GUI-based Automated Generation of Neo4j Cypher Queries for Candidate Patterns and Parallelization Patterns

Master’s Thesis

Erik Koppenhagen

September 14, 2016

KIEL UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE
SOFTWARE ENGINEERING GROUP

Advised by: Prof. Dr. Wilhelm Hasselbring
M. Sc. Christian Wulf
Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

__________________________

ii
Abstract

Multicore CPUs and Hyperthreading allow programs to execute multiple operations at the same time. Nonetheless, sequential programs still exist and continue to be developed. Software engineers created several tools for automated and semi-automated parallelization of sequential programs. Subsequently, parallelizing sequential programs can be achieved in different ways. For example, the source code can be examined for constructs like loops or method calls. Our approach transforms source code into system dependency graphs and stores them in Neo4j graph databases. We use pattern matching to find parallelizable structures and transform the structure of the graph. We provide a graph editor for this purpose. Then, we transform the resulting SDG back to code.

To reach this goal, we translate predefined, graph-based patterns to Neo4j Cypher queries. At first, we give an overview of basic elements and their translation. Then, we translate operations of a protocol, which holds records of changes to patterns. Combining these approaches, we develop a translator and integrate it in our graph editor. Then, we evaluate the selected translation results.

Our results show that the translator complies with our requirements. It supports the complete functionality of our graph editor. We suggest additional features, which can be added in both the translator and the editor.
Contents

1 Introduction 1
   1.1 Motivation ................................................. 1
   1.2 Context .................................................. 1
   1.3 Goals ..................................................... 3
      1.3.1 Query Generation from Visualized Candidate Patterns .... 3
      1.3.2 Query Generation from Recorded Parallelization Patterns .... 3
      1.3.3 Performance Tuning for Cypher Queries ................... 3
      1.3.4 Graph Editor Improvements for Pattern Definition .......... 4
      1.3.5 Evaluation of Feasibility and Correctness ................ 4
   1.4 Document Structure ........................................ 4

2 Foundations and Technologies 5
   2.1 Graph Theory ............................................. 5
      2.1.1 Property Graphs ...................................... 5
      2.1.2 Dependency Graphs .................................... 6
      2.1.3 Program Dependency Graphs ............................ 6
      2.1.4 System Dependency Graphs ............................ 7
   2.2 The Graph Database Neo4j ................................. 11
      2.2.1 Neo4j’s Graph Model .................................. 11
      2.2.2 The Query Language Cypher .......................... 12
   2.3 The Graph Editor .......................................... 14
      2.3.1 The User Interaction Protocol ......................... 15
      2.3.2 The Base Model ....................................... 17

3 Reverse Engineering of the Graph Editor 19

4 Improvement of the Graph Editor 23
   4.1 Implementation of our System Dependency Graph Model .......... 23
   4.2 Renaming of User Interaction Protocol Operations .............. 23
   4.3 Label Updates for Type Changes ................................ 23
   4.4 Implementation of a Multi Edge ................................ 24
   4.5 Concept and Implementation of Forbidden Elements ............. 24
      4.5.1 Different Kinds of Restrictions ........................ 24
      4.5.2 Implementation ....................................... 25
   4.6 Workaround for not Firing Graph Events ........................ 26
   4.7 Changes to the Context Menu ................................ 27
## Contents

4.8 Manual for the Creation of Patterns ........................................... 28

5 Translation of the Candidate Pattern Graph .................... 29
  5.1 Translation of Single Elements ......................................... 29
      5.1.1 Nodes ................................................................ 29
      5.1.2 Edges ................................................................ 31
      5.1.3 Key-Value Pairs .................................................. 33
  5.2 Translation of Advanced Constructs ................................. 34
      5.2.1 Key-Value Pairs .................................................. 34
      5.2.2 Multiple Edges ..................................................... 36
  5.3 Translation of Forbidden Elements ................................. 43
      5.3.1 Forbidden Properties ........................................... 43
      5.3.2 Forbidden Types .................................................. 44
      5.3.3 Forbidden Edges .................................................. 46

6 Translation of the User Interaction Protocol .................. 49
  6.1 Translation of the Protocol ............................................... 49
      6.1.1 Adding Elements ............................................... 50
      6.1.2 Removing Elements ............................................ 50
      6.1.3 Editing Properties .............................................. 50
      6.1.4 Editing Types ..................................................... 51
  6.2 Special Behavior of the User Interaction Protocol ............ 52
      6.2.1 Removing Connected Elements .............................. 52
      6.2.2 Removing Elements with Key-Value Pairs ................. 53
  6.3 Optimization of the User Interaction Protocol ................ 55
      6.3.1 Optimization of Property Operations ...................... 55
      6.3.2 Optimization of Edge Operations ........................... 57
      6.3.3 Optimization of Node Operations ........................... 58

7 Architecture of the Translator ............................................. 61
  7.1 The Translator ............................................................. 61
      7.1.1 GUI Package .................................................... 63
      7.1.2 Languages Package ............................................ 64
  7.2 Integration into the Graph Editor ................................... 68

8 Evaluation of the Candidate Pattern Translation ............... 71
  8.1 Approach and Methodology ............................................. 71
  8.2 Candidate Pattern for our Evaluation .............................. 71
  8.3 Results and Discussion .................................................. 74
  8.4 Comparison of the Cypher Queries ................................. 76
  8.5 Threads to Validity ....................................................... 77
Chapter 1

Introduction

In the era of multicore CPUs and Hyperthreading, sequential programs can get a boost in performance through parallelization. Considering the additional effort in programming, even nowadays developers eschew parallelism. Problems like dead/live locks and race conditions have to be considered. Trying to manually change the source code consumes a lot of time and is prone to error. Also the boost in performance might not justify the costs.

1.1 Motivation
Knowing the problems, developers created several tools for automated and semi-automated parallelization of sequential programs. Automated tools may produce false-positive results or can not achieve much performance boost, because of generally applicable algorithms. Having no limitations in generality, semi-automated tools may find more parallelizing structures, but require user interaction. [Martino and Iannello 1994]

1.2 Context
Subsequently parallelizing of sequential programs can be achieved in different ways. For example, the source code can be examined for constructs like loops or method calls. The approach of the Software Engineering Group at Kiel University transforms source code into system dependency graphs. This is done by static and dynamic analyses. A database stores graph structures, which are able to be parallelized. Then, pattern matching is used to find candidates in the system dependency graph. This approach, developed by Christian Wulf, tries to overcome the limitations mentioned. [Wulf 2014]

Figure 1.1 shows the parallelization process of Wulf’s approach. Java2Neo4j1, which bases on the Soot framework, transforms a sequential program in a static analysis to a system dependency graph (S1. See Section 2.1.4 for details). We use a Neo4j graph database to store the graph (see Section 2.2 for details). A dynamic analysis by Kieker2 gathers further information at runtime, such as the number of invocations of specific methods (S2). This data is used to enrich the system dependency graph. Afterwards, a Parallelism Plan is created (S3). This plan is a sorted list of program sections. These sections would significantly increase the performance, if parallelized. Then, the system dependency graph is checked in these sections for parallelizable patterns (S4). Wulf’s approach provides a set of so called Candidate Patterns, which model certain constellations within a system dependency graph.

---

1http://build.se.informatik.uni-kiel.de/chw/sootexample
2http://kieker-monitoring.net/
1. Introduction

Each of these graph patterns represents a part of code, which potentially can be parallelized. For each Candidate Pattern there is at least one Parallelization Pattern, created by a parallelization expert. Semantically they represent the same application behavior, but a Parallelization Pattern represents parallel source code. The visual transformation of a Candidate Pattern to a Parallelization Pattern is recorded and saved to a User Interaction Protocol. After finding a matching Candidate Pattern, the developer decides whether to parallelize it or not (S5). This will be repeated until the parallelism plan is processed or the user wants to quit the process. Afterwards, the code of the parallel program is generated from the new system dependency graph (S6). [Wulf 2014]
1.3 Goals

Our goal is to develop a translation mechanism, which transforms the Candidate Patterns into queries for the system dependency graph. This enables the pattern matching. Additionally, the creation and editing of Candidate Patterns should be improved. Therefore, we analyze and upgrade our tool for pattern creation.

1.3.1 Query Generation from Visualized Candidate Patterns

To create Candidate Patterns and Parallelization Patterns, we use a custom graph editor (see Section 2.3 for details). Its underlying Base Model defines and visualizes the Candidate Patterns.

We use Neo4j to store the system dependency graph, which is further explained in Section 2.2. Neo4j has an own query language, Cypher, to find certain information or structures within a graph.

Our goal is to create an automated translation mechanism from the Base Model to a Cypher query. This allows finding corresponding structures in the system dependency graph, while we are able to model Candidate Patterns with the graph editor.

Implementation of a Mechanism to Explicitly Exclude Elements

Some Candidate Patterns may only apply if some restrictions are defined. For example, a node must not have a specific type. The syntax of Cypher allows to explicitly exclude certain elements or constructs. Thereby, translations can be more specified to the requirements of the user. Explicitly excluding elements or constructs results in a more possibilities to design patterns.

This kind of elements are not yet implemented in the graph editor. Additionally, we have to consider the explicitly excluded elements in the translation mechanism.

1.3.2 Query Generation from Recorded Parallelization Patterns

Also we require a translation mechanism for Parallelization Patterns to a Cypher query. The User Interaction Protocol of a Candidate Pattern represents a Parallelization Pattern. Parallelization Patterns change the system dependency graph instead of checking for certain constructs. Therefore, their translation differs from the translation of Candidate Patterns.

1.3.3 Performance Tuning for Cypher Queries

Neo4j Cypher queries follow a certain syntax. Automated generation of code may lead to complex expressions. Using special constructs of the query language, these expressions may be simplified. Additionally, some patterns may need special constructs. Perhaps, code can be shortened before the translation. This may apply to Candidate Patterns and the User Interaction Protocol.
1. Introduction

1.3.4 Graph Editor Improvements for Pattern Definition

The current graph editor allows to create different nodes and edges. Additionally, key-value pairs can be defined. For visualization purposes, elements can be colored, too. By creating patterns by ourselves, we check for improvements in functionality and usability.

Throughout our implementations, we modify the graph editor. These changes are necessary for the creation and translation of patterns. Additional changes are mentioned in Chapter 4.

1.3.5 Evaluation of Feasibility and Correctness

This paper is an explorative attempt to further improve the approach of Christian Wulf regarding parallelization. After examining the goals mentioned before, we will come to a conclusion about feasibility and correctness.

1.4 Document Structure

The next chapter presents required foundations and technologies. Because of multiple goals, we cover a broader spectrum. In Chapter 3, we analyze the graph editor. This is mandatory for implementing the desired changes. Then, Chapter 4 shows our changes to the graph editor. Chapter 5 shows our approach of translation rules. This translation is enriched in Chapter 6 by the User Interaction Protocol. In Chapter 7, we present the architecture of a translation mechanism. Also, the translator is integrated into the graph editor. Then, Chapter 8 and Chapter 9 evaluate our approach. We differentiate between the evaluation of the Candidate Pattern translation and the evaluation of the Parallelization Pattern translation. At last, we present in Chapter 10 related work and in Chapter 11 a conclusion.
Chapter 2

Foundations and Technologies

This chapter gives additional information required for a better understanding of the thesis. We give an overall overview of the topic, before explaining the used part of the technology. At first, we give a short explanation about the used graphs. Then, we introduce Neo4j, Cypher, and our used system dependency graph model. Afterwards, we present the graph editor and the Base Model.

2.1 Graph Theory

Graphs are an abstract structure to represent connections between objects. These connections may be directed or undirected and can be weighted. A graph $G$ is an ordered pair $G = (V, E)$, where $V$ is a set of nodes (vertices) and $E$ is a set of edges. Edges may connect only two or more nodes, depending on the graph style.

2.1.1 Property Graphs

A property graph is a directed graph with attributes and labels. Additionally, key-value pairs can be stored. [Rodriguez and Neubauer 2010]

![Figure 2.1. A simple property graph](image)

Figure 2.1 shows a property graph with three labeled nodes. Their labels describe their name, while their role is stored in a key-value pair. These pairs enhance the graph with additional information. Edges can have labels and attributes, too.

Property graphs are used by Neo4j and the Base Model of the graph editor. They are the foundation of the structure of both technologies.
2. Foundations and Technologies

2.1.2 Dependency Graphs

A dependency graph is a directed graph. It represents the dependencies of certain objects towards each other. Nodes represent the objects, while edges model the dependencies. Regarding data flow, these dependencies may influence the order of the execution. There may be more than one possible execution order that will terminate. Topological sorting can derive execution orders.

![Figure 2.2. A simple dependency graph](image)

Figure 2.2 shows a dependency graph with four labeled nodes. The edges represent the data flow. As we can see, E depends on B, D depends on B and C, C depends on A, B depends on A, and A has no dependencies. Calculating the topological order results in the set \{(A, B, C, D, E), (A, B, C, E, D), (A, B, E, C, D), (A, C, B, D, E), (A, C, B, E, D)\}. Having this outcome, we know that there exist no cyclic dependencies and we have execution orders which will terminate. Because of the switches in order, we derive that the nodes B, C and the nodes D, E can be executed simultaneously.

2.1.3 Program Dependency Graphs

Program dependency graphs (PDG) represent control and data dependencies within procedures. Each node represents a statement. A control dependency exists, if a node refers to another node, which represents a method. This relationship is equivalent to the execution order. Data dependency is the requirement of data, which another node processes and/or provides. [Würthinger et al. 2008; Ballance and Maccabe 1992]

![Figure 2.3. A simple program dependency graph](image)

Figure 2.3. A simple program dependency graph
control dependency: ─ data dependency: ─→
2.1. Graph Theory

Figure 2.3 shows a program dependency graph with eight nodes. Each node represents one statement. The arrows represent dependencies. An orange arrow means a control dependency, while a green arrow means a data dependency. The arrow points to a node, which requires the data or is next in execution. The graph corresponds to the pseudocode shown in Listing 2.1.

Listing 2.1. The pseudocode for Figure 2.3

1. `int x = readLine();`
2. `int val = 5;`
3. `if (x <= val) then {`
4. `printLine("smaller");`
5. } else {
6. `val = x;`
7. `printLine("bigger");`
8. }

In contrast to source code, visualizing the semantic coherence in a graph reveals possibilities to change execution orders better, without having an influence on the functionality. Also, independent calculations are shown. Overall, PDGs visualize the possible utilization of multiple processor cores and improve parallelism on a low level. [Würthinger et al. 2008; Ballance and Maccabe 1992]

2.1.4 System Dependency Graphs

System dependency graphs (SDG) are an enhancement of program dependency graphs. They represent control and data dependencies of an application. Here, a node can also represent a function call or a procedure. There exist additional edges to model inheritance and the passing of parameters. Like PDGs, they reveal the possibilities to change execution orders, without having an influence on the functionality. Allowing independent procedures at the same time improves parallelism on a higher level. [Horwitz et al. 1990; Walkinshaw et al. 2003]
2. Foundations and Technologies

Figure 2.4 shows an SDG with three procedures. Again, the arrows represent dependencies. The newly appearing violet edge represents a procedure call. There are three procedure declarations: MAIN, ADD(a), and SUB(b). The main method calls both ADD(a) and SUB(b). Because of no dependencies between those two procedures, they can be executed simultaneously.

The Java System Dependency Graph
Different programming languages follow different paradigms. Therefore, SDGs may differ in functionality. The Java System Dependency Graph (JSDG) is separated in layers with different components, each having a different purpose. This allows to clarify the structure and functionality of the application. The JSDG is used by the graph editor presented in Section 2.3. [Walkinshaw et al. 2003]

The Statement Graph A statement is an atomic construct representing a single expression in the source code. The only type of node of this graph are Statements. Edges can only represent Data Dependence and Control Dependence. A call functionality requires special representations given in The Method Dependency Graph. [Walkinshaw et al. 2003]

The Method Dependency Graph The method dependency graph (MDG) is the layer on top of statements. It represents a single method or a procedure in a program. The layer provides three different nodes and two different edges. The Method Entry node represents the entry point of a procedure. Respectively, a Call Dependence edge is introduced. Actual-In/Out nodes copy each variable to/from a temporal variable. Formal-In/Out nodes are used by called methods to copy variables to the temporal variable. To make this possible, a Parameter In/Out edge exists. Further details are mentioned in the linked source. [Walkinshaw et al. 2003]

The Class Dependency Graph The class dependency graph (ClDG) represents the classes in a program. It represents the hierarchy and provides the Class Entry node for each class declaration. Edges represent a Class Membership, a Class Dependence or a Data Member. Data Member edges connect the class entry with statement nodes or attributes. [Walkinshaw et al. 2003]

The Interface Dependency Graph The interface dependency graph (InDG) represents the interfaces in a program. It provides the Interface Entry node, which is connected to the abstract methods via Abstract Member edges. These methods use Implements Abstract Method edges to connect to their implementing methods. Classes implementing an interface use the Implements edge. [Walkinshaw et al. 2003]

The Package Dependency Graph The package dependency graph (PaDG) represents the packages in a program. It provides the Package Entry node, which is connected to its classes and interfaces via the Package Member edges. [Walkinshaw et al. 2003]
2.1. Graph Theory

Modifications

To create SDGs from source code, we use Java2Neo4j\(^1\). Since it provides different and additional nodes and edges, the SDG model in the graph editor has to be modified. Table 2.1 shows the elements in the SDG and their corresponding element in our implementation. In the following, the changes are explained.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Type</th>
<th>JSDG</th>
<th>Our Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement</td>
<td>Node</td>
<td>Assignment</td>
<td>Assignment</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Condition</td>
<td>Condition</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Declaration</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Invocation</td>
<td>MethodCall</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Loop</td>
<td>MethodCallWithReturnValue</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Noop</td>
<td>NopStmt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Return</td>
<td>ReturnStmt</td>
</tr>
<tr>
<td>Edge</td>
<td>Data Dependence</td>
<td>DATA_FLOW</td>
<td>DATA_FLOW</td>
</tr>
<tr>
<td></td>
<td>Control Dependence</td>
<td>CONTROL_FLOW</td>
<td>CONTROL_FLOW</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AGGR_CTRL_FLOW</td>
<td>AGGR_CTRL_FLOW</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CALLER_OF</td>
<td>CALLER_OF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LAST_UNIT</td>
<td>LAST_UNIT</td>
</tr>
<tr>
<td>Method</td>
<td>Node</td>
<td>Method Entry</td>
<td>Method</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Actual In</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Actual Out</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Formal In</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Formal Out</td>
<td>-</td>
</tr>
<tr>
<td>Edge</td>
<td>Call Dependence</td>
<td>CALLS</td>
<td>CALLS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Parameter In</td>
<td>AGGREGATED_CALLS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Parameter Out</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-</td>
<td>CONTAINS_UNIT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-</td>
<td>LAST_UNIT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-</td>
<td>THROWS</td>
</tr>
</tbody>
</table>

\(^1\)https://build.se.informatik.uni-kiel.de/chw/sootexample
2. Foundations and Technologies

<table>
<thead>
<tr>
<th>Layer</th>
<th>Type</th>
<th>JSGD</th>
<th>Our Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement Layer</td>
<td>Class</td>
<td>Node Class Entry</td>
<td>Class</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Field</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ConstructorCall</td>
</tr>
<tr>
<td></td>
<td>Edge Class Dependence</td>
<td></td>
<td>CONTAINS_CONSTRUCTOR</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CONTAINS_METHOD</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CONTAINS_FIELD</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>AGGREGATED_FIELD_READ</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>AGGREGATED_FIELD_WRITE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>EXTENDS</td>
</tr>
<tr>
<td></td>
<td>Node Interface Entry</td>
<td></td>
<td>Interface</td>
</tr>
<tr>
<td></td>
<td>Edge Abstract Member</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IMPLEMENTS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Implements Abstract Method</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Node Package Entry</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>Edge Package Member</td>
<td></td>
<td>CONTAINS_TYPE</td>
</tr>
</tbody>
</table>

**Statement Layer**  The node Declaration is removed. The nodes MethodCall and MethodCallWithReturnValue call nodes of the type Method. The node Loop and the node Noop are combined in the node NopStmt. The node NopStmt has the additional property nopkind, which distinguishes the loops. This property has one of the following values: NULL, IF_COND, IF_THEN, IF_ELSE, IF_END, FOR_INIT, FOR_COND, FOR_UPDATE, FOR_BODY, FOR_END, FOREACH_INIT, FOREACH_COND, FOREACH_BODY, FOREACH_END

Additionally, three edges are introduced. AGGR_CTRL_FLOW is used for control flow between for loops. CALLER_OF leads from a node of the type Assignment to a node of the type MethodCallWithReturnValue. LAST_UNIT leads from the last node of a condition to the first node of the condition.

