to interact with data in the Neo4j database.

NMF-loader is packaged as an executable jar file and can be used as runtime dependency. NMF-generator takes a metamodel in Ecore format as an input and produces a set of files with Kotlin code. By default, NMF-generator outputs the result API in `domain-specific editor` directory which represents a module with preconfigured dependencies.

```
1 java -jar <NMF_LOADER_PATH> --help
2   -mm Path of the input metamodel
3   -o,--output Output directory
```

Listing A.4. NMF-generator usage example

Example of the generated domain-specific API for Graf DSL: <https://github.com/geodes-sms/NeoModelingFramework/tree/master/modelEditor/src/main/kotlin/geodes/sms/nmf/editor/graph>

Metamodel examples: <https://github.com/geodes-sms/NeoModelingFramework/tree/master/EmfModel/metamodel>

Appendix B

Graf metamodel in Ecore format

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <ecore:EPackage xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
3   xmlns:xmi="http://www.omg.org/XMI"
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5   xmi:version="2.0" name="graph" nsURI="graph" nsPrefix="graph">
6   <eClassifiers xsi:type="ecore:EClass" name="Graph">
7     <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
8       eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString" />
9     <eStructuralFeatures xsi:type="ecore:EReference" name="vertices"
10       eType="#//Vertex" upperBound="-1" containment="true" />
11   </eClassifiers>
12   <eClassifiers xsi:type="ecore:EClass" name="Vertex">
13     <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
14       eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString" />
15     <eStructuralFeatures xsi:type="ecore:EAttribute" name="id"
16       eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EInt" id="true" />
17     <eStructuralFeatures xsi:type="ecore:EAttribute" name="is_initial"
18       eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EBoolean" />
19     <eStructuralFeatures xsi:type="ecore:EReference" name="edge"
20       eType="#//Vertex" lowerBound="0" upperBound="5" containment="false" />
21   </eClassifiers>
22   <eClassifiers xsi:type="ecore:EClass" name="CompositeVertex" eSuperTypes="#//Vertex">
23     <eStructuralFeatures xsi:type="ecore:EAttribute" name="capacity"
24       eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EInt" />
25     <eStructuralFeatures xsi:type="ecore:EReference" name="default_vertex"
26       eType="#//Vertex" upperBound="1" />
27     <eStructuralFeatures xsi:type="ecore:EReference" name="sub_vertices"
28       eType="#//Vertex" upperBound="-1" containment="true" />
29   </eClassifiers>
30 </ecore:EPackage>
```


Appendix C

Graf model instance serialized in XMI format

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <graph:Graph
3   xmlns:graph="graph"
4   xmlns:xmi="http://www.omg.org/XMI"
5   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6   xmi:version="2.0"
7   xsi:schemaLocation="graph ./Graph.ecore"
8   name="G1">
9   <vertices xsi:type="graph:CompositeVertex" id="1" name="V0" default_vertex="2">
10     <sub_vertices name="V1" id="2" edge="3" />
11     <sub_vertices name="V2" id="3" />
12   </vertices>
13 </graph:Graph>
```

The listing presents a model from Fig. 3.3a. Line 7 specifies that a metamodel file `Graph.ecore` (that provides xml namespace `graph`) is located in the same directory alongside the original model instance file. This attribute must be specified to allow parsers automatically link the metamodel.