**Method Layer**  The nodes Actual In, Actual Out, Formal In, and Formal Out are not yet supported. The edge CALLS leads from a node of the type Assignment to the implementation of a method. AGGREGATED_CALLS leads from a method to another method. The edges Parameter In and Parameter Out are not yet supported. The edge CONTAINS_UNIT points to local fields inside the method. LAST_UNIT leads from the last node of a method to the first node of the method. The edge THROWS leads to the node Class of the thrown exception.
2.2. The Graph Database Neo4j

Class Layer  We add the node Field to the Class Layer. It represents global variables in a class. The edge Class Dependence is removed. Methods and constructors can be members of a class. Therefore, the edges CONTAINS_CONSTRUCTOR and the CONTAINS_METHOD are introduced. They replace the Class Member edge. The edge CONTAINS_FIELD leads from a node of the type Class to a node of the type Field. AGGREGATED_FIELD_READ and AGGREGATED_FIELD_WRITE represent the usage of a global variable. The edge EXTENDS is similar to the keyword extends in Java.

Interface Layer  The edges Abstract Member and Implements Abstract Method are removed. All other objects are renamed.

Package Layer  No changes except for renames.

2.2 The Graph Database Neo4j

Database systems (DBS) manage huge amounts of data. The data has to be stored in an efficient, permanent, and consistent way. The system is made of a database and the management system itself.

A database management system (DBMS) defines the used database model. It is a software organizing the internal storage and the structure of the data. Additionally, it provides different tools to manage the used databases. Interfaces give applications and users the possibility to access, evaluate, change and manage selected data sets. These data sets can be accessed by a query language supported by the DBS. The management system governs the read/write access of different users to the database and optimizes queries.

Neo4j is a NoSQL graph DBMS implemented in Java and was initially released in 2007. In contrast to relational databases, graph databases use graph structures instead of tables. In contrast to relational databases, they can represent rules and functions connecting the data. Additionally, semantic relationships between entities can be expressed. Complex contexts are faster to access because there is no need to join different tables. [Neo Technology 2007; Mutschke 1995]

2.2.1 Neo4j’s Graph Model

Figure 2.5 shows the components of a property graph. Neo4j uses this model to store the data. Each piece of information is either a node, an edge (relationship), or an attribute (property) of one of them. This allows complex relationships. Additionally, graph databases map more directly to the structure of object-oriented applications. This allows to query for patterns within the program, instead of checking source code. [Neo Technology 2007; 2015]

For looking up huge amount of values, a key-value implementation is highly performant. If the values themself are connected with relationships, a graph store is set up. Figure 2.6 shows the difference between a normal key-value store and a graph key-value store. While containing the same data, the graph stores less redundant information. [Neo Technology 2007]
2. Foundations and Technologies

2.2.2 The Query Language Cypher

Software written in other languages than Java is supported through an HTTP endpoint. This endpoint grants communication with the database via the language **Cypher**. **Cypher** is a declarative graph query language. It is mostly inspired by **SQL** and **SPARQL** for its structure and pattern matching. It allows efficient querying and updating of the graph store. Regular expressions are supported, too. [Neo Technology 2007]
The queries follow a certain syntax, starting with a keyword. For our purpose, these are the most important ones:

- **CREATE** - Creates nodes or edges
- **DELETE** - Deletes nodes or edges
- **DETACH DELETE** - Deletes a node and all connected edges
- **SET** - Sets or updates a key-value pair
- **REMOVE** - Removes a key-value pair
- **MATCH** - Finds all data having the form of the described graph
- **WHERE** - Defines the properties of the graph and result
- **WITH** - Chains query parts together by passing elements
- **RETURN** - Defines return values

Nodes are enclosed in round brackets. Properties of nodes are enclosed in curly brackets within the node. Edges/relationships are written like arrows, containing a description (\-[:Example]-\>).

**Listing 2.2. A Neo4j Cypher query to create a graph**

```
1 CREATE (node1:Method {name: 'main'}),
2 (node2:Method {name: 'translate()'})
3 (node3:Method {name: 'getString()'}),
4 (node4:Method {name: 'createOutput'()}),
5 (node5:Method {name: 'gatherData('})
6 (node1)-[:calls]-> (node2),
7 (node1)-[:calls]-> (node4),
8 (node2)-[:calls]-> (node3),
9 (node4)-[:calls]-> (node5)
```

Listing 2.2 shows a Cypher query to create a new graph. The first five lines create nodes of the type Method with the property name. Note that there is a difference between the name of a node and the property name. The following four lines define the relationship between the nodes and give them a name. This query results in the graph shown in Figure 2.7.

**Listing 2.3. A Neo4j Cypher query to find structures**

```
1 MATCH (node)-[edge:calls]->(function)
2 WITH node, COUNT(DISTINCT function) AS method_calls
3 WHERE node.name=~ 'm.*' AND method_calls > 1
4 RETURN node.name
```

13
2. Foundations and Technologies

Listing 2.3 shows a Cypher query to find properties in structures like Figure 2.7. The first line (MATCH) filters all subgraphs where a child exists. This child has to be connected via a calls edge. Note that the elements are represented by a variable. For example, node and function are variables for two nodes. In the second line (WITH) all children of a parent are counted. The value is saved in the variable method_calls. This line of code acts like a border in the query and defines the variables to be passed. The third line (WHERE) contains a regular expression. The name of the parent has to start with the small letter 'm'. Additionally, its number of children has to be bigger than 1. The last line (RETURN) is responsible for returning the name of the node matching these conditions. When applying this query to our created graph, the output is “main”.

2.3 The Graph Editor

As a part of his Bachelors thesis, Lars Erik Blümke created an Eclipse plugin to visualize and edit graphs. The focus was set on creating and adapting system dependency graphs. It is based on KLighD (KIELER Lightweight Diagrams), which is a subcomponent of the KIELER\(^2\) project. [Blümke 2015]

The graph editor can be operated with a mouse, has an own context menu and comes with predefined elements for system dependency graphs. These elements correspond to the Java System Dependency Graph presented in Section 2.1.4. Additionally, it supports key-value tables for nodes and edges. Users can define highlighting colors for selected key-value pairs. Graphs are stored in a .propertygraph file. [Blümke 2015]

\(^2\)http://www.rtsys.informatik.uni-kiel.de/en/research/kieler/
2.3. The Graph Editor

Figure 2.8. The GUI of the graph editor

Figure 2.8 shows the graph editor as an Eclipse application. On the left side is the standard Java package explorer ①. The KlighD Diagram View displays the current graph ②. On the right, a tree editor shows all components and their children ③. At the bottom, the properties of the currently selected element can be seen ④. A click with the right mouse button on the KlighD Diagram View opens a context menu ⑤. There, several options for editing the graph are provided. On the top, two new buttons are added to the toolbar. A colored circle allows to set key-value-color triplets ⑥. Pressing the button results in a new window to set the properties ⑦. The other new button enables a recording mechanism ⑧. Changes to the graph can be recorded and saved into a file.

2.3.1 The User Interaction Protocol

The User Interaction Protocol (UIP) is a part of the graph editor and is stored in a .history file. It records the changes made to a graph when the record button is pressed. It can be opened with a regular text editor. There are 10 different operations, which can be recorded. [Benekov 2015]

**ADD_NODE**

UIP history:  `ADD_NODE, id`

Semantics: A new node with the ID `id` has been created.
2. Foundations and Technologies

**ADD_NODE_PROPERTY**

UIP history: ADD_NODE_PROPERTY, id, key, value
Semantics: The node with the ID id got the new property key with the value value.

**UPDATE_NODE_PROPERTY**

UIP history: UPDATE_NODE_PROPERTY, id, key, new.value
Semantics: The node with the ID id has had its property key set to new.value.

**REMOVE_NODE_PROPERTY**

UIP history: REMOVE_NODE_PROPERTY, id, key
Semantics: The node with the ID id has had its property key removed.

**REMOVE_NODE**

UIP history: REMOVE_NODE, id
Semantics: The node with the ID id has been removed.

**ADD_EDGE**

UIP history: ADD_EDGE, id, source.ID, target.ID
Semantics: A new edge with the ID id has been created. It starts at the node with the ID source.ID and ends at the node with the ID target.ID.

**ADD_EDGE_PROPERTY**

UIP history: ADD_EDGE_PROPERTY, id, key, value
Semantics: The edge with the ID id got the new property key with the value value.

**UPDATE_EDGE_PROPERTY**

UIP history: UPDATE_EDGE_PROPERTY, id, key, new.value
Semantics: The edge with the ID id has had its property key set to new.value.

**REMOVE_EDGE_PROPERTY**

UIP history: REMOVE_EDGE_PROPERTY, id, key
Semantics: The edge with the ID id has had its property key removed.

**REMOVE_EDGE**

UIP history: REMOVE_EDGE, id
Semantics: The edge with the ID id has been removed.
2.3. The Graph Editor

2.3.2 The Base Model

The graph editor differentiates between two models: the View Model and the Base Model. The View Model offers visual elements representing parts of the code. The Base Model is a modified property graph and represents the source code. It supports one type of nodes and edges, which are classified by their key-value pairs. A mechanism called synthesis transforms the Base Model into the View Model. [Blümke 2015]

![Diagram of the Base Model Structure](image)

**Figure 2.9.** The architecture of the used base model [Blümke 2015]

Figure 2.9 shows the architecture of the Base Model. The property graph consists of an amount of nodes and edges. Nodes and edges are the only elements in the graph. Graph elements have a label, an ID and properties for visualization. Additionally, they can be extended by key-value pairs. Key-value pairs are stored in HashMaps and may have a global coloring set by the property graph. An edge always has a source node and a target node. Deleting one of these nodes will result in the edge being deleted, too. [Blümke 2015]

Listing 2.4 shows the XML serialization of the graph the shown in Figure 2.8. Opening the graph file in a text editor reveals XML code. Each node is listed with its properties. This structure allows a translation to a Cypher query by traversing the structure.
2. Foundations and Technologies

<table>
<thead>
<tr>
<th>Listing 2.4. The XML serialization of the graph shown in Figure 2.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  &lt;?xml version=&quot;1.0&quot; encoding=&quot;UTF-8&quot;?&gt;</td>
</tr>
<tr>
<td>2  &lt;propertygraph xmlns:xmi=&quot;http://www.omg.org/XMI&quot; nodeCounter=&quot;1&quot; edgeCounter=&quot;1&quot;&gt;</td>
</tr>
<tr>
<td>3  xmlns:propertygraph=&quot;<a href="http://de.cau.se.grapheditor.propertygraph/propertygraph">http://de.cau.se.grapheditor.propertygraph/propertygraph</a>&quot; nodeCounter=&quot;1&quot; edgeCounter=&quot;1&quot;&gt;</td>
</tr>
<tr>
<td>4  &lt;nodes label=&quot;CONDITION&quot; color=&quot;FFFF00&quot; ID=&quot;node_2&quot; incomingEdges=&quot;edge_2&quot;&gt;</td>
</tr>
<tr>
<td>5  &lt;keyValuePairs key=&quot;type&quot; value=&quot;condition&quot;/&gt;</td>
</tr>
<tr>
<td>6  &lt;outgoingEdges color=&quot;000000&quot; ID=&quot;edge_3&quot; targetNode=&quot;node_5&quot;&gt;</td>
</tr>
<tr>
<td>7  &lt;keyValuePairs key=&quot;case&quot; value=&quot;false&quot;/&gt;</td>
</tr>
<tr>
<td>8  &lt;/outgoingEdges&gt;</td>
</tr>
<tr>
<td>9  &lt;outgoingEdges color=&quot;000000&quot; ID=&quot;edge_4&quot; targetNode=&quot;node_6&quot;&gt;</td>
</tr>
<tr>
<td>10  &lt;keyValuePairs key=&quot;case&quot; value=&quot;true&quot;/&gt;</td>
</tr>
<tr>
<td>11  &lt;/outgoingEdges&gt;</td>
</tr>
<tr>
<td>12  &lt;/nodes&gt;</td>
</tr>
<tr>
<td>13  &lt;nodes label=&quot;ASSIGNMENT&quot; color=&quot;FFFFFF&quot; ID=&quot;node_3&quot; incomingEdges=&quot;edge_1&quot;&gt;</td>
</tr>
<tr>
<td>14  &lt;keyValuePairs key=&quot;type&quot; value=&quot;assignment&quot;/&gt;</td>
</tr>
<tr>
<td>15  &lt;outgoingEdges label=&quot;DATA FLOW&quot; color=&quot;000000&quot; ID=&quot;edge_2&quot; targetNode=&quot;node_2&quot;&gt;</td>
</tr>
<tr>
<td>16  &lt;keyValuePairs key=&quot;case&quot; value=&quot;control dependence&quot;/&gt;</td>
</tr>
<tr>
<td>17  &lt;/outgoingEdges&gt;</td>
</tr>
<tr>
<td>18  &lt;/nodes&gt;</td>
</tr>
<tr>
<td>19  &lt;nodes label=&quot;DECLARATION&quot; color=&quot;FFFFFF&quot; ID=&quot;node_4&quot;&gt;</td>
</tr>
<tr>
<td>20  &lt;keyValuePairs key=&quot;type&quot; value=&quot;declaration&quot;/&gt;</td>
</tr>
<tr>
<td>21  &lt;outgoingEdges label=&quot;color&quot; color=&quot;0077FF&quot; ID=&quot;edge_1&quot; targetNode=&quot;node_3&quot;&gt;</td>
</tr>
<tr>
<td>22  &lt;keyValuePairs key=&quot;color&quot; value=&quot;blue&quot;/&gt;</td>
</tr>
<tr>
<td>23  &lt;/outgoingEdges&gt;</td>
</tr>
<tr>
<td>24  &lt;/nodes&gt;</td>
</tr>
<tr>
<td>25  &lt;nodes label=&quot;RETURN&quot; color=&quot;FF0000&quot; ID=&quot;node_5&quot; incomingEdges=&quot;edge_3&quot;&gt;</td>
</tr>
<tr>
<td>26  &lt;keyValuePairs key=&quot;type&quot; value=&quot;return&quot;/&gt;</td>
</tr>
<tr>
<td>27  &lt;/nodes&gt;</td>
</tr>
<tr>
<td>28  &lt;nodes label=&quot;LOOP&quot; color=&quot;00FF00&quot; ID=&quot;node_6&quot; incomingEdges=&quot;edge_4&quot;&gt;</td>
</tr>
<tr>
<td>29  &lt;keyValuePairs key=&quot;type&quot; value=&quot;loop&quot;/&gt;</td>
</tr>
<tr>
<td>30  &lt;/nodes&gt;</td>
</tr>
<tr>
<td>31  &lt;keyValueColorTriplets key=&quot;type&quot; value=&quot;condition&quot; color=&quot;FFFF00&quot;/&gt;</td>
</tr>
<tr>
<td>32  &lt;keyValueColorTriplets key=&quot;type&quot; value=&quot;loop&quot; color=&quot;00FF00&quot;/&gt;</td>
</tr>
<tr>
<td>33  &lt;keyValueColorTriplets key=&quot;type&quot; value=&quot;return&quot; color=&quot;FF0000&quot;/&gt;</td>
</tr>
<tr>
<td>34  &lt;keyValueColorTriplets key=&quot;color&quot; value=&quot;blue&quot; color=&quot;0077FF&quot;/&gt;</td>
</tr>
<tr>
<td>35  &lt;/propertygraph:Propertygraph&gt;</td>
</tr>
</tbody>
</table>
Reverse Engineering of the Graph Editor

To implement our desired changes, we analyze the source code of the graph editor. We have to differentiate between parts of the KLighD plugin and the graph editor itself. This allows to evaluate the feasibility of our desired improvements and the effort to implement them.

Figure 3.1. Simplified package diagram of the graph editor
3. Reverse Engineering of the Graph Editor

Figure 3.1 shows a simplified package diagram of the graph editor. It renounces to show further subpackages to maintain clarity. In the following, we describe the content and functionality of each package shown. If a package further contains important files or classes for or desired changes, they are presented and analyzed.

**de.cau.se.grapheditor**

The package **grapheditor** contains all other packages. Additionally, it is responsible for assembling the layout of the workbench. The class **ElementTypes** provides enums of the implemented nodes and edges. The file **plugin.xml** sets up the context menu for the graph editor (see figure 2.8).

**de.cau.se.grapheditor.actions**

The package **actions** is responsible for mouse interaction. It provides the implementation of the actions of the context menu (see figure 2.8). Subpackages of the action package extend these actions according to the selected element. Additionally, the class **PropertygraphActionDefinitions** defines the graph events for each action done in the graph editor.

**de.cau.se.grapheditor.wizards**

The package **wizards** is responsible for creating, importing, and exporting graphs.

**de.cau.se.grapheditor.propertygraph**

The package **propertygraph** contains the EMF ecore model, which we call **Base Model**. The package is also responsible for the XML serialization. Additionally, it contains the subpackages **edit** and **editor**, which set up the layout for the tree editor (see figure 2.8).

**de.cau.se.grapheditor.propertygraphKLighD**

The package **propertygraphKLighD** is a created KLighD project. It contains the class **PropertygraphDiagramSynthesis**, which is responsible for the synthesis of the used models. Therefore, it requires access to the **propertygraph** package.

**de.cau.se.grapheditor.workbench**

The package **workbench** contains the setup for the view of properties and key-value pairs (see figure 2.8). If graph events fire, the class **KeyValueView** updates the view of key-value pairs.

**de.cau.se.grapheditor.colorchoicedialog**

The package **colorchoicedialog** adds the color choice dialog for key-value-color triplets to the graph editor.
The package recorder contains the files and classes required for creating the UIP. The class GraphRecorder contains a switch statement for all graph events to create the UIP. The subpackage listeners contains the classes GraphChangeType and GraphChangeEvent. They provide enums and handlers for each recorded graph event.
Improvement of the Graph Editor

Some Candidate Patterns require additional constructs to be visualized. For example, two nodes may not be allowed to have a connecting edge. This has to be visualized and translated. Additionally, our first impressions of the graph editor gave us some ideas for improvements. In this chapter, we present these modifications and their implementation.

4.1 Implementation of our System Dependency Graph Model

In Section 2.1.4, we presented the types of nodes and edges we use in our SDG model. These types replace the existing ones and are added to the context menu.

4.2 Renaming of User Interaction Protocol Operations

Implementing an optimization for the UIP requires to analyze each entry in the .history file. For an easier analyzing of strings, they are required to have the same length and the same order of their operations. Therefore, we rename the operations before being recorded. They follow the pattern <ELEMENT>_<TYPE>_<ACTION>. <ELEMENT> refers to a node or an edge. <TYPE> defines, if either the element itself or a property of it is manipulated. <ACTION> is the operation done in the graph editor. This allows to distinguish early between different operations. Table 4.1 shows the old operations and their corresponding new name. Deleting a key-value pair can delete a property or a type. Because their Cypher translation differs, the <ELEMENT>_PROP_DEL operation now stores the value, too.

4.3 Label Updates for Type Changes

Changing the type of an edge or a node is a common task in the process of pattern creation. Therefore, all information concerning that element needs to be refreshed in the view. When adding a new node, its label is set to their type at creation. Edges do not have any labels at all. We implemented a mechanism, which automatically sets the label of edges and nodes according to their current type.
4. Improvement of the Graph Editor

Table 4.1. Operations of the UIP

<table>
<thead>
<tr>
<th>Old Operation</th>
<th>New Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD_NODE, id</td>
<td>NODE_ELEM_ADD, id</td>
</tr>
<tr>
<td>REMOVE_NODE, id</td>
<td>NODE_ELEM_DEL, id</td>
</tr>
<tr>
<td>ADD_EDGE, id, source, target</td>
<td>EDGE_ELEM_ADD, id, source, target</td>
</tr>
<tr>
<td>REMOVE_EDGE, id</td>
<td>EDGE_ELEM_DEL, id</td>
</tr>
<tr>
<td>ADD_NODEPROPERTY, id, key, value</td>
<td>NODE_PROP_ADD, id, key, value</td>
</tr>
<tr>
<td>UPDATE_NODEPROPERTY, id, key, new_value</td>
<td>NODE_PROP_SET, id, key, new_value</td>
</tr>
<tr>
<td>REMOVE_NODEPROPERTY, id, key</td>
<td>NODE_PROP_DEL, id, key, value</td>
</tr>
<tr>
<td>ADD_EDGEPROPERTY, id, key, value</td>
<td>EDGE_PROP_ADD, id, key, value</td>
</tr>
<tr>
<td>UPDATE_EDGEPROPERTY, id, key, new_value</td>
<td>EDGE_PROP_SET, id, key, new_value</td>
</tr>
<tr>
<td>REMOVE_EDGEPROPERTY, id, key</td>
<td>EDGE_PROP_DEL, id, key, value</td>
</tr>
</tbody>
</table>

4.4 Implementation of a Multi Edge

In some cases, a path between two nodes has to be modeled. This path can include more than one edge and its length can be unknown. The nodes on this path are negligible. Therefore, we implement the property \texttt{MULTIEDGE}.

\texttt{A MULTIEDGE} is a path of \(n\) edges between two nodes, where \(n \in \mathbb{N}^+\). It can have a type, which applies to all edges on the path. It is implemented by an additional key-value pair with the key \texttt{multiedge} and the value \(m-n\), where \(m < n\) and \(m, n \in \mathbb{N}^+\). \(m\) defines the minimum amount of edges on the path, \(n\) the maximum amount. To describe a path of uncertain length, we allow the \(*\) operator. For example, the key-value pair \texttt{multiedge 3-\ *} describes an amount of at least three edges on the path. On creation, the user is asked for these borders of the path. The label of the edge is set to \texttt{MULTIEDGE: m..n}, ignoring the type.

4.5 Concept and Implementation of Forbidden Elements

Forbidding edges and nodes in a certain constellation allows to design additional patterns. In some cases, a specific type of edge should not connect two nodes. Or, a specific type of node must not exist in a part of the graph. In this chapter, we develop a concept for the implementation of forbidden elements. Additionally, we describe their implementation.

4.5.1 Different Kinds of Restrictions

Before designing a concept for forbidden elements, we have to consider which restrictions can occur in a \textit{Candidate Pattern} or a \textit{Parallelization Pattern}. These restrictions are shown in Figure 4.1 and explained in the following.
4.5. Concept and Implementation of Forbidden Elements

Figure 4.1. Different cases of forbidden elements

The first case shows forbidden properties for nodes and edges, which are presented by key-value pairs. The second case shows forbidden types for nodes and edges. Types are also presented by a key-value pair. The third case shows a forbidden edge, meaning that two nodes are not allowed to have a direct connection. The last case shows a forbidden node. Edges always have a source and a target. Therefore, this case is redundant with the first case or the third case. Either the type of the node is restricted, or the nodes must not have a connecting edge. Other cases are combinations of the ones presented above.

4.5.2 Implementation

Forbidden nodes and edges can be designed as new elements in the editor. This results in a new element for every existing type of nodes and edges. Therefore, new elements have to be considered in the translation mechanism. So when adding a new element, it has to be implemented twice. Additionally, each property requires a forbidden equivalent, represented by a key-value pair. Because there is an infinite number of possible key-value pairs, this implementation is impossible.

To avoid these circumstances, a boolean \texttt{allowed} can be added to all key-value pairs. The default value of this boolean is set to \texttt{true}. This contradicts the Base Model. HashMaps consist of a key and a value only. Therefore, a new data type containing a string and a boolean is necessary to replace the value. This new key-value pair is incompatible with the imported KLighD libraries.

Another solution is to add the prefix \texttt{forbidden} to forbidden types or other key-value pairs. Obviously, this excludes forbidden edges. Because there is an infinite number of possible types, their coloring can not be done with a global key-value-color triplet. So the coloring of nodes or edges with forbidden types has to be done at the element itself.

Our solution is a combination of the approaches above. Forbidden elements are implemented as shown in Table 4.2. While the value for properties depends on the property itself, the value for types and edges is fixed. Forbidden properties are expanded by the prefix \texttt{forbidden}. They are not colored for the reasons mentioned before. A type of a node or an edge is declared forbidden, if the key-value pair \texttt{forbiddden\_type true} is added to that element. This requires the element to already have a type. An edge is declared forbidden, if the key-value pair \texttt{forbidden\_edge FORBIDDEN} is added.
4. Improvement of the Graph Editor

Table 4.2. Key-value pairs for forbidden elements

<table>
<thead>
<tr>
<th>Forbidden Element</th>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property</td>
<td>forbidden&lt;property key&gt;</td>
<td>&lt;property value&gt;</td>
</tr>
<tr>
<td>Type</td>
<td>forbiddentype</td>
<td>FORBIDDEN</td>
</tr>
<tr>
<td>Edge</td>
<td>forbiddenedge</td>
<td>FORBIDDEN</td>
</tr>
</tbody>
</table>

To highlight the visualization of forbidden elements, we color them red. So when declaring a type or an edge as forbidden, its color property must change. This could be done by manipulating the color property itself. If the user wants to change the predefined color of forbidden elements, a manual change of every single forbidden element would be required. Therefore, a key-value-color triplet is the better choice. It globally colors all forbidden elements. It has to be changed at one location to change the color of all forbidden elements. It is limited to forbidden types and edges, because there exists an unlimited amount of forbidden properties, which would have to be defined as a key-value-color triplet. The two key-value-color triplets are created every time a new graph is created for the keys forbiddenedge and forbiddentype and the value true.

4.6 Workaround for not Firing Graph Events

Operating in the graph editor with the keyboard can cause graph events to not be registered by the implemented logic. For example, deleting a key-value pair in the tree view with the delete key removes the element, but does not fire a graph event. These graph events are required by the UIP to record the changes by the user.

To fix this circumstance and create an alternative for the tree editor (see 3 at Figure 2.8), we implemented a menu to modify and delete key value pairs. Figure 4.2 shows this menu. It shows all key value pairs of the currently selected element in a table, except for types. Additionally, there is an extra column to show if a key-value pair is forbidden. This is done by checking if a key-value pair contains the prefix forbidden. The button Modify Value allows to set a new value for the selected key-value pair. The button Delete KVP deletes the selected key-value pair.
4.7 Changes to the Context Menu

The context menu allows to create and modify elements. While a lot of options are predefined by the KlighD plugin, some are customizable and created for pattern definitions.

![Figure 4.3. Changes to the context menu](image)

Figure 4.3 shows the previous context menu on the left and the changes done to the right. The operations listed below are predefined by the KlighD plugin and can not be changed, grouped, or deleted:

- Show Layout View
- Add Template
- Clip selected
- Clip towards selection
- Clip parent
- Clip root
- Scale up
- Scale down
- Scale 100%
- Save as image...
- Export KGraphs
- Connect Nodes
- Add New Node
- Add Successor
- Remove Element
- Set Predefined Type
- Save history

- Show Layout View
- Add Template
- Clip selected
- Clip towards selection
- Clip parent
- Clip root
- Scale up
- Scale down
- Scale 100%
- Save as image...
- Export KGraphs

The order of the remaining operations is changed to match the workflow. At first, Add New Node allows to create new nodes. Then, Connect Nodes allows to connect two nodes. We added the possibility to set the type of the edge or to create a forbidden edge. Then,
4. Improvement of the Graph Editor

Set Type allows to change the type of already existing nodes or edges. This operation was previously called Set Predefined Type. Now it is also possible to make the current type forbidden or set it to UNDEFINED. The menu Add Successor has been removed completely. We add the menu Key-Value Pairs, which allows to set new (forbidden) key-value pairs. Also the table shown in Figure 4.2 can be opened in this menu. At last, the currently selected element can be deleted with Remove Element.

4.8 Manual for the Creation of Patterns

To summarize our changes and explain the process of pattern creation and translation, we create a manual. In this manual, we cover the first configuration of the editor up to translating patterns. The manual is shown in Appendix B.
To develop a translation for Candidate Patterns, the XML serialization of the base model has to be analyzed. For single elements and different constructs, a corresponding Cypher query has to be created.

Figure 5.1. Translating simple structures

Figure 5.1 shows the procedure of the manual graph translation. At first, all different types of nodes and edges are translated into a Cypher query. Finally, we have a look at advanced constructs.

5.1 Translation of Single Elements

The base model of the graph editor supports two types of objects: nodes and edges. At first, a Cypher query is created for all types of nodes. Afterwards, the same is done for all types of edges.

5.1.1 Nodes

Our used 506 provides 12 different node types to express different program elements. Figure 5.2 shows how an unspecified node is visualized in the graph editor. We will use it as an example for the manual graph translation.

Listing 5.1 shows the code representation of an unspecified node. The lines 1-5 set up the XML file and property graph settings. Line 5 shows the counter for nodes. It is used for having unique IDs for each node. Line 6 defines the unspecified node. It gives information about the label, the color and the ID. The last line closes the property graph definition.
5. Translation of the Candidate Pattern Graph

Listing 5.1. The XML serialization representing an unspecified node

```xml
1  <?xml version="1.0" encoding="UTF-8"?>
3     xmlns:propertygraph="http://de.cau.se.grapheditor.propertygraph/propertygraph"
4     nodeCounter="1">
5     <nodes label="UNSPECIFIED" color="FFFFFF" ID="node_1"/>
6  </propertygraph:xmi:Propertygraph>
```

Listing 5.2 shows the corresponding Cypher query for the unspecified node. Since the lines 1-4 of the XML serialization are only used for setting up the graph, they can be ignored. Line 5 translates into MATCH (node_1), because the type of the node is not specified. The properties color and ID are used in the graph editor for visualization only. Line 6 can be left out, too. At the end of the Cypher, the return statement is responsible for returning a possible assignment for node_1.

Listing 5.2. The Cypher query for Listing 5.1

```cypher
1 MATCH (node_1)
2 RETURN node_1
```

Nodes other than the unspecified node follow a slightly different syntax. Their XML serialization has an additional line containing a key-value pair in the declaration of the node. Listing 5.3 highlights these differences in the XML serialization. While the text in light blue depends on the type of the node, the dark blue text represents the additional key-value pair.

Listing 5.3. The XML serialization representing a node with a type

```xml
6  <nodes label="label" color="FFFFFF" ID="node_ID">
7  
8  </nodes>
```

Listing 5.4 shows the translation.

The corresponding Cypher query uses an additional property for typed nodes. This is added in a WHERE statement. The label is only used for the visualization in the graph editor. Listing 5.4 shows the translation.
5.1. Translation of Single Elements

Listing 5.4. The Cypher query for Listing 5.3

1. MATCH (node_ID)
2. WHERE node_id.type="value"
3. RETURN node_ID

While value and label are different for each type of node, the overall syntax remains the same. Thus, they are placeholders for node types and user-defined labels. Table 5.1 shows all nodes of our SDG presented in Section 2.1.4.

Table 5.1. Different nodes and their values in the graph editor

<table>
<thead>
<tr>
<th>Layer</th>
<th>Node Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement</td>
<td>Assignment Node</td>
<td>Assignment</td>
</tr>
<tr>
<td></td>
<td>Condition Node</td>
<td>Condition</td>
</tr>
<tr>
<td></td>
<td>Method Call With Return Value Node</td>
<td>MethodCallWithReturnValue</td>
</tr>
<tr>
<td></td>
<td>NopStmt Node</td>
<td>NopStmt</td>
</tr>
<tr>
<td></td>
<td>ReturnStmt Node</td>
<td>ReturnStmt</td>
</tr>
<tr>
<td>Method</td>
<td>Method Node</td>
<td>Method</td>
</tr>
<tr>
<td>Class</td>
<td>Class Node</td>
<td>Class</td>
</tr>
<tr>
<td></td>
<td>Field Node</td>
<td>Field</td>
</tr>
<tr>
<td>Interface</td>
<td>Interface Node</td>
<td>Interface</td>
</tr>
<tr>
<td>Package</td>
<td>Package Node</td>
<td>Package</td>
</tr>
</tbody>
</table>

5.1.2 Edges

Our SDG provides 12 different edges to express a transition between nodes. Figure 5.3 shows how an unspecified edge connecting two unspecified nodes is visualized in the graph editor.

![Unspecified Edge](image)

Figure 5.3. An unspecified edge in the graph editor

Listing 5.5 shows the code representation of Figure 5.3. The lines 1-5 set up the XML file and property graph settings. In Line 5, the counter for nodes and edges is given. It is used for having unique IDs for each element. Lines 6-9 define the first unspecified node. In Line 7 and Line 8 the unspecified edge is defined as a property of its origin node. The unspecified edge has no additional label. Line 10 and Line 11 define the second unspecified node. It has the additional property of an incoming edge. The last line closes the definition of the property graph.

![Code Representation](image)
5. Translation of the Candidate Pattern Graph

Listing 5.5. The XML serialization representing an unspecified edge connecting two unspecified nodes

```xml
<?xml version="1.0" encoding="UTF-8"?>
<propertygraph:xmi version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:propertygraph="http://de.cau.se.grapheditor.propertygraph/propertygraph"
nodeCounter="2" edgeCounter="1">
  <nodes label="UNSPECIFIED" color="FFFFFF" ID="node_1">
    <outgoingEdges color="000000" ID="edge_1" targetNode="node_2"/>
  </nodes>
  <nodes label="UNSPECIFIED" color="FFFFFF" ID="node_2" incomingEdges="edge_1"/>
</propertygraph:xmi>
```

Listing 5.6 shows the corresponding Cypher query for the unspecified edge. The first five lines set up the property graph. The sixth line translates into MATCH (node_1). The next line shows the edge as a child of the previous node. Therefore, the default edge translates to -[edge_1]->. The properties `targetNode` in Line 8 and `incomingEdges` in Line 11 define the following node. So Line 10 translates into (node_2), being next to the edge. Now the `RETURN` statement passes node_1, edge_1 and node_2.

```cypher
MATCH (node_1)-[edge_1]->(node_2)
RETURN node_1, edge_1, node_2
```

Edges other than the unspecified edge follow a slightly different syntax. Their XML serialization has an additional line containing a key-value pair in the declaration of the edge. Listing 5.7 highlights these differences in the XML serialization. While the text in light blue depends on the type of the node, the dark blue text is the additional key-value pair.

Listing 5.7. The XML serialization representing a typed edge connecting two unspecified nodes

```xml
<outgoingEdges label="label" color="000000" ID="edge_ID"
targetNode="node_ID">
  <keyValuePairs key="type" value="value"/>
</outgoingEdges>
```

In contrast to typed nodes, the corresponding Cypher query uses a label for typed edges. This is added in the `MATCH` statement. Provided that edge_ID is an outgoing edge of node_ID2, Listing 5.8 shows the translation for Listing 5.7.

```cypher
MATCH (node_1)-[edge_ID:label]->(node_2)
RETURN node_1, edge_ID, node_2
```
5.1. Translation of Single Elements

<table>
<thead>
<tr>
<th>Listing 5.8. The Cypher query for Listing 5.7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MATCH (node_ID2)-[edge_ID:value]-&gt;(node_ID)</td>
</tr>
<tr>
<td>2 RETURN node_ID2, edge_ID, node_ID</td>
</tr>
</tbody>
</table>

In Listing 5.7, `value` and `label` are placeholders for edge types and user defined labels. Table 5.2 shows all edge types of our SDG and their values in the graph editor.

<table>
<thead>
<tr>
<th>Table 5.2. Different edges and their values in the graph editor</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Layer</strong></td>
</tr>
<tr>
<td>Statement</td>
</tr>
<tr>
<td>Statement</td>
</tr>
<tr>
<td>Statement</td>
</tr>
<tr>
<td>Statement</td>
</tr>
<tr>
<td>Statement</td>
</tr>
<tr>
<td>Method</td>
</tr>
<tr>
<td>Method</td>
</tr>
<tr>
<td>Method</td>
</tr>
<tr>
<td>Method</td>
</tr>
<tr>
<td>Method</td>
</tr>
<tr>
<td>Class</td>
</tr>
<tr>
<td>Class</td>
</tr>
<tr>
<td>Class</td>
</tr>
<tr>
<td>Class</td>
</tr>
<tr>
<td>Class</td>
</tr>
<tr>
<td>Class</td>
</tr>
<tr>
<td>Interface</td>
</tr>
<tr>
<td>Package</td>
</tr>
</tbody>
</table>

5.1.3 Key-Value Pairs

So far, key-value pairs stored the type of a node or an edge. Additional information about these elements is also stored in key-value pairs. Listing 5.9 shows the XML serialization of an unspecified node with an additional key-value pair. The lines 1-5 set up the XML file and property graph settings. In Line 5, the counter for nodes is given. Lines 6-8 define the unspecified node. In Line 7 the key-value pair of the node is defined. The last line closes the definition of the property graph.
5. Translation of the Candidate Pattern Graph

**Listing 5.9.** The XML serialization representing an unspecified node with a key-value pair

```xml
<?xml version="1.0" encoding="UTF-8"?>
<propertygraph xmlns:xmi="http://www.omg.org/XMI"
xmlns:propertygraph="http://de.cau.se.grapheditor.propertygraph/propertygraph"
nodeCounter="1">
  <nodes label="UNSPECIFIED" color="FFFFFF" ID="node_1">
    <keyValuePairs key="key" value="value"/>
  </nodes>
</propertygraph>
```

Listing 5.10 shows the corresponding Cypher query for the unspecified node with a key-value pair. The key-value pair is translated into a WHERE statement, which defines the additional properties. In the same way, key-value pairs for edges are translated.

**Listing 5.10.** The Cypher query for Listing 5.5

```cypher
MATCH (node_1)
WHERE node_1.key="value"
RETURN node_1
```

5.2 Translation of Advanced Constructs

Having translated all types of nodes and edges, some constructs are still missing. Special key-value pairs may require a different handling. Additionally, there are some other cases of edges, which we have to consider.

5.2.1 Key-Value Pairs

In Section 5.1.3, we worked with key-value pairs as a definition for the type or a property of an element. But elements may have more than one property. Also, the graph editor supports a key-value-color triplet.

**Multiple Key-Value Pairs**

Each node (except the unspecified) has a type defined by a key-value pair. If a node has additional properties, it requires additional key-value pairs.
5.2. Translation of Advanced Constructs

Listing 5.11. The XML serialization of an unspecified node with two key-value pairs

```xml
<?xml version="1.0" encoding="UTF-8"?>
<propertygraph:xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:propertygraph="http://de.cau.se.grapheditor.propertygraph/propertygraph"
nodeCounter="1">
  <nodes label="UNSPECIFIED" color="FFFFFF" ID="node_1">
    <keyValuePairs key="key" value="value"/>
    <keyValuePairs key="key2" value2="value"/>
  </nodes>
</propertygraph:xmi>
```

Listing 5.11 shows the XML serialization of a typed node with an additional key-value pair. The lines 1-5 set up the XML file and the graph properties. Lines 6-9 define the unspecified node. In Line 7 and Line 8, two key-value pairs are defined. They represent a property inside the definition of the node. Line 10 closes the definition of the property graph.

Listing 5.12. The Cypher query for Listing 5.11

```cypher
MATCH (node_1)
WHERE node_1.key="value" AND node_1.key2="value2"
RETURN node_1
```

Listing 5.12 shows the Cypher query for the defined unspecified node. The additional key-value pair is added with an additional check in the WHERE statement. It is concatenated with the keyword AND.

Key-Value-Color Triplets

In the presented graph editor, additional key-value-color triplets can be defined. They allow global coloring of elements matching a defined condition.

Figure 5.4. A colored node in the graph editor

Figure 5.4 shows the unspecified node defined in Listing 5.11 with a coloring. This coloring is achieved with a key-value-color triplet.
5. Translation of the Candidate Pattern Graph

**Listing 5.13. The XML serialization representing Figure 5.4**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<propertygraph:xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:propertygraph="http://de.cau.se.grapheditor.propertygraph/propertygraph"
nodeCounter="1">
  <nodes label="ENTRY_LABEL" color="5B92E5" ID="node_1">
    <keyValuePairs key="key" value="value"/>
    <keyValuePairs key="key2" value="value"/>
  </nodes>
  <keyValueColorTriplets key="key2" value="value" color="5B92E5"/>
</propertygraph:xmi:propertygraph>
```

Listing 5.13 highlights the changes in the XML serialization. The text in dark blue (Line 9) describes the key-value-color triplet. The text in light blue shows the dependencies and relations of properties and the key-value-color triplet. The triplet relates to the key-value pair in Line 7 and changes the color property of the parent node described in Line 5.

**Listing 5.14. The Cypher query for Listing 5.13**

```cypher
MATCH (node_1)
WHERE node_1.key="value" AND node_1.key2="value"
RETURN node_1
```

Listing 5.14 shows the corresponding Cypher query for Listing 5.13. It is the same as Listing 5.12. In Section 5.1.1, we already noticed that the color and the ID property of an element do not influence the Cypher query. Because key-value-color triplets only affects the color property, they can be ignored completely in the translation process.

### 5.2.2 Multiple Edges

We already defined edges connecting two nodes in Section 5.1.2. In the following, we analyze paths, which may continue, fork or join at a node. At last, we describe the translation of multi edges.

#### Three Nodes in a Row

Nodes can have incoming and outgoing edges. Figure 5.5 shows a graph with a node having both. The nodes and edges are unspecified. For a better distinction, we add a label to the edges.

![Figure 5.5. A continuing path in a graph](image)

36
5.2. Translation of Advanced Constructs

Listing 5.15 shows the corresponding XML serialization for Figure 5.5. It is similar to Listing 5.5, except for an additional node with an edge and labels for edges. The overall structure remains the same.

```
Listing 5.15. The XML serialization representing Figure 5.5
1 <xml version="1.0" encoding="UTF-8"?>
2 <propertygraph:Propertygraph xmi:version="2.0"
3 xmlns:xmi="http://www.omg.org/XMI"
4 xmlns:propertygraph="http://de.cau.se.grapheditor.propertygraph/propertygraph"
5 nodeCounter="3" edgeCounter="2">
6 <nodes label="UNSPECIFIED" color="FFFFFF" ID="node_1">
7   <outgoingEdges label="edge_1" color="000000" ID="edge_1"
8     targetNode="node_2"/>
9   </nodes>
10 <nodes label="UNSPECIFIED" color="FFFFFF" ID="node_2" incomingEdges="edge_1">
11   <outgoingEdges label="edge_2" color="000000" ID="edge_2"
12     targetNode="node_3"/>
13   </nodes>
14 <nodes label="UNSPECIFIED" color="FFFFFF" ID="node_3" incomingEdges="edge_2"/>
15 </propertygraph:Propertygraph>
```

Listing 5.16 shows a naive translation of Listing 5.15. Additional edges and nodes are added in the MATCH and treated like a new query. A comma separates both graph structures, which share the node node_2. This composition allows an easy translation for every edge.

```
Listing 5.16. A naive Cypher query for Listing 5.15
1 MATCH (node_1)-[edge_1]->(node_2),
2 (node_2)-[edge_2]->(node_3)
3 RETURN node_1, edge_1, node_2, edge_2, node_3
```

Listing 5.17 shows another translation for Listing 5.15. Additional edges and nodes are concatenated in the MATCH statement and added in the RETURN statement. This composition represents the structure of the graph the most.

```
Listing 5.17. A Cypher query for Listing 5.15
1 MATCH (node_1)-[edge_1]->(node_2)-[edge_2]->(node_3)
2 RETURN node_1, edge_1, node_2, edge_2, node_3
```

One Node to Two Nodes

A node can also have two outgoing edges. In this case, a path forks at this node. Figure 5.6 shows this fork in the graph editor. The edges are labeled again for a better distinction.
5. Translation of the Candidate Pattern Graph

Listing 5.18 shows the corresponding XML serialization for Figure 5.6. The node defined in Lines 6-11 has two outgoing edges. This is comparable to two key-value pairs. In Line 12 and Line 13, two additional nodes are defined. The first of them has the edge defined in Line 7 as an incoming edge. The second one has the edge defined in Line 9 as an incoming edge.

There are multiple possibilities to translate this XML serialization into a Cypher query. Listing 5.19 shows a first approach. Relationships can also be defined in a reversed order. Having two outgoing edges from node_1, we can define relationships in two different directions. This works for having only one fork in a Candidate Pattern, but breaks at more forks.
Our second approach is shown in Listing 5.20. Each path is handled in a separated \texttt{MATCH} statement. The \texttt{WITH} statement allows to connect multiple queries. It acts like a border, where the passing variables are defined. In Line 2, the variables \texttt{node\_1}, \texttt{edge\_1} and \texttt{node\_2} are carried to the new query beginning in Line 3. Thereby, \texttt{node\_1} in Line 3 is the same as \texttt{node\_1} in the first line. Using this approach keeps the code clean, but adds two lines of code for each fork. Additionally, it has to be considered, which variables have to be passed to the upcoming query.

\textbf{Listing 5.20. Another possible Cypher query for Listing 5.18}

\begin{verbatim}
1 \texttt{MATCH (node\_1)-[edge\_1]->(node\_2)}
2 \texttt{WITH node\_1, edge\_1, node\_2}
3 \texttt{MATCH (node\_1)-[edge\_2]->(node\_3)}
4 \texttt{RETURN node\_1, edge\_1, node\_2, edge\_2, node\_3}
\end{verbatim}

Our final approach is shown in Listing 5.21. Again, each path is handled separately. A comma in a \texttt{MATCH} statement allows to concatenate multiple graph structures. Similar to our previous approach, \texttt{node\_1} represents the same node. This approach results in a long \texttt{MATCH} statement for multiple forks. For each comma it requires a node twice, but adds no additional lines. Furthermore, without having to consider which variables to pass, it allows an easier automated generation.

\textbf{Listing 5.21. Our Cypher query for Listing 5.18}

\begin{verbatim}
1 \texttt{MATCH (node\_1)-[edge\_1]->(node\_2), (node\_1)-[edge\_2]->(node\_3)}
2 \texttt{RETURN node\_1, edge\_1, node\_2, edge\_2, node\_3}
\end{verbatim}

We can also use this approach for the next advanced construct for the same reasons. The advantages and disadvantages stay the same.

\textbf{Two Nodes to One Node}

Instead of having two outgoing edges, a node can have two incoming edges. In this case, two paths merge in one node. Figure 5.7 shows this construct in the graph editor. Only \textit{unspecified} nodes and \textit{unspecified} edges are used.

\textbf{Figure 5.7. A node with two incoming edges}
Listing 5.22. The XML serialization representing ??

```xml
<?xml version="1.0" encoding="UTF-8"?>
<propertygraph xmlns:xmi="http://www.omg.org/XMI"
xmlns:propertygraph="http://de.cau.se.grapheditor.propertygraph/propertygraph"
nodeCounter="3" edgeCounter="2">
  <nodes label="UNSPECIFIED" color="FFFFFF" ID="node_1">
    <outgoingEdges label="edge_1" color="000000" ID="edge_1">
      targetNode="node_3" />
  </nodes>

  <nodes label="UNSPECIFIED" color="FFFFFF" ID="node_2">
    <outgoingEdges label="edge_2" color="000000" ID="edge_2">
      targetNode="node_3" />
  </nodes>

  <nodes label="UNSPECIFIED" color="FFFFFF" ID="node_3">
    incomingEdges="edge_1 edge_2" />
</propertygraph>```

Listing 5.22 shows the corresponding XML serialization for Figure 5.7. The additional node and the additional edge are highlighted in dark blue. Line 10 defines a node in the property graph. In Line 15, the node has two incoming edges instead of one.

This construct allows the same approaches as described in Section 5.2.2. This results in having multiple ways to describe the graph with Cypher. Because of the easier implementation, we decide to translate to the Cypher query shown in Listing 5.23.

Listing 5.23. The Cypher query for Listing 5.22

```cypher
MATCH (node_1)-[edge_1]->(node_3), (node_2)-[edge_2]->(node_3)
RETURN node_1, edge_1, node_2, edge_2, node_3
```

Vice Versa

Two nodes can have a relationship, visualized by an edge. When there is one outgoing edge for every node, and one incoming edge for every node, we get the graph shown in Figure 5.8.

![Figure 5.8: Two nodes with reciprocally relationships](image.png)
This special case can be divided into two sub-cases, depending on the type of the edges. In the following, we will show the XML serialization for two unspecified edges.

**Listing 5.24.** The XML serialization representing Figure 5.7

```xml
<?xml version="1.0" encoding="UTF-8"?>
<propertygraph:xmi:xmi version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:propertygraph="http://de.cau.se.grapheditor.propertygraph/propertygraph"
nodeCounter="2" edgeCounter="2">
<nodes label="UNSPECIFIED" color="FFFFFF" ID="node_1"
incomingEdges="edge_2">
  <outgoingEdges label="edge_1" color="000000" ID="edge_1"
targetNode="node_2"/>
</nodes>
<nodes label="UNSPECIFIED" color="FFFFFF" ID="node_2"
incomingEdges="edge_1">
  <outgoingEdges label="edge_2" color="000000" ID="edge_2"
targetNode="node_1"/>
</nodes>
</propertygraph:xmi:xmi>
```

Listing 5.24 shows the XML serialization representing Figure 5.8. The two nodes defined in Line 6-9 and Line 10-13 are nearly identical. They only differ in their IDs and the label of the edges. Depending on the **type** and **label** of the edges, the code can be translated in two different **Cypher** queries.

**Listing 5.25.** A **Cypher** query for Listing 5.24

```cypher
MATCH (node_1)-[edge_1]->(node_2)-[edge_2]->(node_1)
RETURN node_1, edge_1, node_2, edge_2
```

Listing 5.25 shows the **Cypher** query for the XML serialization shown in Listing 5.24. It is similar to Listing 5.17, but the last node in the **MATCH** statement is the same as the first node.

**Listing 5.26.** A **Cypher** query for Listing 5.24

```cypher
MATCH (node_1)-[edge_1]-(node_2)
RETURN node_1, edge_1, node_2
```

Assuming that `edge_1` and `edge_2` would have the same **type**, there exists another translation for the XML code. Listing 5.25 shows this translation. In **Neo4j**, we have the possibility to represent bidirectional relationships by leaving out the arrow of an edge. This allows to shorten queries.
5. Translation of the Candidate Pattern Graph

A Path of Multiple Edges

So far, we translated relationships with one edge. Figure 5.9 shows a path of undefined length between two nodes. This requires the modeling of every node and edge on the path, if the length of the path is known.

![Figure 5.9. A path of undefined length](image)

If the nodes on this path are negligible, we can use the multi edge introduced in Section 4.4. This requires either all edges on this path to have the same type, or the type of the edges is not important for the pattern. Figure 5.10 shows a multi edge in the graph editor.

![Figure 5.10. A multi edge between two nodes](image)

Listing 5.27 shows the XML serialization of Figure 5.10. In Line 6-11, the first node is defined. The node has an outgoing edge, which is defined in Line 7-10. Line 9 defines the key-value pair representing the multi edge. The value n-m represents the length of the path. n and m are placeholders, where n is the minimum amount of edges and m is the maximum. In Line 12 and Line 13, the second node is defined.

```
<?xml version="1.0" encoding="UTF-8"?>
<propertygraph xmlns:xmi="http://www.omg.org/XMI"
    xmlns:propertygraph="http://de.cau.se.grapheditor.propertygraph/propertygraph"
    nodeCounter="2" edgeCounter="1">
  <nodes label="UNSPECIFIED" color="FFFFFF" ID="node_1">
    <outgoingEdges label="MULTIEDGE" color="000000" ID="edge_1"
      targetNode="node_2">  
      <keyValuePairs key="multiedge" value="n-m"/>
    </outgoingEdges>
  </nodes>

  <nodes label="UNSPECIFIED" color="FFFFFF" ID="node_2"
    incomingEdges="edge_1"/>
</propertygraph>
```

Listing 5.27. The XML serialization representing Figure 5.10
5.3. Translation of Forbidden Elements

Listing 5.28 shows the corresponding Cypher query for Listing 5.27. The length of the path, specified by \( n \) and \( m \), is added to the edge in the MATCH statement. It is starting with a * symbol and separated by two dots. \( n \) and \( m \) are both optional, so no borders have to be defined. When leaving out both, the dots have to be left out. Because edge_1 can be more than one edge, the RETURN statement now returns a list of all edges called edges_1.

<table>
<thead>
<tr>
<th>Listing 5.28. The Cypher query for Listing 5.27</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MATCH (node_1)-[edge1*n..m]-&gt;(node_2)</td>
</tr>
<tr>
<td>2 RETURN node_1, edge_1 AS edges_1, node_2</td>
</tr>
</tbody>
</table>

Listing 5.29 shows another Cypher query for a multi edge. This time, the path is not limited by any borders. Additionally, all edges on the path must have the type example.

<table>
<thead>
<tr>
<th>Listing 5.29. A Cypher query for Listing 5.27 with no borders</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MATCH (node_1)-[edge_1:example*]-&gt;(node_2)</td>
</tr>
<tr>
<td>2 RETURN node_1, edge_1 AS edges_1, node_2</td>
</tr>
</tbody>
</table>

5.3 Translation of Forbidden Elements

Having implemented the forbidden elements in Section 4.5, their translation to a Neo4j Cypher query is required. At first, we translate forbidden properties. Then, forbidden types are translated. This subsection is divided in forbidden types for nodes and forbidden types for edges. At last, we translate forbidden edges.

5.3.1 Forbidden Properties

Listing 5.30 shows the XML serialization representing a node with a forbidden property. In Line 6-8, the node is defined. Line 7 shows the key-value pair of the node. It has the key forbiddenkey and the value value. A forbidden key is specified by the prefix forbidden in the key field. This also applies to key-value pairs of edges.

<table>
<thead>
<tr>
<th>Listing 5.30. The XML serialization representing a node with a forbidden property</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 &lt;?xml version=&quot;1.0&quot; encoding=&quot;UTF-8&quot;?&gt;</td>
</tr>
<tr>
<td>2 &lt;propertygraph:Propertygraph xmlns:version=&quot;2.0&quot;</td>
</tr>
<tr>
<td>3 xmlns:xmi=&quot;<a href="http://www.omg.org/XMI">http://www.omg.org/XMI</a>&quot;</td>
</tr>
<tr>
<td>4 xmlns:propertygraph=&quot;<a href="http://de.cau.se.grapheditor.propertygraph/propertygraph%22">http://de.cau.se.grapheditor.propertygraph/propertygraph&quot;</a></td>
</tr>
<tr>
<td>5 nodeCounter=&quot;1&quot;&gt;</td>
</tr>
<tr>
<td>6   &lt;nodes label=&quot;UNSPECIFIED&quot; color=&quot;#FFFFFF&quot; ID=&quot;node_1&quot;&gt;</td>
</tr>
<tr>
<td>7     &lt;keyValuePairs key=&quot;forbiddenkey&quot; value=&quot;value&quot;/&gt;</td>
</tr>
<tr>
<td>8   &lt;/nodes&gt;</td>
</tr>
<tr>
<td>9 &lt;/propertygraph:Propertygraph&gt;</td>
</tr>
</tbody>
</table>
5. Translation of the Candidate Pattern Graph

Listing 5.31 shows the Cypher query for Listing 5.30. Neo4j offers a function `exists(args)`, which returns a boolean. It checks if a property (args) exists. In this case, Line 2 checks for the property `key` of the element `node_1`. The keyword `NOT` negates the statement.

<table>
<thead>
<tr>
<th>Listing 5.31. The Cypher query for Listing 5.30</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MATCH (node_1)</td>
</tr>
<tr>
<td>2 WHERE NOT exists(node_1.key)</td>
</tr>
<tr>
<td>3 RETURN node_1</td>
</tr>
</tbody>
</table>

5.3.2 Forbidden Types

As shown in Section 4.5, we implement additional key-value-color triplets to color nodes or edges with forbidden types. Because of their different visual representation, we divide this subsection in nodes and edges.

Nodes with Forbidden Types

Figure 5.11 shows the visualization of a node with a forbidden type in the graph editor. The node has the type `example` and the label `EXAMPLE`. A key-value-color triplet causes the node to be colored red.

![EXAMPLE](image)

Figure 5.11. A node with a forbidden type

Listing 5.32 shows the XML serialization of Figure 5.11. In Line 6-9, the node is defined. Line 7 defines the type of the node. In Line 8, another key-value pair is defined. It has the key `forbiddentype` and the value `FORBIDDEN`. In Line 10, a key-value-color triplet is defined. It causes all matching nodes and edges to be colored red (hexadecimal `0xFF0000`).

<table>
<thead>
<tr>
<th>Listing 5.32. The XML serialization representing Figure 5.11</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 &lt;?xml version=&quot;1.0&quot; encoding=&quot;UTF-8&quot;?&gt;</td>
</tr>
<tr>
<td>2 &lt;propertygraph:Propertygraph xmi:version=&quot;2.0&quot;</td>
</tr>
<tr>
<td>3 xmlns:xmi=&quot;<a href="http://www.omg.org/XMI">http://www.omg.org/XMI</a>&quot;</td>
</tr>
<tr>
<td>4 xmlns:propertygraph=&quot;<a href="http://de.cau.se.grapheditor.propertygraph/propertygraph">http://de.cau.se.grapheditor.propertygraph/propertygraph</a>&quot;</td>
</tr>
<tr>
<td>5 nodeCounter=&quot;1&quot;&gt;</td>
</tr>
<tr>
<td>6 &lt;nodes label=&quot;EXAMPLE&quot; color=&quot;FF0000&quot; ID=&quot;node_1&quot;&gt;</td>
</tr>
<tr>
<td>7   &lt;keyValuePairs key=&quot;type&quot; value=&quot;example&quot;/&gt;</td>
</tr>
<tr>
<td>8   &lt;keyValuePairs key=&quot;forbiddentype&quot; value=&quot;FORBIDDEN&quot;/&gt;</td>
</tr>
<tr>
<td>9 &lt;/nodes&gt;</td>
</tr>
<tr>
<td>10 &lt;keyValueColorTriplets key=&quot;forbiddentype&quot; value=&quot;FORBIDDEN&quot; color=&quot;FF0000&quot;/&gt;</td>
</tr>
<tr>
<td>11 &lt;/propertygraph:Propertygraph&gt;</td>
</tr>
</tbody>
</table>
5.3. Translation of Forbidden Elements

Listing 5.33 shows the Cypher query for Listing 5.32. In the WHERE statement, the keyword NOT negates the statement. If we check for a type of a node, we check for a property.

<table>
<thead>
<tr>
<th>Listing 5.33. The Cypher query for Listing 5.32</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MATCH (node_1)</td>
</tr>
<tr>
<td>2 WHERE NOT exists(node_1.type)</td>
</tr>
<tr>
<td>3 RETURN node_1</td>
</tr>
</tbody>
</table>

Edges with Forbidden Types

Figure 5.12 shows the visualization of an edge with a forbidden type in the graph editor. The edge connects two UNSPECIFIED nodes. The edge has the type example and the label EXAMPLE. A key-value-color triplet causes the edge to be colored red.

![Figure 5.12. An edge with a forbidden type](image)

Listing 5.34 shows the XML serialization of Figure 5.12. In Line 6-12, the first node is defined. It has an outgoing edge defined in Line 7-11. Line 9 defines the type of the edge. In Line 10, another key-value pair is defined. It has the key forbiddentype and the value FORBIDDEN. In Line 13, the second node is defined. It has edge_1 as an incoming edge. In Line 14, a key-value-color triplet is defined. It causes all matching nodes and edges to be colored red (hexadecimal \(0xFF0000\)).

<table>
<thead>
<tr>
<th>Listing 5.34. The XML serialization representing Figure 5.12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 &lt;?xml version=&quot;1.0&quot; encoding=&quot;UTF-8&quot;?&gt;</td>
</tr>
<tr>
<td>2 &lt;propertygraph:xmi xmlns:xmi=&quot;<a href="http://www.omg.org/XMI">http://www.omg.org/XMI</a>&quot;</td>
</tr>
<tr>
<td>3 xmlns:propertygraph=&quot;<a href="http://de.cau.se.grapheditor.propertygraph/propertygraph">http://de.cau.se.grapheditor.propertygraph/propertygraph</a>&quot;</td>
</tr>
<tr>
<td>4 nodeCounter=&quot;2&quot; edgeCounter=&quot;1&quot;&gt;</td>
</tr>
<tr>
<td>5 &lt;nodes label=&quot;UNSPECIFIED&quot; color=&quot;FFFFFF&quot; ID=&quot;node_1&quot;&gt;</td>
</tr>
<tr>
<td>6  &lt;outgoingEdges label=&quot;EXAMPLE&quot; color=&quot;FF0000&quot; ID=&quot;edge_1&quot;</td>
</tr>
</tbody>
</table>
| 7  <keyValuePairs key="type" value="example="/>
| 8  <keyValuePairs key="forbiddentype" value="FORBIDDEN="/>
| 9  </outgoingEdges>
| 10 </nodes>
| 11 <nodes label="UNSPECIFIED" color="FFFFFF" ID="node_2" incomingEdges="edge_1">|
| 12  <keyValueColorTriplets key="forbiddentype" value="FORBIDDEN" color="FF0000">|
| 13 </propertygraph:xmi>                                      |
5. Translation of the Candidate Pattern Graph

Listing 5.35 shows the Cypher query for Listing 5.34. In the WHERE statement, the keyword NOT negates the statement. If we check for a type of an edge, we check for a label. Therefore, we do not require the exists function.

<table>
<thead>
<tr>
<th>Listing 5.35. The Cypher query for Listing 5.34</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MATCH (node_1)</td>
</tr>
<tr>
<td>2     WHERE NOT edge_1:example</td>
</tr>
<tr>
<td>3     RETURN node_1</td>
</tr>
</tbody>
</table>

5.3.3 Forbidden Edges

Figure 5.13 shows the visualization of a forbidden edge in the graph editor. The edge connects two UNSPECIFIED nodes. The edge is UNSPECIFIED and the label FORBIDDEN. A key-value-color triplet causes the edge to be colored red.

![Figure 5.13. A forbidden edge connecting two nodes](image)

Listing 5.36 shows the XML serialization of Figure 5.13. In Line 6-11, the first node is defined. It has an outgoing edge defined in Line 7-10. In Line 9, a key-value pair with the key forbiddenedge and the value FORBIDDEN is defined. In Line 12, the second node is defined. It has edge_1 as an incoming edge. In Line 13, a key-value-color triplet is defined. It causes all matching nodes and edges to be colored red (hexadecimal 0xFF0000).

<table>
<thead>
<tr>
<th>Listing 5.36. The XML serialization representing Figure 5.13</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  &lt;xml version=&quot;1.0&quot; encoding=&quot;UTF-8&quot;?&gt;</td>
</tr>
<tr>
<td>2  &lt;propertygraph:Propertygraph xmi:version=&quot;2.0&quot;</td>
</tr>
<tr>
<td>3   xmlns:xmi=&quot;<a href="http://www.omg.org/XMI">http://www.omg.org/XMI</a>&quot;</td>
</tr>
<tr>
<td>4   xmlns:propertygraph=&quot;<a href="http://de.cau.se.grapheditor.propertygraph/propertygraph">http://de.cau.se.grapheditor.propertygraph/propertygraph</a>&quot;</td>
</tr>
<tr>
<td>5 nodeCounter=&quot;2&quot; edgeCounter=&quot;1&quot;&gt;</td>
</tr>
<tr>
<td>6  &lt;nodes label=&quot;UNSPECIFIED&quot; color=&quot;FFFFF&quot; ID=&quot;node_1&quot;&gt;</td>
</tr>
<tr>
<td>7    &lt;outgoingEdges label=&quot;FORBIDDEN&quot; color=&quot;FF0000&quot; ID=&quot;edge_1&quot; targetNode=&quot;node_2&quot;/&gt;</td>
</tr>
<tr>
<td>9    &lt;keyValuePairs key=&quot;forbiddenedge&quot; value=&quot;FORBIDDEN&quot;/&gt;</td>
</tr>
<tr>
<td>10 &lt;/outgoingEdges&gt;</td>
</tr>
<tr>
<td>11 &lt;/nodes&gt;</td>
</tr>
<tr>
<td>12 &lt;nodes label=&quot;UNSPECIFIED&quot; color=&quot;FFFFF&quot; ID=&quot;node_2&quot; incomingEdges=&quot;edge_1&quot;/&gt;</td>
</tr>
<tr>
<td>13    &lt;keyValueColorTriplets key=&quot;forbiddenedge&quot; value=&quot;FORBIDDEN&quot; color=&quot;FF0000&quot;/&gt;</td>
</tr>
<tr>
<td>14 &lt;/propertygraph:Propertygraph&gt;</td>
</tr>
</tbody>
</table>
5.3. Translation of Forbidden Elements

Listing 5.37 shows the Cypher query for Listing 5.36. In the `WHERE` statement, a relationship is defined. It is similar to the relationships we defined in the `MATCH` statements so far. The keyword `NOT` negates the statement.

<table>
<thead>
<tr>
<th>Listing 5.37. The Cypher query for Listing 5.36</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MATCH (node_1), (node_2)</td>
</tr>
<tr>
<td>2 WHERE NOT (node_1)-[edge_1]-&gt;(node_2)</td>
</tr>
<tr>
<td>3 RETURN node_1, node_2</td>
</tr>
</tbody>
</table>
Translation of the User Interaction Protocol

The User Interaction Protocol (UIP) is a list of the changes made to a graph created in the graph editor. Therefore, it records the transformation of a Candidate Pattern to a Parallelization Pattern. In order to apply the transformation to the SDG, we require a Cypher query for the UIP. As a result, new Cypher operations are required and a different translation mechanism has to be implemented.

Figure 6.1. Analyzing and translating the UIP

Figure 6.1 shows our approach to create the translation algorithm for UIPs. We use the same approach as in Chapter 5. At first, we translate every UIP entry to a Cypher query. Then, we have a look at some special behavior of the UIP. This leads to possibilities for optimization in translation.

6.1 Translation of the Protocol

Deleting nodes, adding nodes, and editing properties requires different operations in Cypher. Those operations work on variables of the object to be manipulated. If the variable has not been declared in the current query, the operation requires an additional MATCH WHERE statement to declare such a variable. The id given by the UIP refers to an object in the graph editor. It is not similar to the id of the object in the Neo4j SDG. Cypher handles this id as a variable, because it has no reference to the defined object. In the following, we will group the different operations of the UIP and translate them to a Cypher query. The operations in each group are similar in their translation.
6. Translation of the User Interaction Protocol

6.1.1 Adding Elements

When creating a new element, a variable is associated with that element. Here, the \textit{id} given by the graph editor is used for that purpose. During the current query, the element can be modified using this variable.

\textbf{NODE_ELEM_ADD}

UIP history: NODE_ELEM_ADD, \textit{id}

Cypher: CREATE (\textit{id})

\textbf{EDGE_ELEM_ADD}

UIP history: EDGE_ELEM_ADD, \textit{id}, source\_ID, target\_ID

Cypher: CREATE (source\_ID)-[\textit{id}]->(target\_ID)

6.1.2 Removing Elements

A removal has the same Cypher syntax for nodes and edges. Therefore, \textit{<ELEMENT>} refers to either \texttt{NODE} or \texttt{EDGE}. There is a special case for nodes with attached edges, which is shown in Section 6.2.1

\textit{<ELEMENT>\_ELEM_DEL}

UIP history: NODE\_ELEM\_DEL/EDGE\_ELEM\_DEL, \textit{id}

Cypher: DELETE \textit{id}

6.1.3 Editing Properties

The graph editor uses key-value pairs to store properties of elements. \texttt{Cypher} does not differentiate between setting and updating a key-value pair. While the UIP differentiates between nodes and edges, Cypher does not. Again we refer to nodes and edges as \textit{<ELEMENT>}.

\textit{<ELEMENT>\_PROP_ADD}

UIP history: NODE\_PROP\_ADD/EDGE\_PROP\_ADD, \textit{id}, \textit{key}, value

Cypher: SET \textit{id}.\textit{key} = "value"

\textit{<ELEMENT>\_PROP_SET}

UIP history: NODE\_PROP\_SET/EDGE\_PROP\_SET, \textit{id}, \textit{key}, new\_value

Cypher: SET \textit{id}.\textit{key} = "new\_value"
6.1. Translation of the Protocol

<ELEMENT>_PROP_DEL
UIP history: NODE_PROP_DEL/EDGE_PROP_DEL, id, key
Cypher: REMOVE id.key

6.1.4 Editing Types

If the key of a property to set is “type”, the syntax for edges differs from editing other key-value pairs. This is caused by using a label instead of a property to define a type. In Neo4j edges are required to always have exactly one label. Therefore, edges require a label at initialization. EDGEPROP_ADD requires a preceding EDGE_ELEM_ADD operation in the same UIP and vice versa. The operations have to be combined as shown in the following.

Due to the fact that edges always must have a label, the label can not be changed after creation. Therefore, a change of the label requires the creation of a new edge. This new edge is initialized with the new label. It receives all properties of the old edge. Then, the old edge is deleted. EDGEPROP_DEL is an illegal operation for the property type.

NODE_PROP_ADD
UIP history: NODEPROP_ADD, id, type, value
Cypher: SET id.type = "value"

EDGE_PROP_ADD
UIP history: EDGE_ELEM_ADD, id, source_ID, target_ID
EDGE_PROP_ADD, id, type, value
Cypher: CREATE (source_ID)-[id:value]->(target_ID)

NODE_PROP_SET
UIP history: NODE_PROP_SET, id, type, new_value
Cypher: SET id.type = "new_value"

EDGE_PROP_SET
UIP history: EDGE_PROP_SET, id, type, new_value
Cypher: MATCH (n)-[id:old_value]->(m)
CREATE (n)-[r:new_value]->(m)
SET r = id
DELETE id
6. Translation of the User Interaction Protocol

NODE_PROP_DEL
UIP history: NODE_PROP_DEL/EDGE_PROP_DEL, id, type
Cypher: REMOVE id.type

6.2 Special Behavior of the User Interaction Protocol

Deleting elements in the graph editor often removes more than one element. This behavior is partly represented in the User Interaction Protocol and may be simplified in a Cypher query.

6.2.1 Removing Connected Elements

The graph editor allows to delete nodes and edges. When deleting an edge, the edge is removed. When deleting a node, the node and all attached edges are deleted. Cypher refuses to delete nodes, if they still have edges attached to them.

Figure 6.2. Deleting a node with connected edges

Figure 6.2 shows two instances of Eclipse. The instance in the background shows a graph with three nodes and two edges. The instance in the foreground shows the same graph
6.2. Special Behavior of the User Interaction Protocol

after removing node_2. When deleting node_2, the edges edge_1 and edge_2 are removed, too. All elements shown are unspecified elements.

Listing 6.1 shows the recorded UIP. If a node has attached edges, they are removed first (Line 1 and Line 2). In Line 3, the node itself is deleted.

<table>
<thead>
<tr>
<th>Listing 6.1. The recorded User Interaction Protocol for Figure 6.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 EDGE_ELEM_DEL, edge_1</td>
</tr>
<tr>
<td>2 EDGE_ELEM_DEL, edge_2</td>
</tr>
<tr>
<td>3 NODE_ELEM_DEL, node_2</td>
</tr>
</tbody>
</table>

A naive translation would mimic the UIP step by step. Listing 6.2 shows this translation. The order of the protocol is preserved.

<table>
<thead>
<tr>
<th>Listing 6.2. The naive Cypher query for Listing 6.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 DELETE edge_1</td>
</tr>
<tr>
<td>2 DELETE edge_2</td>
</tr>
<tr>
<td>3 DELETE node_2</td>
</tr>
</tbody>
</table>

Neo4j’s Cypher allows to delete nodes and remove all attached edges with the DETACH DELETE command. Hence, edges do not have to be treated in an additional statement, if their source or target is a deleted node. Listing 6.3 shows the recommend translation for Listing 6.1.

<table>
<thead>
<tr>
<th>Listing 6.3. The recommend Cypher query for Listing 6.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 DETACH DELETE node_2</td>
</tr>
</tbody>
</table>

6.2.2 Removing Elements with Key-Value Pairs

Both nodes and edges may store additional information in key-value pairs. When deleting one of these elements in the graph editor, the key-value pairs are removed, too.
6. Translation of the User Interaction Protocol

Removing Nodes with Key-Value Pairs
Listing 6.4 shows the XML serialization of an unspecified node. In Line 7, it has an additional key-value pair with the key `nodeNumber` and the value 1.

<table>
<thead>
<tr>
<th>Listing 6.4. An unspecified node with a key-value pair</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 &lt;?xml version=&quot;1.0&quot; encoding=&quot;UTF-8&quot;?&gt;</td>
</tr>
<tr>
<td>2 &lt;propertygraph:Propertygraph xmi:version=&quot;2.0&quot;</td>
</tr>
<tr>
<td>3 xmlns:xmi=&quot;<a href="http://www.omg.org/XMI">http://www.omg.org/XMI</a>&quot;</td>
</tr>
<tr>
<td>4 xmlns:propertygraph=&quot;<a href="http://de.cau.se.grapheditor.propertygraph/propertygraph">http://de.cau.se.grapheditor.propertygraph/propertygraph</a>&quot;</td>
</tr>
<tr>
<td>5 nodeCounter=&quot;1&quot;&gt;</td>
</tr>
<tr>
<td>6 &lt;nodes label=&quot;node_1&quot; color=&quot;FFFFFF&quot; ID=&quot;node_1&quot;&gt;</td>
</tr>
<tr>
<td>7 &lt;keyValuePairs key=&quot;nodeNumber&quot; value=&quot;1&quot;/&gt;</td>
</tr>
<tr>
<td>8 &lt;/propertygraph:Propertygraph&gt;</td>
</tr>
</tbody>
</table>

Listing 6.5 shows the UIP when deleting `node_1` in Listing 6.4. While there is an entry for deleting the node, its key-value pairs are ignored.

<table>
<thead>
<tr>
<th>Listing 6.5. The UIP for deleting <code>node_1</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 NODE_ELEM_DEL, node_1</td>
</tr>
</tbody>
</table>

Listing 6.6 shows the according Cypher query for Listing 6.5. Deleting a node in Cypher also deletes its attributes. Therefore, no further operations are required.

<table>
<thead>
<tr>
<th>Listing 6.6. The Cypher query for Listing 6.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 DELETE node_1</td>
</tr>
</tbody>
</table>

Removing Edges with Key-Value Pairs
Listing 6.7 shows the XML serialization of an unspecified edge connecting two unspecified nodes. In Line 9, the edge has an additional key-value pair with the `key` `edgeNumber` and the `value` 1.
6.3 Optimization of the User Interaction Protocol

Listing 6.7. Two nodes connected via an edge with a key-value pair

```xml
<?xml version="1.0" encoding="UTF-8"?>
<propertygraph xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:propertygraph="http://de.cau.se.grapheditor.propertygraph/propertygraph"
nodeCounter="2" edgeCounter="1">

  <nodes label="node_1" color="FFFFFF" ID="node_1">
    <outgoingEdges label="edge_1" color="000000" ID="edge_1"
      targetNode="node_2">
      <keyValuePairs key="edgeNumber" value="1"/>
    </outgoingEdges>
  </nodes>

  <nodes label="node_2" color="FFFFFF" ID="node_2" incomingEdges="edge_1"/>
</propertygraph:xmi:propertygraph>
```

Listing 6.8 shows the UIP when deleting edge_1 in Listing 6.7. Again, there are no further actions taken for deleting the key-value pair.

Listing 6.8. The UIP for deleting edge_1

```text
EDGE_ELEM_DEL, edge_1
```

Listing 6.9 shows the according Cypher query for Listing 6.8. Deleting an edge in Cypher also deletes its attributes. Therefore, no further operations are required.

Listing 6.9. The Cypher query for Listing 6.8

```cypher
DELETE edge_1
```

6.3 Optimization of the User Interaction Protocol

Throughout the process of creating a Parallelization Pattern, redundant or unnecessary protocol entries may occur. For example, an object can be added and later on be deleted. This is caused by the human interaction. Additionally, some operations can make previous operations superfluous, which we explain in the following sections. Multiple operations may also be combined in one operation. In the following, we present the possible optimization of the UIP before translating it into a Cypher query.

6.3.1 Optimization of Property Operations

Property entries in the UIP allow a limited optimization on a low level. Because Cypher does not differentiate between nodes and edges when editing properties, we again refer to these two objects as `<ELEMENT>`.
6. Translation of the User Interaction Protocol

Multiple Updates

Listing 6.10 shows a UIP with two update operations. In Line 1, the key-value pair with the key type of the element elem is updated to the value example. In Line 2, the value of this key is updated to newexample.

<table>
<thead>
<tr>
<th>Listing 6.10. A UIP to demonstrate the optimization of multiple updates</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. <code>&lt;ELEMENT&gt;_PROP_SET, elem, type, example</code></td>
</tr>
<tr>
<td>2. <code>&lt;ELEMENT&gt;_PROP_SET, elem, type, newexample</code></td>
</tr>
</tbody>
</table>

If a UIP contains multiple updates on the same key, each update overwrites the previous update. In conclusions, we only require the latest property update in the UIP. Hence, Listing 6.10 can be optimized to Listing 6.11.

<table>
<thead>
<tr>
<th>Listing 6.11. An optimized UIP basing on Listing 6.10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. <code>&lt;ELEMENT&gt;_PROP_SET, elem, type, newexample</code></td>
</tr>
</tbody>
</table>

Updates for Previously Added Properties

Listing 6.12 shows a UIP with two operations concerning properties. In Line 1, the key-value pair with the key type and the value example is added to the element elem. In Line 2, the value of this key is updated to newexample.

<table>
<thead>
<tr>
<th>Listing 6.12. A UIP to demonstrate the optimization of property updates</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. <code>&lt;ELEMENT&gt;_PROP_ADD, elem, type, example</code></td>
</tr>
<tr>
<td>2. <code>&lt;ELEMENT&gt;_PROP_SET, elem, type, newexample</code></td>
</tr>
</tbody>
</table>

If a property is added to an element in the UIP, it has an initial value. Instead of updating the value, we can remove the update operation and exchange the value in the initialization with the update. Therefore, Listing 6.12 can be optimized to Listing 6.13.

<table>
<thead>
<tr>
<th>Listing 6.13. An optimized UIP basing on Listing 6.12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. <code>&lt;ELEMENT&gt;_PROP_SET, elem, type, newexample</code></td>
</tr>
</tbody>
</table>

Removing Properties

Listing 6.14 shows a UIP with two operations concerning properties. At first, the key-value pair with the key type of the element elem is updated to the value example. Then, this key is removed from the element.
6.3. Optimization of the User Interaction Protocol

Listing 6.14. A UIP to demonstrate the optimization of property removal

1. `<ELEMENT>_PROP_ADD, elem, type, example`
2. `<ELEMENT>_PROP_DEL, elem, type`

Removing a property in the UIP makes all operations concerning this property superfluous up to the point of removal. Hence, Listing 6.14 can be optimized to Listing 6.15.

Listing 6.15. An optimized UIP basing on Listing 6.14

1. `<ELEMENT>_PROP_DEL, elem, type`

If a key-value pair has been added in the same UIP, the protocol can be optimized even more. Listing 6.16 shows a UIP with two operations concerning properties. At first, the key-value pair with the key `type` and the value `example` is added to the element `elem`. Then, this key is removed from the element.

Listing 6.16. A UIP to demonstrate optimization of property removal

1. `<ELEMENT>_PROP_ADD, elem, type, example`
2. `<ELEMENT>_PROP_DEL, elem, type`

Adding and removing a property in the UIP is like creating a temporal object. It can be ignored completely and does not have to be translated. Therefore, all operations (including `ADD` and `DEL`) concerning the property can be deleted up to the point of removal. Optimizing Listing 6.16 results in a blank UIP.

6.3.2 Optimization of Edge Operations

Edge entries in the UIP allow an optimization on a medium level. Optimizing property entries only affects the property itself. Optimizing edge entries affects the edge itself and can also affect all its properties.

Removing Edges

In the first scenario, there already exists an edge `edge`. To maintain an overview, we dispense with an additional `MATCH WHERE` statement. Listing 6.17 shows a UIP with two operations. In Line 1, a key-value pair with the key `type` and the value `example` is added to the edge `edge`. In Line 2, the edge is removed.

Listing 6.17. A UIP to demonstrate the optimization of edge removal

1. `EDGE_PROP_SET, edge, type, example`
2. `EDGE_ELEM_DEL, edge`
6. Translation of the User Interaction Protocol

Removing an edge in the UIP makes all operations concerning this edge superfluous up to the point of its removal. This applies especially to properties of the edge. Hence, Listing 6.17 can be optimized to Listing 6.18.

Listing 6.18. An optimized UIP basing on Listing 6.17

1. \texttt{EDGE_ELEM_DEL, edge}

If an edge has been added in the same UIP, the protocol can be optimized even more. In the next scenario, there already exist the nodes \texttt{node\_1} and \texttt{node\_2}. Listing 6.19 shows a UIP with two operations concerning an edge. In Line 1, the edge with the source \texttt{node\_1} and the target \texttt{node\_2} is added. In Line 2, this edge is removed.

Listing 6.19. A UIP to demonstrate the optimization of edge removal

1. \texttt{EDGE_ELEM_ADD, edge, node\_1, node\_2}
2. \texttt{EDGE_ELEM_DEL, edge}

Adding and removing an edge in the UIP is like creating a temporal object. It can be ignored completely and does not require to be translated. Therefore, all operations (including \texttt{ADD} and \texttt{DEL}) concerning the edge and its properties can be deleted up to the point of removal. Optimizing Listing 6.19 results in a blank UIP.

6.3.3 Optimization of Node Operations

Node entries in the UIP allow an optimization on a high level. Optimizing node entries affects the node itself and can also affect all its properties and connected edges, which may also affect the properties of these edges.

Removing Nodes

In the first scenario, there already exist two nodes \texttt{node\_1} and \texttt{node\_2}. For the sake of clarity, we omit displaying the obligatory \texttt{MATCH WHERE} statement in the following Cypher queries.

Listing 6.20 shows a UIP with four operations. In Line 1, a key-value pair with the key \texttt{type} and the value \texttt{example} is added to the node \texttt{node\_1}. Then, the edge \texttt{edge} connecting \texttt{node\_1} and \texttt{node\_2} is added in Line 2. In Line 3, a key-value pair with the key \texttt{type} and the value \texttt{example} is added to the edge \texttt{edge}. In Line 4, the node \texttt{node\_1} is removed.

Listing 6.20. A UIP to demonstrate optimization of node removal

1. \texttt{NODE_PROP_ADD, node\_1, type, example}
2. \texttt{EDGE_ELEM_ADD, edge, node\_1, node\_2}
3. \texttt{EDGE_PROP_ADD, edge, type, example}
4. \texttt{NODE_ELEM_DEL, node\_1}
If a node is removed, all other operations containing this node can be ignored up to the point of its removal. Because an edge requires a target node and a source node, attached edges can be deleted, too. This requires the use of the `DETACH DELETE` operation. Therefore, operations concerning the properties of the node or the attached edges can be deleted. Hence, Listing 6.20 can be optimized to Listing 6.21.

Listing 6.21. An optimized UIP basing on Listing 6.20

```
1 NODE_ELEM_DEL, node_1
```

If a node has been added in the same UIP, the protocol can be optimized even more. Like properties and edges, adding and removing a node in the UIP creates a temporal object. It can be ignored completely and does not require to be translated. Therefore, all operations (including `ADD` and `DEL`) concerning the node and its properties can be deleted up to the point of removal. Additionally, all attached edges and their properties have to be considered. So, optimizing Listing 6.22 results in a blank UIP.

Listing 6.22. A UIP to demonstrate optimization of node removal

```
1 NODE_ELEM_ADD, node_1
2 NODE_ELEM_DEL, node_1
```
Chapter 7

Architecture of the Translator

Having done the translations in the previous chapters, we have achieved conclusions on how to implement the translation mechanism. Graph structures are inductively defined by their single elements. This allows us to reduce complex structures to their basics, which we are able to translate.

At first, we develop a standalone translator. In Section 7.1.2, we explain the translation mechanism. We differentiate between the elements and structures presented in Chapter 5 and the translation of the UIP. Finally, we integrate the translation mechanism into the graph editor in Section 7.2.

7.1 The Translator

To translate the propertygraph files without the graph editor, a GUI is required. This allows quick testing without having to create a plugin for the graph editor.

![Figure 7.1. The GUI of the standalone translator](image)

Figure 7.1. The GUI of the standalone translator
7. Architecture of the Translator

Figure 7.1 shows the GUI of the translator. The button in the right upper corner \(\textcircled{1}\) allows to load `.propertygraph` and `.history` files. Therefore, a `JFileChooser` is opened as shown in Figure 7.2. Loading a file results in a visualization of the code in the main window \(\textcircled{2}\). Also, code can be copied and pasted there. Additionally, the file path \(\textcircled{3}\) is set to the current file. The button on the bottom left \(\textcircled{4}\) allows to translate the code from a loaded `.propertygraph` file. The button on the bottom right \(\textcircled{5}\) translates the code from a loaded `.history` file or pasted code. After the translation process, the translated code is shown in the main window \(\textcircled{2}\).

![Figure 7.2. The JFileChooser to load files](image)

Figure 7.2. The JFileChooser to load files

Figure 7.3 shows the packages of the translator. The GUI package uses the language package. In the following, we describe the content of each package and its purpose.

![Figure 7.3. Package diagram of the translator](image)

Figure 7.3. Package diagram of the translator
7.1. The Translator

7.1.1 GUI Package

The package GUI contains the classes providing a standalone version of the translator. They define the layout and functionality of the GUI.

![Class diagram of the GUI package]

Figure 7.4. Class diagram of the GUI package

Figure 7.4 shows a class diagram of the classes contained in the GUI package. The class GUIstarter uses the class GUIdesign to visualize the translator. The class GUIdesign has access to the methods defined in the class GUIfunctions.

GUIstarter

The class GUIstarter includes the main method of the translator. It initializes a new design provided by the class GUIdesign and sets the used language. The language is stored in the local translator variable and can be accessed by the method getTranslator().

GUIdesign

The class GUIdesign defines the frame of the GUI and its content. At initialization, the method createWindow() is called. It assembles the GUI seen in Figure 7.1 by creating the objects and setting their properties. Other classes gain access to the single elements via the getter and setter methods.
7. Architecture of the Translator

GUIfunctions

The class GUIfunctions provides the methods for the ActionListeners of the class GUidesign. The method loadFile() defines the functionality of the load file button. It tries to open a file selected by the JFileChooser. It checks the file for a .propertygraph or .history ending and throws an exception otherwise. When the file is read, the filePathTextField and the codeTextField are set according to the loaded file.

The method translateGraph() delegates the file given by the filePathTextField to the translator. Therefore, it uses the method getTranslator() of the class GUiStarter. If the translator cannot verify the file, an exception is thrown. Otherwise, the codeTextField is set to the translated code. The same is done be the method translateUUP(). The two translate methods only differ in the additional parameter they pass to the translator.

7.1.2 Languages Package

The package languages contains an interface for the query language of other databases. This allows to translate the files into multiple languages, if they are implemented. The interface TranslatorInterface provides three methods for the translation. The method processCode is used for the whole translation process. The additional parameter option allows to differentiate the process. The methods translateGraph and translateUUP translate the corresponding files.

The class Neo4j implements the interface TranslatorInterface and provides a translation to a Neo4j Cypher query. Figure 7.5 shows the class diagram of the package languages.

```
<<Interface>>
TranslatorInterface
+processCode(file : File, option : int) : String
+translateGraph(document : Document) : String
+translateUUP(file : List<String>) : String

Ne04j
+processCode(file : File, option : int) : String
+translateGraph(document : Document) : String
+optimizeUUP(commandList : List<String>) : List<String>
+translateUUP(file : List<String>) : String
+setIDRelation(cypherReturn : String, selectePattern : int) : void
+typeCheck(type : String) : boolean
+assembleGraph(matchList : List<String>, whereList : List<String>, returnList : List<String>) : String
+assembleUUP(whereList : List<String>, translatedList : List<String>, newEdgeList : List<String>) : String
```

Figure 7.5. Class diagram of the language package
7.1. The Translator

The Translation Process

In the following, we describe the translation process provided by the class Neo4j. Additionally, the methods and objects are explained. Figure 7.6 shows an overall overview of the functionality.

At first, the translator creates a list of all types included in our SDG. This list is used in the separated translation mechanisms. It prevents the translation of types, which are not included in our SDG. Then, depending on the given integer, one of the translation mechanisms is chosen.

If the integer equals 0, the translator parses the file to an XML document. The mechanism for Candidate Patterns translates the document and returns the resulting Cypher query as a String. If the integer equals 1, the translator puts every single line of the file in a list. The mechanism for Parallelization Patterns translates the list and returns the resulting Cypher query as a String.
7. Architecture of the Translator

Figure 7.7 shows the translation mechanism for Candidate Patterns. At first, the translator checks the root node of the XML document. If it is not a propertygraph, the translation process returns an error string. Otherwise, three lists for each part of the Cypher query are initialized. The first part is the MATCH query, where we define the structure we are looking for. The second part is the WHERE statement, which defines properties of the structure. The last part is the RETURN statement, where we define which elements and properties we want the query to return. After the initialization, the translation mechanism analyzes the nodes. Because of the structure of .propertygraph files, key-value pairs and outgoing edges are children of the nodes. Outgoing edges can have children, too. The translation mechanism creates Cypher strings for all elements and puts them in the corresponding list. Thereby, different keywords require a different translation. For example, a key-value pair with the key runtime is translated differently than a key-value pair with the key forbiddentype. If the key equals type, the translation mechanism uses the typeCheck function to verify if our SDG model includes the given value. Having finished the analysis, the Cypher lists are transformed into a Cypher query.
Figure 7.8 shows the translation mechanism for Parallelization Patterns. At first, the translation mechanism optimizes the list of UIP operations. This merges and removes operations, which increases the translation performance. Then, a relation between the variables in the graph editor and the element IDs of the Neo4j database is created. As mentioned in Section 6.1, this is mandatory for a successful translation. Again, three lists for each part of the Cypher query are initialized. The first part is the WHERE statement, which defines the IDs of the translated elements. The second part contains all translated UIP operations, except for changing types of edges. These operations form the third group. Because of their structure, they are translated last. After the initialization, the translation mechanism analyzes each operation. As seen in Section 6.1, some operations require a different handling, while others are similar to each other. Again, if a key-value pair has the key type, the translation mechanism uses the typeCheck function to verify if our SDG model supports the given value. Having finished the analysis, the Cypher lists are transformed into a Cypher query.
7. Architecture of the Translator

7.2 Integration into the Graph Editor

To integrate the translator into the graph editor, it is changed to a plugin. Therefore, a new package, new classes, and new files are required. Additionally, we modify some existing classes. In the following, we present these changes and describe their functionality.

Figure 7.9. The translator as a plugin to the graph editor

Figure 7.9 shows the translator as a plugin for the graph editor. In the background is Eclipse IDE, which starts the graph editor from source code. It shows the packages, classes and files of the graph editor, especially the translator (1). In the middle layer, the Eclipse runtime application provides a new button in the menu bar (2). It opens the translator (3) shown in the foreground.

The integration includes renaming the existing packages to fit to the layout of the graph editor. Therefore, we add “grapheditor” in the package names as shown in Figure 7.10.

To use the translator as a plugin, we implement the classes TranslatorPlugin.java and ToolbarHandler.java. They are added to the translator package. Additionally, we modify classes in the package de.cau.se.grapheditor.translator.GUI. We remove the main method in the class GUIstarter.java and modify the constructor. This allows to start the translator from the graph editor. In the class GUIfunctions.java, we set the default location to load files to the current runtime workspace. In the class GUIdesign.java, we changed the default close operation of the translator to DISPOSE_ON_CLOSE. Additionally, the translator starts always on top.

To visualize the button in the Eclipse application, we add an icon file to the project (see (4) in Figure 7.9). Additionally, a plugin requires an OSGi bundle manifest. This includes the files plugin.xml, build.properties and a folder META-INF containing a MANIFEST.MF file.
7.2. Integration into the Graph Editor

Figure 7.10. The package diagram of the plugin
To evaluate the translator, we compare its abilities to already existing examples. In this chapter, we evaluate the translation of Candidate Patterns. In Section 8.1, we explain our methodology. Then, we present our Candidate Pattern in Section 8.2. Afterwards, we describe our translation in Section 8.3 and evaluate it in Section 8.4. At last, we present threads to the validity in Section 8.5.

8.1 Approach and Methodology

In her master’s thesis, Johanna Elisabeth Krause describes three different Candidate Patterns. For our evaluation, we pick the pattern with the least complex SDG to maintain an overview. We design the chosen Candidate Pattern in the graph editor and translate it. Then, we compare our translation to the query of J. E. Krause. This includes a test Cypher query on the SDG.

We use Eclipse Modeling Tools Mars.2 Release 4.5.2 x64 with Java SDK 8 Update 102 to design and translate the Candidate Pattern. The graph editor uses KLighD v0.8.0.201602161004. To transform Java source code to an SDG, we use Java2Neo4j and Neo4j v2.3.6 Community Edition. Both applications require the Java SDK 7 Update 80 x64. All applications run on a Windows 10 Pro x64 platform.

8.2 Candidate Pattern for our Evaluation

J. E. Krause describes the reduction of an array as a Candidate Pattern. Listing 8.1 shows her example of an array reduction in Java. We use this pattern for our evaluation. [Krause 2015]

<table>
<thead>
<tr>
<th>Listing 8.1. The reduction of an array implemented in Java (Krause 2015)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 int[] array = {1,2,3,4,5,6,7,8,9,10};</td>
</tr>
<tr>
<td>2 int sum = 0 ;</td>
</tr>
<tr>
<td>3 for (int i=0; i &lt; array.length; i++) {</td>
</tr>
<tr>
<td>4 sum = sum + array[i];</td>
</tr>
<tr>
<td>5 }</td>
</tr>
<tr>
<td>6 System.out.println(&quot;Sum = &quot; + sum);</td>
</tr>
</tbody>
</table>
8. Evaluation of the Candidate Pattern Translation

In Line 1, an array of integers is initialized. It contains the numbers from 1 to 10. In Line 2, the integer sum is initialized with the value 0. In Line 3-5, a for-loop sums up all values of the array and stores the sum in the variable sum. In Line 6, the result is printed on the command line.

Instead of adding one value at a time to sum, we can do multiple operations at the same time. Figure 8.1 shows a possible parallel execution of Listing 8.1. Here, as many as possible additions are executed at the same time. This is only allowed for commutative operations. Therefore, we can create a Candidate Pattern for Listing 8.1.

To create a Cypher query, we require a graph of the Candidate Pattern. Therefore, we create a Java application with a main method, which contains Listing 8.1. Then, we use the Java2Neo4j to transform the source code to an SDG. Figure 8.2 shows the resulting SDG.

Figure 8.1. A possible parallel execution of Listing 8.1

Figure 8.2. The system dependency graph representing a main method with Listing 8.1
8.2. *Candidate Pattern* for our Evaluation

In the *Neo4j* graph editor, circles represent nodes. In Figure 8.2, nodes of the type `Package` are colored red. `Class` nodes are colored yellow and `Constructor` nodes are colored pink. `Method` nodes are colored blue while `Field` nodes are colored green. The upper red node represents the package containing our code. The other used packages are `java.lang` and `java.io`. The upper right yellow node represents the class containing the main method. The classes left to it are `Object` and `System`. `System` contains the `Field` node `out`. The two `Class` nodes in the middle are `StringBuffer` and `PrintStream`. The upper blue node represents the `main` method. The other blues nodes represent the method declarations `println`, `toString` and `append`. The upper circle of grey nodes represents the initialization of the array. The straight line right to it represents the initialization and the condition of the for-loop. The grey circle on the bottom right represents the body of the for loop.

We rebuild the required parts of the SDG in the graph editor to create a *Candidate Pattern*. This includes the first node before the for loop and the first node after it. Figure 8.3 shows the *Candidate Pattern* in the graph editor. It has 21 nodes and 40 edges. We defined a total amount of 93 properties. The automatic layout of the graph editor causes the *Candidate Pattern* to be stretched in the wide. To improve the readability of Figure 8.3, we present its XML serialization in Listing A.1.

![Figure 8.3. Our designed Candidate Pattern in the graph editor](image)

Listing 8.2 shows the *Cypher* query created by J. E. Krause. In the Lines 1-14, the `MATCH` statement is defined. The names of the variables for nodes reflect their purpose. The types of the nodes are defined by their label. Additionally, the property `operation` is defined for some nodes. The Lines 1-9 represent one path in the graph, which consists of 8 nodes. In the Lines 15-27, the `WHERE` statement is defined. It predetermines multiple conditions for properties of the element. In this case, it focuses on an array of numbers. At last, in Line 28, the `RETURN` statement returns one of the nodes. In the following, we will refer to the *Cypher* query of J. E. Krause as *target query*. 

73
8. Evaluation of the Candidate Pattern Translation

<table>
<thead>
<tr>
<th>Listing 8.2. A Cypher query to find structures like Listing 8.1 in a SDG (Krause 2015)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATCH (initial: Assignment{operation: 'value'}) -[:CONTROL_FLOW]-&gt;</td>
</tr>
<tr>
<td>(itstart: Assignment{operation: 'value'}) -[:CONTROL_FLOW]-&gt;</td>
</tr>
<tr>
<td>(length: Assignment{operation: 'length'}) -[:CONTROL_FLOW]-&gt;</td>
</tr>
<tr>
<td>(if: Condition) -[:CONTROL_FLOW]-&gt;</td>
</tr>
<tr>
<td>(assign: Assignment{operation: 'arrayaccess'}) -[:CONTROL_FLOW]-&gt;</td>
</tr>
<tr>
<td>(rd: Assignment) -[:DATA_FLOW]-&gt; (assign) &lt;-[d1:DATA_FLOW]- (array) -[d2:DATA_FLOW]-&gt; (length),</td>
</tr>
<tr>
<td>(itstart) -[:DATA_FLOW]-&gt; (itplus) -[:DATA_FLOW]-&gt; (if),</td>
</tr>
<tr>
<td>(assign) -[:DATA_FLOW]-&gt; (assign),</td>
</tr>
<tr>
<td>(itplus) -[:DATA_FLOW]-&gt; (itplus)</td>
</tr>
<tr>
<td>WHERE</td>
</tr>
<tr>
<td>itstart.var = itplus.var</td>
</tr>
<tr>
<td>AND if.name = itstart.var + '&lt;' + length.var</td>
</tr>
<tr>
<td>AND (rd.operation = '+'</td>
</tr>
<tr>
<td>OR rd.operation = '-'</td>
</tr>
<tr>
<td>OR rd.operation = '*' )</td>
</tr>
<tr>
<td>AND assign.var = d5.var</td>
</tr>
<tr>
<td>AND (array.vartype = 'int[]'</td>
</tr>
<tr>
<td>OR array.vartype = 'long[]'</td>
</tr>
<tr>
<td>OR array.vartype = 'double[]' )</td>
</tr>
<tr>
<td>AND (array.var = d1.var = d2.var</td>
</tr>
<tr>
<td>OR array.name = d1.var = d2.var)</td>
</tr>
<tr>
<td>AND initial.var = rd.var</td>
</tr>
<tr>
<td>RETURN length</td>
</tr>
</tbody>
</table>

8.3 Results and Discussion

The translation produced the Cypher query shown in Listing A.2. Listing 8.3 shows a formatted version of the Cypher query. It contains the same code, but we deleted some line breaks and created longer paths. This allows an easier comparison of both Candidate Patterns. The MATCH statement defines the paths and the type of the edges. The WHERE statement verifies the types of the nodes and the properties of nodes and edges. The RETURN statement defines the return values. They comprise the object and its ID.
### Listing 8.3. Formatted Cypher query for the Candidate Pattern of the evaluation

```
MATCH (node_1)-[edge_1:DATA_FLOW]->(node_2)-[edge_9:DATA_FLOW]->(node_8),
(node_1)-[edge_2:CONTROL_FLOW]->(node_3)-[edge_31:DATA_FLOW]->(node_18),
(node_1)-[edge_18:DATA_FLOW]->(node_13)-[edge_19:CONTROL_FLOW]->(node_14),
(node_2)-[edge_10:CONTROL_FLOW]->(node_8)-[edge_14:CONTROL_FLOW]->(node_10),
(node_3)-[edge_4:CONTROL_FLOW]->(node_5)-[edge_5:CONTROL_FLOW]->(node_6),
(node_5)-[edge_6:CONTAINS_UNIT]->(node_6)-[edge_11:DATA_FLOW]->(node_8),
(node_6)-[edge_7:CONTROL_FLOW]->(node_7)-[edge_8:CONTROL_FLOW]->(node_2),
(node_6)-[edge_20:DATA_FLOW]->(node_14)-[edge_21:DATA_FLOW]->(node_15),
(node_6)-[edge_34:DATA_FLOW]->(node_20)-[edge_35:DATA_FLOW]->(node_21),
(node_8)-[edge_15:CONTROL_FLOW]->(node_11)-[edge_16:CONTROL_FLOW]->(node_12),
(node_12)-[edge_17:CONTROL_FLOW]->(node_13)-[edge_23:DATA_FLOW]->(node_15),
(node_12)-[edge_44:DATA_FLOW]->(node_16)-[edge_27:DATA_FLOW]->(node_17),
(node_14)-[edge_22:CONTROL_FLOW]->(node_15)-[edge_24:DATA_FLOW]->(node_16),
(node_15)-[edge_25:CONTROL_FLOW]->(node_16)-[edge_28:CONTROL_FLOW]->(node_17),
(node_17)-[edge_32:CONTROL_FLOW]->(node_19)-[edge_33:CONTROL_FLOW]->(node_20),
(node_20)-[edge_36:CONTROL_FLOW]->(node_21)-[edge_37:DATA_FLOW]->(node_22),
(node_21)-[edge_38:CONTROL_FLOW]->(node_22)-[edge_41:DATA_FLOW]->(node_8),
(node_18)-[edge_43:CONTROL_FLOW]->(node_23),
(node_18)-[edge_29:DATA_FLOW]->(node_12),
(node_17)-[edge_30:DATA_FLOW]->(node_18),
(node_22)-[edge_39:DATA_FLOW]->(node_20),
(node_22)-[edge_40:DATA_FLOW]->(node_14),
(node_22)-[edge_42:CONTROL_FLOW]->(node_7)
WHERE node_1.type="Assignment" AND node_1.operation="value"
AND node_1.vartype="int[]" AND node_2.type="Assignment"
AND node_3.type="Assignment" AND node_13.type="Assignment"
AND node_2.operation="length" AND node_2.vartype="int"
AND node_8.type="Condition" AND node_3.vartype="int"
AND node_5.type="NopStmt" AND node_18.type="MethodCallWithReturnValue"
AND node_5.nopkind="FOR_INIT" AND node_6.type="Assignment"
AND node_6.vartype="int" AND node_6.operation="value"
AND node_7.type="NopStmt" AND node_14.type="Assignment"
AND node_8.type="NopStmt" AND node_11.type="NopStmt"
AND node_10.type="NopStmt" AND node_23.type="Assignment"
AND node_11.type="NopStmt" AND node_12.type="Assignment"
AND node_12.operation="value" AND node_12.vartype="int"
AND node_16.type="Assignment" AND node_13.operation="value"
AND node_13.vartype="int[]" AND node_15.type="Assignment"
AND node_14.operation="value" AND node_14.vartype="int"
```

75
8. Evaluation of the Candidate Pattern Translation

41  AND node_15.vartype="int" AND node_15.operation="arrayaccess"
42  AND node_16.vartype="int" AND node_17.type="Assignment"
43  AND node_17.vartype="int" AND node_17.operation="value"
44  AND node_19.type="NopStmt" AND node_18.fqn="java.lang.StringBuffer.append(int)"
45  AND node_19.nopkind="FOR_UPDATE" AND node_20.operation="value"
46  AND node_20.vartype="int" AND node_22.type="Assignment"
47  AND node_22.operation="value" AND node_22.vartype="int"
48  AND node_23.vartype="java.io.PrintStream"
49  AND node_23.operation="staticFieldAccess"
50  RETURN node_1.ID(node_1),edge_1.ID(edge_1),edge_2.ID(edge_2),
51  edge_18.ID(edge_18),node_2.ID(node_2),edge_9.ID(edge_9),edge_10.ID(edge_10),
52  node_3.ID(node_3),edge_4.ID(edge_4),edge_31.ID(edge_31),node_5.ID(node_5),
53  edge_5.ID(edge_5),edge_6.ID(edge_6),node_6.ID(node_6),edge_7.ID(edge_7),
54  edge_11.ID(edge_11),edge_20.ID(edge_20),edge_34.ID(edge_34),node_7.ID(node_7),
55  edge_8.ID(edge_8),node_8.ID(node_8),edge_14.ID(edge_14),edge_15.ID(edge_15),
56  node_10.ID(node_10),edge_43.ID(edge_43),node_11.ID(node_11),edge_16,
57  ID(edge_16),node_12.ID(node_12),edge_17.ID(edge_17),edge_44.ID(edge_44),
58  node_13.ID(node_13),edge_19.ID(edge_19),edge_23.ID(edge_23),node_14,
59  ID(edge_14),edge_21.ID(edge_21),edge_22.ID(edge_22),node_15.ID(node_15),
60  edge_24.ID(edge_24),edge_25.ID(edge_25),node_16.ID(node_16),edge_27,
61  ID(edge_27),edge_28.ID(edge_28),node_17.ID(node_17),edge_29.ID(edge_29),
62  edge_30.ID(edge_30),edge_32.ID(edge_32),node_18.ID(node_18),node_19,
63  ID(node_19),edge_33.ID(edge_33),node_20.ID(node_20),edge_35.ID(edge_35),
64  edge_36.ID(edge_36),node_21.ID(node_21),edge_37.ID(edge_37),edge_38,
65  ID(edge_38),node_22.ID(node_22),edge_39.ID(edge_39),edge_40.ID(edge_40),
66  edge_41.ID(edge_41),edge_42.ID(edge_42),node_23.ID(node_23)

8.4 Comparison of the Cypher Queries

The target query uses meaningful variable names for nodes and edges. If not necessary, edges do not have a name. The code is arranged in a way humans can understand it. Long paths use line breaks to maintain readability. The target query uses tabs to organize and group statements. The MATCH statement defines the type of nodes by labels. Also, the operation property is defined. The AND operator lists properties in the WHERE statement. Optional properties are defined with the OR operator. The RETURN statement consists of the last node in the pattern.

Our query uses the variable names given by the graph editor. Every node and edge has a name. Paths are not longer than one edge. Each statement creates a new line. The
AND operator lists all properties in the WHERE statement. This includes the types of all nodes. The RETURN statement consists of every node and every edge in the pattern. Additionally, it returns their IDs.

The automated generation of a Cypher query requires additional input to name the variables in a meaningful way. The mechanism cannot detect, whether a variable is used later on or not. Therefore, every node and every edge is represented by a variable. The structure of a .propertygraph file causes the paths to be not longer than one edge. Additionally, properties appear after the definition of a node or an edge. Therefore, an automated translation uses the WHERE statement for all properties. Java2Neo4j uses the property type to define the type of a node. Therefore, a label is not required. The OR operator is not yet implemented in the graph editor and can not be translated. To create the query of the User Interaction Protocol, we require the name of every variable and the corresponding ID in the database. Therefore, our RETURN statement contains these values. The readability of the code always depends on the aspect ratio and the size of the screen. However, the purpose of the translation is a query on a database. Therefore, the readability is negligible.

At last, our Cypher query returns a possible match for the Candidate Pattern. The target query matches no structure in our SDG. This may have multiple reasons. First of all, the model of our SDG is in continuous development. Some types may not have existed at the time, the target query was created. Additionally, some properties may be renamed or deleted. It may contain not existing or deprecated operations. For example, properties containing only a mathematical symbol are not allowed.

8.5 Threads to Validity

Internal Validity

The resulting query depends on the modeling in the editor. There may exist other ways to model our Candidate Pattern. Therefore, other queries may find the same results.

External Validity

In our evaluation, we considered the example of an array reduction. This example does not use the full set of functionality the graph offers. Unusual combinations, not foreseen structures, or a wrong handling may cause the translator to fail.
Chapter 9

Evaluation of the Parallelization Pattern Translation

In this chapter, we evaluate the translation of Parallelization Patterns. At first, we explain our methodology in Section 9.1. In Section 9.2, we present the corresponding Parallelization Pattern to Listing 8.1. Afterwards, we describe and evaluate results in Section 9.3. At last, we present threads to the validity in Section 9.4.

9.1 Approach and Methodology

For our evaluation, we exchange the for-loop in Listing 8.1 with a construct for parallel execution. Then, we transform the code to an SDG. We use the graph editor to transform our Candidate Pattern to match the structure of the parallel source code in the SDG. This creates a Parallelization Pattern. Then, we translate the Parallelization Pattern. To evaluate the result, we apply the resulting Cypher query on the SDG created in Section 8.2. At last, we compare the two SDGs.

The graph editor uses the same setup as in Section 8.1. We cannot use Java2Neo4j for the code transformation. The source code bases on Java 8, which is not supported by Java2Neo4j. To modify the SDG, we use Neo4j v3.0.4 Community Edition. All applications run on a Windows 10 Pro x64 platform.

9.2 System Dependency Graph for our Evaluation

Listing 9.1 shows the parallel source code for Listing 8.1. In Line 1, an array of integers is initialized. It contains the numbers from 1 to 10. In Line 2, the integer sum is initialized with the value of the sum of the array. This construct for parallel execution was introduced in Java 8. In Line 3, the result is printed on the command line.

```
Listing 9.1. The parallel source code for Listing 8.1 (Krause 2015)
1 int[] array = {1,2,3,4,5,6,7,8,9,10};
2 int sum = Arrays.stream(array).parallel().sum();
3 System.out.println("Sum = " + sum);
```
9. Evaluation of the Parallelization Pattern Translation

To transform the source code to an SDG without Java2Neo4j, we use the SDG of Section 8.2. It provides a foundation for our desired changes. Then, we replace the for-loop with a parallel construct. Therefore, we use the Java API to get additional information about the class Arrays and the methods stream(int[]), parallel(), and sum().

Figure 9.1. The assumed System Dependency Graph including Listing 9.1

Figure 9.1 shows our assumed SDG. For an improved distinction, nodes have a different size. Comparing it to Figure 8.2, the left half remains the same. On the right, the for-loop is removed. It is replaced by three methods from two different classes and packages.

We modify the Candidate Pattern shown in Figure 8.3 to create a Parallelization Pattern. Figure 9.2 shows the Parallelization Pattern in the graph editor. The automatic layout of the graph editor causes the Parallelization Pattern to be stretched in the wide. To improve the readability of Figure 9.2, we present its XML serialization in Listing A.3. Listing A.4 shows the resulting UIP after the transformation.
9.3 Results and Discussion

The translation produced the Cypher query shown in Listing A.5. At first, the **MATCH** statement rebuilds the structure of the **Candidate Pattern**. Then, the **WHERE** statement creates a relation between the names of nodes and edges and their ID in the graph database. The single statements are concatenated with the operator **AND**. Then, the **DELETE** and **DETACH** operators remove no longer required nodes and edges. At last, the **CREATE** operator creates new nodes and edges. The **SET** operator creates properties of nodes and edges.

![Figure 9.2. Our designed Parallelization Pattern in the graph editor](image)

![Figure 9.3. The system dependency graph after executing Listing A.5](image)
9. Evaluation of the Parallelization Pattern Translation

We apply the Cypher query shown in Listing A.5 to the SDG shown in Section 8.2. This results in the SDG shown in Figure 9.3. The left half of the graph remains unchanged. On the right, the for-loop is removed. It is replaced by the 10 unnamed grey nodes. They represent the methods `texttstream(int[]), parallel(), and sum()`. The other nodes represent the classes `Arrays` and `IntStream` and the packages `java.util` and `java.util.stream`. The data flow and the control flow run from the last node of the array declaration to three nodes of the type `MethodCallWithReturnValue`. Each of them calls the method they represent. These methods are contained in classes, which are contained in packages.

The structure of Figure 9.1 and Figure 9.3 is the same. While the replaced nodes in Figure 9.1 are colored and have a different size, all new nodes in Figure 9.1 are grey and have no label. Moreover, all nodes and all edges have the same type and the same properties.

The visual difference is caused by missing labels. If we add a label to a node, it is colored and labeled.

9.4 Threads to Validity

Internal Validity

So far, Java2Neo4j does not support Java 8. A future translation by Java2Neo4j may result in a different SDG. Furthermore, our SDG is only an estimation. A valid SDG for our scenario may look different.

External Validity

In our evaluation, we considered the example of an array reduction. This example does not use the full set of functionality the graph offers. Unusual combinations, not foreseen structures, or a wrong handling may cause the translator to fail.
Chapter 10

Related Work

This chapter presents scientific papers or other theses, which topics intersect or are similar to this master thesis.

10.1 Pattern-Based Detection in Software Systems

C. Wulf presents the idea of using graph patterns to parallelize source code. Instead of matching program structures and restructuring the source code, a semi-automatic matching on a system dependency graph is proposed. This paper is the foundation for our approach. [Wulf 2014]

J. E. Krause describes the translation of three prototypes of Candidate Patterns. They are used in a first approach of pattern matching. In contrast to our approach, the patterns are created and translated manually. [Krause 2015]

AutoFuture is a tool to automatically identify sections in sequential code, which can be parallelized. Then, the section is reengineered to parallel code. In contrast to our approach, AutoFuture is a fully automated tool. Therefore, it has the disadvantages we mentioned in Section 1.1. [Molitorisz et al. 2012]

10.2 Graph Editors

L. E. Blümke presents the development of an Eclipse-based graph editor with focus on system dependency graphs. The implemented layers and types base on the “Java System Dependency Graph” presented in Section 2.1.4. We improved this graph editor by adding features and fixing design flaws. These changes are shown in Chapter 4. [Blümke 2015]

Y. Benekov describes the development of an Eclipse plugin to detect changes to a graph in the graph editor. These changes are recorded and stored in a file. This allows the construction of a new graph. We improve this mechanism and present further design flaws. [Benekov 2015]
10. Related Work

D. Finkes adds hierarchies to elements in the graph editor. Additionally, import and export of Neo4j graph databases is introduced. In contrast to our approach, this thesis focuses on system dependency graphs and not on Candidate Patterns and Parallelization Patterns. [Finkes 2016]

The Neo4j graph editor is a browser-based application. By running in a web browser, the graph editor can use only one thread. Additionally, the editor only supports a single layout algorithm which arranges nodes in star clusters. The user has to layout the graph by hand. Each new query to the graph database resets the layout. [Neo Technology 2007]
Chapter 11

Conclusions and Future Work

11.1 Conclusions

Our explorative approach created an inductive mechanism to translate our Candidate Patterns. We revealed all currently available structures and their counterparts in a Neo4j Cypher query. Additionally, we created a translator for our Parallelization Patterns. We revealed the problems of the UIP and suggest improvements.

The translator successfully translates Candidate Patterns and Parallelization Patterns. The optimization of the User Interaction Protocol removes lines from the .history file, without changing the outcome of the query. Nonetheless, the translation mechanism only supports a limited set of operations. These operations represent the current functionality of the graph editor. Therefore, both can be enriched by further features.

11.2 Future Work

The current graph editor is a composition of an Eclipse IDE, imported libraries and plugins to add features. Therefore, it lacks in performance and possibilities in customization. Additionally, these external resources are updated regularly, which leads to incompatibility issues. Creating a standalone graph editor and relinquishing on external libraries would probably fix these issues.

The graph editor could also be expanded by additional features. For example, starting the editor for the first time results in missing views. They have to be added and organized by hand. This could be done automatically at launch. Additionally, the graph layout is predefined by the KLighD library. Allowing the user to move nodes and edges could increase the quality of visualization of the graphs. Furthermore, the graph editor has to be operated with the mouse. Implementing keyboard commands, for example removing selected nodes with the delete key, could increase the user experience.

The improvements to the graph editor in this thesis revealed further aspects, which could be enhanced. The node NopStmt requires further properties. A multiple choice menu at creation simplifies the process. The implemented forbidden edge can restrict a relation completely or for a specific type. Forbidding two or more types has to be designed with a respectively amount of edges. Therefore, forbidden edges with multiple types simplify the design process. Additionally, coloring nodes or edges with a key-value-color triplet
11. Conclusions and Future Work

overwrites the color property of the element. Deleting the key-value-color triplet afterwards does not reset the color. Therefore, a storage for the colors of nodes and edges fixes this issue.

Also, we noticed a different handling of specific key-value pairs in the translation of the User Interaction Protocol. To improve the translator, new UIP operations of the form `<ELEM>_TYPE_<OPERATION>` could be introduced. An early distinction improves the translation mechanism. Additionally, setting a type for nodes could also change the label of the node. This improves the visualization and allows a better distinction. At last, our changes introduced some pop-up windows. Working on a multi monitor setup can cause these windows to open on a different screen than the graph editor. Displaying the windows on the same screen improves the user experience.
Appendix

A.1 Candidate Pattern

A.1.1 XML Serialization

<table>
<thead>
<tr>
<th>Listing A.1. The XML serialization representing our Candidate Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>17</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>19</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>21</td>
</tr>
<tr>
<td>22</td>
</tr>
<tr>
<td>23</td>
</tr>
</tbody>
</table>
A. Appendix

```
<nodes label="nop" color="FFFFFF" ID="node_11" incomingEdges="edge_15">
    <keyValuePairs key="type" value="NopStmt"/>
    <keyValuePairs key="nopkind" value="FOR BODY"/>
    <outgoingEdges label="CONTROL_FLOW" color="000000" ID="edge_16" targetNode="node_12">
        <keyValuePairs key="type" value="CONTROL_FLOW"/>
    </outgoingEdges>
</nodes>

<nodes label="temp2=sum" color="FFFFFF" ID="node_12" incomingEdges="edge_16 edge_29">
    <keyValuePairs key="type" value="Assignment"/>
    <keyValuePairs key="operation" value="value"/>
    <keyValuePairs key="vartype" value="int"/>
    <outgoingEdges label="CONTROL_FLOW" color="000000" ID="edge_17" targetNode="node_13">
        <keyValuePairs key="type" value="CONTROL_FLOW"/>
    </outgoingEdges>
    <outgoingEdges label="DATA_FLOW" color="000000" ID="edge_44" targetNode="node_16">
        <keyValuePairs key="type" value="DATA_FLOW"/>
    </outgoingEdges>
</nodes>

<nodes label="temp3=array" color="FFFFFF" ID="node_13" incomingEdges="edge_17 edge_18">
    <keyValuePairs key="type" value="Assignment"/>
    <keyValuePairs key="operation" value="value"/>
    <keyValuePairs key="vartype" value="int[]"/>
    <outgoingEdges label="CONTROL_FLOW" color="000000" ID="edge_19" targetNode="node_14">
        <keyValuePairs key="type" value="CONTROL_FLOW"/>
    </outgoingEdges>
    <outgoingEdges label="DATAFLOW" color="000000" ID="edge_23" targetNode="node_15">
        <keyValuePairs key="type" value="DATA_FLOW"/>
    </outgoingEdges>
</nodes>

<nodes label="temp4=i" color="FFFFFF" ID="node_14" incomingEdges="edge_19 edge_20 edge_40">
    <keyValuePairs key="type" value="Assignment"/>
    <keyValuePairs key="operation" value="value"/>
    <keyValuePairs key="vartype" value="int"/>
</nodes>
```
A.1. Candidate Pattern

```xml
<outgoingEdges label="DATA_FLOW" color="000000" ID="edge_21"
targetNode="node_15">
   <keyValuePairs key="type" value="DATA_FLOW"/>
</outgoingEdges>
<outgoingEdges label="CONTROL_FLOW" color="000000" ID="edge_22"
targetNode="node_15">
   <keyValuePairs key="type" value="CONTROL_FLOW"/>
</outgoingEdges>
</nodes>
<nodes label="temp5=temp3[temp4]" color="FFFFFF" ID="node_15"
incomingEdges="edge_21 edge_22 edge_23">
   <keyValuePairs key="type" value="Assignment"/>
   <keyValuePairs key="vartype" value="int"/>
   <keyValuePairs key="operation" value="arrayaccess"/>
   <outgoingEdges label="DATA_FLOW" color="000000" ID="edge_24"
targetNode="node_16">
      <keyValuePairs key="type" value="DATA_FLOW"/>
   </outgoingEdges>
   <outgoingEdges label="CONTROL_FLOW" color="000000" ID="edge_25"
targetNode="node_16">
      <keyValuePairs key="type" value="CONTROL_FLOW"/>
   </outgoingEdges>
</nodes>
<nodes label="temp6=temp2+temp5" color="FFFFFF" ID="node_16"
incomingEdges="edge_24 edge_25 edge_44">
   <keyValuePairs key="type" value="Assignment"/>
   <keyValuePairs key="vartype" value="int"/>
   <outgoingEdges label="DATA_FLOW" color="000000" ID="edge_27"
targetNode="node_17">
      <keyValuePairs key="type" value="DATA_FLOW"/>
   </outgoingEdges>
   <outgoingEdges label="CONTROL_FLOW" color="000000" ID="edge_28"
targetNode="node_17">
      <keyValuePairs key="type" value="CONTROL_FLOW"/>
   </outgoingEdges>
</nodes>
<nodes label="sum=temp6" color="FFFFFF" ID="node_17"
incomingEdges="edge_27 edge_28">
   <keyValuePairs key="type" value="Assignment"/>
   <keyValuePairs key="vartype" value="int"/>
   <keyValuePairs key="operation" value="value"/>
</nodes>
```
A. Appendix

```xml
<outgoingEdges label="DATA_FLOW" color="000000" ID="edge_29"
  targetNode="node_12">
  <keyValuePairs key="type" value="DATA_FLOW"/>
</outgoingEdges>

<outgoingEdges label="DATA_FLOW" color="000000" ID="edge_30"
  targetNode="node_18">
  <keyValuePairs key="type" value="DATA_FLOW"/>
</outgoingEdges>

<outgoingEdges label="CONTROL_FLOW" color="000000" ID="edge_32"
  targetNode="node_19">
  <keyValuePairs key="type" value="CONTROL_FLOW"/>
</outgoingEdges>

<nodes label="append" color="FFFFFF" ID="node_18"
  incomingEdges="edge_30 edge_31">
  <keyValuePairs key="type" value="MethodCallWithReturnValue"/>
  <keyValuePairs key="fqn" value="java.lang.StringBuffer.append(int)"/>
  <keyValuePairs key="returntype" value="java.lang.StringBuffer"/>
</nodes>

<nodes label="nop" color="FFFFFF" ID="node_19" incomingEdges="edge_32">
  <keyValuePairs key="type" value="NopStmt"/>
  <keyValuePairs key="nopkind" value="FOR_UPDATE"/>
  <outgoingEdges label="CONTROL_FLOW" color="000000" ID="edge_33"
    targetNode="node_20">
    <keyValuePairs key="type" value="CONTROL_FLOW"/>
  </outgoingEdges>
</nodes>

<nodes label="temp7=i" color="FFFFFF" ID="node_20"
  incomingEdges="edge_33 edge_34 edge_39">
  <keyValuePairs key="type" value="Assignment"/>
  <keyValuePairs key="operation" value="value"/>
  <keyValuePairs key="vartype" value="int"/>
  <outgoingEdges label="DATA_FLOW" color="000000" ID="edge_35"
    targetNode="node_21">
    <keyValuePairs key="type" value="DATA_FLOW"/>
  </outgoingEdges>
  <outgoingEdges label="CONTROL_FLOW" color="000000" ID="edge_36"
    targetNode="node_21">
    <keyValuePairs key="type" value="CONTROL_FLOW"/>
  </outgoingEdges>
</nodes>
```
A.1. Candidate Pattern

```xml
<nodes label="temp8=temp7+1" color="FFFFFF" ID="node_21"
incomingEdges="edge_35 edge_36">
  <keyValuePairs key="type" value="Assignment"/>
  <keyValuePairs key="vartype" value="int"/>
  <outgoingEdges label="DATA_FLOW" color="000000" ID="edge_37"
targetNode="node_22">
    <keyValuePairs key="type" value="DATA_FLOW"/>
  </outgoingEdges>
  <outgoingEdges label="CONTROL_FLOW" color="000000" ID="edge_38"
targetNode="node_22">
    <keyValuePairs key="type" value="CONTROL_FLOW"/>
  </outgoingEdges>
</nodes>

<nodes label="i=temp8" color="FFFFFF" ID="node_22"
incomingEdges="edge_37 edge_38">
  <keyValuePairs key="type" value="Assignment"/>
  <keyValuePairs key="operation" value="value"/>
  <keyValuePairs key="vartype" value="int"/>
  <outgoingEdges label="DATA_FLOW" color="000000" ID="edge_39"
targetNode="node_20">
    <keyValuePairs key="type" value="DATA_FLOW"/>
  </outgoingEdges>
  <outgoingEdges label="DATA_FLOW" color="000000" ID="edge_40"
targetNode="node_14">
    <keyValuePairs key="type" value="DATA_FLOW"/>
  </outgoingEdges>
  <outgoingEdges label="DATA_FLOW" color="000000" ID="edge_41"
targetNode="node_8">
    <keyValuePairs key="type" value="DATA_FLOW"/>
  </outgoingEdges>
  <outgoingEdges label="CONTROL_FLOW" color="000000" ID="edge_42"
targetNode="node_7">
    <keyValuePairs key="type" value="CONTROL_FLOW"/>
  </outgoingEdges>
</nodes>

<nodes label="temp9=\&lt;java.lang.System\" color="FFFFFF" ID="node_23"
incomingEdges="edge_43">
  <keyValuePairs key="type" value="Assignment"/>
  <keyValuePairs key="vartype" value="java.io.PrintStream"/>
  <keyValuePairs key="operation" value="staticFieldAccess"/>
</nodes>
```
A. Appendix

Listing A.2. Cypher query for the Candidate Pattern of the evaluation

```cypher
MATCH (node_1)-[edge_1:DATA_FLOW]->(node_2),
    (node_1)-[edge_2:CONTROL_FLOW]->(node_3),
    (node_1)-[edge_18:DATA_FLOW]->(node_13),
    (node_2)-[edge_9:DATA_FLOW]->(node_8),
    (node_2)-[edge_10:CONTROL_FLOW]->(node_8),
    (node_3)-[edge_4:CONTROL_FLOW]->(node_5),
    (node_3)-[edge_31:DATA_FLOW]->(node_18),
    (node_5)-[edge_5:CONTROL_FLOW]->(node_6),
    (node_5)-[edge_6:CONTAINS_UNIT]->(node_6),
    (node_6)-[edge_7:CONTROL_FLOW]->(node_7),
    (node_6)-[edge_11:DATA_FLOW]->(node_8),
    (node_6)-[edge_20:DATA_FLOW]->(node_14),
    (node_6)-[edge_34:DATA_FLOW]->(node_20),
    (node_7)-[edge_8:CONTROL_FLOW]->(node_2),
    (node_8)-[edge_14:CONTROL_FLOW]->(node_10),
    (node_8)-[edge_15:CONTROL_FLOW]->(node_11),
    (node_10)-[edge_43:CONTROL_FLOW]->(node_23),
    (node_11)-[edge_16:CONTROL_FLOW]->(node_12),
    (node_12)-[edge_17:CONTROL_FLOW]->(node_13),
    (node_12)-[edge_44:DATA_FLOW]->(node_16),
    (node_13)-[edge_19:CONTROL_FLOW]->(node_14),
    (node_13)-[edge_23:DATA_FLOW]->(node_15),
    (node_14)-[edge_21:DATA_FLOW]->(node_15),
    (node_14)-[edge_22:CONTROL_FLOW]->(node_15),
    (node_15)-[edge_24:DATA_FLOW]->(node_16),
    (node_15)-[edge_25:CONTROL_FLOW]->(node_16),
    (node_16)-[edge_27:DATA_FLOW]->(node_17),
    (node_16)-[edge_28:CONTROL_FLOW]->(node_17),
    (node_17)-[edge_29:DATA_FLOW]->(node_12),
    (node_17)-[edge_30:DATA_FLOW]->(node_18),
    (node_17)-[edge_32:CONTROL_FLOW]->(node_19),
    (node_19)-[edge_33:CONTROL_FLOW]->(node_20),
    (node_20)-[edge_35:DATA_FLOW]->(node_21),
```
A.1. Candidate Pattern

(node_20) - [edge_36:CONTROL_FLOW] -> (node_21),
(node_21) - [edge_37:DATA_FLOW] -> (node_22),
(node_21) - [edge_38:CONTROL_FLOW] -> (node_22),
(node_22) - [edge_39:DATA_FLOW] -> (node_20),
(node_22) - [edge_40:DATA_FLOW] -> (node_14),
(node_22) - [edge_41:DATA_FLOW] -> (node_8),
(node_22) - [edge_42:CONTROL_FLOW] -> (node_7)
WHERE node_1.type = "Assignment"
AND node_1.operation = "value"
AND node_1.vartype = "int[]"
AND node_2.type = "Assignment"
AND node_3.type = "Assignment"
AND node_13.type = "Assignment"
AND node_2.operation = "length"
AND node_2.vartype = "int"
AND node_8.type = "Condition"
AND node_3.vartype = "int"
AND node_5.type = "NopStmt"
AND node_18.type = "MethodCallWithReturnValue"
AND node_5.nopkind = "FOR_INIT"
AND node_6.type = "Assignment"
AND node_6.vartype = "int"
AND node_6.operation = "value"
AND node_7.type = "NopStmt"
AND node_14.type = "Assignment"
AND node_20.type = "Assignment"
AND node_7.nopkind = "FOR_COND"
AND node_10.type = "NopStmt"
AND node_11.type = "NopStmt"
AND node_10.nopkind = "FOR_END"
AND node_23.type = "Assignment"
AND node_11.nopkind = "FOR_BODY"
AND node_12.type = "Assignment"
AND node_12.operation = "value"
AND node_12.vartype = "int"
AND node_16.type = "Assignment"
AND node_13.operation = "value"
AND node_13.vartype = "int[]"
AND node_15.type = "Assignment"
AND node_14.operation = "value"
AND node_14.vartype = "int"
A. Appendix

75 AND node_15.vartype="int"
76 AND node_15.operation="arrayaccess"
77 AND node_16.vartype="int"
78 AND node_17.type="Assignment"
79 AND node_17.vartype="int"
80 AND node_17.operation="value"
81 AND node_19.type="NopStmt"
82 AND node_18.fqn="java.lang.StringBuffer.append(int)"
83 AND node_18.returntype="java.lang.StringBuffer"
84 AND node_19.nopkind="FOR_UPDATE"
85 AND node_20.operation="value"
86 AND node_20.vartype="int"
87 AND node_21.type="Assignment"
88 AND node_21.vartype="int"
89 AND node_22.type="Assignment"
90 AND node_22.operation="value"
91 AND node_22.vartype="int"
92 AND node_23.vartype="java.io.PrintStream"
93 AND node_23.operation="staticFieldAccess"
94 RETURN node_1,ID(node_1),
95 edge_1,ID(edge_1),
96 edge_2,ID(edge_2),
97 edge_18,ID(edge_18),
98 node_2,ID(node_2),
99 edge_9,ID(edge_9),
100 edge_11,ID(edge_11),
101 node_3,ID(node_3),
102 edge_4,ID(edge_4),
103 edge_31,ID(edge_31),
104 node_5,ID(node_5),
105 edge_5,ID(edge_5),
106 edge_6,ID(edge_6),
107 node_6,ID(node_6),
108 edge_7,ID(edge_7),
109 edge_11,ID(edge_11),
110 edge_20,ID(edge_20),
111 edge_34,ID(edge_34),
112 node_7,ID(node_7),
113 edge_8,ID(edge_8),
114 node_8,ID(node_8),
115 edge_14,ID(edge_14),
116
A.1. Candidate Pattern

edge_15, ID(edge_15),
node_10, ID(node_10),
edge_43, ID(edge_43),
node_11, ID(node_11),
edge_16, ID(edge_16),
node_12, ID(node_12),
edge_17, ID(edge_17),
edge_44, ID(edge_44),
node_13, ID(node_13),
edge_19, ID(edge_19),
edge_23, ID(edge_23),
node_14, ID(node_14),
edge_21, ID(edge_21),
edge_22, ID(edge_22),
node_15, ID(node_15),
edge_24, ID(edge_24),
edge_25, ID(edge_25),
node_16, ID(node_16),
edge_27, ID(edge_27),
edge_28, ID(edge_28),
node_17, ID(node_17),
edge_29, ID(edge_29),
edge_30, ID(edge_30),
edge_32, ID(edge_32),
node_18, ID(node_18),
node_19, ID(node_19),
edge_33, ID(edge_33),
node_20, ID(node_20),
edge_35, ID(edge_35),
edge_36, ID(edge_36),
node_21, ID(node_21),
edge_37, ID(edge_37),
edge_38, ID(edge_38),
node_22, ID(node_22),
edge_39, ID(edge_39),
edge_40, ID(edge_40),
edge_41, ID(edge_41),
edge_42, ID(edge_42),
node_23, ID(node_23)
A. Appendix

A.2 Parallelization Pattern

A.2.1 XML Serialization

Listing A.3. The XML serialization representing our Parallelization Pattern

```xml
<?xml version="1.0" encoding="UTF-8"?>
<propertygraph xmlns:xmi="http://www.omg.org/XMI"
    xmlns:propertygraph="http://de.cau.se.grapheditor.propertygraph/propertygraph"
    nodeCounter="33" edgeCounter="61">

    <nodes label="array=temp0" color="FFFFFF" ID="node_1">
        <keyValuePairs key="type" value="Assignment"/>
        <keyValuePairs key="operation" value="value"/>
        <keyValuePairs key="vartype" value="int[]"/>
        <outgoingEdges label="DATA_FLOW" color="000000" ID="edge_45" targetNode="node_24">
            <keyValuePairs key="type" value="DATA_FLOW"/>
            <keyValuePairs key="var" value="array"/>
        </outgoingEdges>
        <outgoingEdges label="CONTROL_FLOW" color="000000" ID="edge_46" targetNode="node_24">
            <keyValuePairs key="type" value="CONTROL_FLOW"/>
        </outgoingEdges>
    </nodes>

    <nodes label="sum=0" color="FFFFFF" ID="node_3" incomingEdges="edge_51">
        <keyValuePairs key="type" value="Assignment"/>
        <keyValuePairs key="vartype" value="int"/>
        <outgoingEdges label="DATA_FLOW" color="000000" ID="edge_53" targetNode="node_18">
            <keyValuePairs key="type" value="DATA_FLOW"/>
            <keyValuePairs key="var" value="sum"/>
        </outgoingEdges>
    </nodes>

    <nodes label="append" color="FFFFFF" ID="node_18" incomingEdges="edge_53">
        <keyValuePairs key="type" value="MethodCallWithReturnValue"/>
        <keyValuePairs key="fqn" value="java.lang.StringBuffer.append(int)"/>
        <keyValuePairs key="returntype" value="java.lang.StringBuffer"/>
    </nodes>

    <nodes label="temp9=\&lt;java.lang.System\>
    color="FFFFFF" ID="node_23" incomingEdges="edge_52">
        <keyValuePairs key="type" value="Assignment"/>
    </nodes>
```
A.2. Parallelization Pattern

```xml
<keyValuePairs key="vartype" value="java.io.PrintStream"/>
<keyValuePairs key="operation" value="staticFieldAccess"/>
</nodes>

<nodes label="stream" color="FFFFFF" ID="node_24"
incomingEdges="edge_45 edge_46">
<keyValuePairs key="type" value="MethodCallWithReturnValue"/>
<keyValuePairs key="fqn" value="java.util.Arrays.stream(int[])"/>
<keyValuePairs key="argumentscount" value="1"/>
<keyValuePairs key="returntype" value="java.util.stream.IntStream"/>
<keyValuePairs key="name" value="stream"/>
<outgoingEdges label="DATA_FLOW" color="000000" ID="edge_47"
targetNode="node_25">
<keyValuePairs key="type" value="DATA_FLOW"/>
<keyValuePairs key="var" value="stream(array)"/>
</outgoingEdges>
<outgoingEdges label="CONTROL_FLOW" color="000000" ID="edge_48"
targetNode="node_25">
<keyValuePairs key="type" value="CONTROL_FLOW"/>
</outgoingEdges>
<outgoingEdges label="CALLS" color="000000" ID="edge_54"
targetNode="node_27">
<keyValuePairs key="type" value="CALLS"/>
</outgoingEdges>
</nodes>

<nodes label="parallel" color="FFFFFF" ID="node_25"
incomingEdges="edge_47 edge_48">
<keyValuePairs key="type" value="MethodCallWithReturnValue"/>
<keyValuePairs key="fqn" value="java.util.stream.IntStream.parallel()"/>
<keyValuePairs key="name" value="parallel"/>
<keyValuePairs key="returntype" value="int"/>
<outgoingEdges label="CONTROL_FLOW" color="000000" ID="edge_49"
targetNode="node_26">
<keyValuePairs key="type" value="CONTROL_FLOW"/>
</outgoingEdges>
<outgoingEdges label="DATA_FLOW" color="000000" ID="edge_50"
targetNode="node_26">
<keyValuePairs key="type" value="DATA_FLOW"/>
<keyValuePairs key="var" value="stream(array).parallel()"/>
</outgoingEdges>
<outgoingEdges label="CALLS" color="000000" ID="edge_59"/>
```

99
A.2. Parallelization Pattern

```
  <outgoingEdges>
  </nodes>
  <nodes label="java.util.stream" color="FFFFFF" ID="node_30">
  <keyValuePairs key="type" value="Package"/>
  <keyValuePairs key="name" value="java.util.stream"/>
  <outgoingEdges label="CONTAINS_TYPE" color="000000" ID="edge_57" targetNode="node_31">
  <keyValuePairs key="type" value="CONTAINS_TYPE"/>
  </outgoingEdges>
  </nodes>
  <nodes label="IntStream" color="FFFFFF" ID="node_31" incomingEdges="edge_57">
  <keyValuePairs key="type" value="Class"/>
  <keyValuePairs key="name" value="IntStream"/>
  <outgoingEdges label="CONTAINS_METHOD" color="000000" ID="edge_58" targetNode="node_32">
  <keyValuePairs key="type" value="CONTAINS_METHOD"/>
  </outgoingEdges>
  <outgoingEdges label="CONTAINS_METHOD" color="000000" ID="edge_60" targetNode="node_33">
  <keyValuePairs key="type" value="CONTAINS_METHOD"/>
  </outgoingEdges>
  </nodes>
  <nodes label="parallel" color="FFFFFF" ID="node_32">
  <keyValuePairs key="type" value="Method"/>
  <keyValuePairs key="name" value="parallel"/>
  </nodes>
  <nodes label="sum" color="FFFFFF" ID="node_33">
  <keyValuePairs key="type" value="Method"/>
  <keyValuePairs key="name" value="sum"/>
  </nodes>
  <keyValueColorTriplets key="forbiddentype" value="FORBIDDEN" color="FF0000"/>
  <keyValueColorTriplets key="forbiddenedge" value="FORBIDDEN" color="FF0000"/>
```

101
A. Appendix

A.2.2 Recorded User Interaction Protocol

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>EDGE_ELEM_DEL, edge_37</td>
</tr>
<tr>
<td>2</td>
<td>EDGE_ELEM_DEL, edge_38</td>
</tr>
<tr>
<td>3</td>
<td>EDGE_ELEM_DEL, edge_39</td>
</tr>
<tr>
<td>4</td>
<td>EDGE_ELEM_DEL, edge_40</td>
</tr>
<tr>
<td>5</td>
<td>EDGE_ELEM_DEL, edge_41</td>
</tr>
<tr>
<td>6</td>
<td>EDGE_ELEM_DEL, edge_42</td>
</tr>
<tr>
<td>7</td>
<td>NODE_ELEM_DEL, node_22</td>
</tr>
<tr>
<td>8</td>
<td>EDGE_ELEM_DEL, edge_9</td>
</tr>
<tr>
<td>9</td>
<td>EDGE_ELEM_DEL, edge_10</td>
</tr>
<tr>
<td>10</td>
<td>EDGE_ELEM_DEL, edge_11</td>
</tr>
<tr>
<td>11</td>
<td>EDGE_ELEM_DEL, edge_14</td>
</tr>
<tr>
<td>12</td>
<td>EDGE_ELEM_DEL, edge_15</td>
</tr>
<tr>
<td>13</td>
<td>NODE_ELEM_DEL, node_8</td>
</tr>
<tr>
<td>14</td>
<td>EDGE_ELEM_DEL, edge_43</td>
</tr>
<tr>
<td>15</td>
<td>NODE_ELEM_DEL, node_10</td>
</tr>
<tr>
<td>16</td>
<td>EDGE_ELEM_DEL, edge_16</td>
</tr>
<tr>
<td>17</td>
<td>EDGE_ELEM_DEL, edge_29</td>
</tr>
<tr>
<td>18</td>
<td>EDGE_ELEM_DEL, edge_17</td>
</tr>
<tr>
<td>19</td>
<td>EDGE_ELEM_DEL, edge_44</td>
</tr>
<tr>
<td>20</td>
<td>NODE_ELEM_DEL, node_12</td>
</tr>
<tr>
<td>21</td>
<td>EDGE_ELEM_DEL, edge_4</td>
</tr>
<tr>
<td>22</td>
<td>EDGE_ELEM_DEL, edge_5</td>
</tr>
<tr>
<td>23</td>
<td>EDGE_ELEM_DEL, edge_6</td>
</tr>
<tr>
<td>24</td>
<td>NODE_ELEM_DEL, node_5</td>
</tr>
<tr>
<td>25</td>
<td>EDGE_ELEM_DEL, edge_21</td>
</tr>
<tr>
<td>26</td>
<td>EDGE_ELEM_DEL, edge_22</td>
</tr>
<tr>
<td>27</td>
<td>EDGE_ELEM_DEL, edge_23</td>
</tr>
<tr>
<td>28</td>
<td>EDGE_ELEM_DEL, edge_24</td>
</tr>
<tr>
<td>29</td>
<td>EDGE_ELEM_DEL, edge_25</td>
</tr>
<tr>
<td>30</td>
<td>NODE_ELEM_DEL, node_15</td>
</tr>
<tr>
<td>31</td>
<td>EDGE_ELEM_DEL, edge_18</td>
</tr>
<tr>
<td>32</td>
<td>EDGE_ELEM_DEL, edge_19</td>
</tr>
<tr>
<td>33</td>
<td>NODE_ELEM_DEL, node_13</td>
</tr>
<tr>
<td>34</td>
<td>EDGE_ELEM_DEL, edge_32</td>
</tr>
<tr>
<td>35</td>
<td>EDGE_ELEM_DEL, edge_33</td>
</tr>
<tr>
<td>36</td>
<td>NODE_ELEM_DEL, node_19</td>
</tr>
<tr>
<td>37</td>
<td>EDGE_ELEM_DEL, edge_7</td>
</tr>
<tr>
<td>38</td>
<td>EDGE_ELEM_DEL, edge_8</td>
</tr>
</tbody>
</table>

102
A.2. Parallelization Pattern

39  NODE_ELEM_DEL, node_7
40  EDGE_ELEM_DEL, edge_27
41  EDGE_ELEM_DEL, edge_28
42  EDGE_ELEM_DEL, edge_30
43  NODE_ELEM_DEL, node_17
44  EDGE_ELEM_DEL, edge_31
45  EDGE_ELEM_DEL, edge_34
46  EDGE_ELEM_DEL, edge_35
47  EDGE_ELEM_DEL, edge_36
48  NODE_ELEM_DEL, node_20
49  EDGE_ELEM_DEL, edge_1
50  NODE_ELEM_DEL, node_2
51  EDGE_ELEM_DEL, edge_20
52  NODE_ELEM_DEL, node_14
53  NODE_ELEM_DEL, node_21
54  NODE_ELEM_DEL, node_16
55  NODE_ELEM_DEL, node_6
56  NODE_ELEM_DEL, node_11
57  EDGE_ELEM_DEL, edge_2
58  NODE_ELEM_ADD, node_24
59  NODE_PROP_ADD, node_24, type, MethodCallWithReturnValue
60  EDGE_ELEM_ADD, edge_45, node_1, node_24
61  EDGE_PROP_ADD, edge_45, type, DATA_FLOW
62  EDGE_ELEM_ADD, edge_46, node_1, node_24
63  EDGE_PROP_ADD, edge_46, type, CONTROL_FLOW
64  NODE_PROP_ADD, node_24, args, array
65  NODE_PROP_ADD, node_24, fqn, java.util.Arrays.stream(int[])
66  NODE_PROP_ADD, node_24, argumentscount, 1
67  NODE_PROP_ADD, node_24, returntype, java.util.stream.IntStream
68  NODE_PROP_ADD, node_24, name, stream
69  EDGE_PROP_ADD, edge_45, var, array
70  NODE_ELEM_ADD, node_25
71  NODE_PROP_ADD, node_25, type, MethodCallWithReturnValue
72  EDGE_ELEM_ADD, edge_47, node_24, node_25
73  EDGE_PROP_ADD, edge_47, type, DATA_FLOW
74  EDGE_ELEM_ADD, edge_48, node_24, node_25
75  EDGE_PROP_ADD, edge_48, type, CONTROL_FLOW
76  EDGE_PROP_ADD, edge_47, var, stream(array)
77  NODE_PROP_ADD, node_25, fqn, java.util.stream.IntStream.parallel()
78  NODE_PROP_ADD, node_25, name, parallel
79  NODE_PROP_ADD, node_25, returntype, int
A. Appendix

```
80  NODE_ELEM_ADD, node_26
81  NODE_PROP_ADD, node_26, type, MethodCallWithReturnValue
82  EDGE_ELEM_ADD, edge_49, node_25, node_26
83  EDGE_PROP_ADD, edge_49, type, CONTROL_FLOW
84  EDGE_ELEM_ADD, edge_50, node_25, node_26
85  EDGE_PROP_ADD, edge_50, type, DATA_FLOW
86  EDGE_PROP_ADD, edge_50, var, stream(array).parallel()
87  NODE_PROP_ADD, node_26, fqn, java.util.stream.IntStream.sum()
88  NODE_PROP_ADD, node_26, name, sum
89  EDGE_ELEM_ADD, edge_51, node_26, node_3
90  EDGE_PROP_ADD, edge_51, type, DATA_FLOW
91  EDGE_PROP_ADD, edge_51, var, stream(array).parallel().sum()
92  EDGE_ELEM_ADD, edge_52, node_26, node_23
93  EDGE_PROP_ADD, edge_52, type, CONTROL_FLOW
94  EDGE_ELEM_ADD, edge_53, node_3, node_18
95  EDGE_PROP_ADD, edge_53, type, DATA_FLOW
96  EDGE_PROP_ADD, edge_53, var, sum
97  NODE_ELEM_ADD, node_27
98  NODE_PROP_ADD, node_27, type, Method
99  EDGE_ELEM_ADD, edge_54, node_24, node_27
100 EDGE_PROP_ADD, edge_54, type, CALLS
101 NODE_ELEM_ADD, node_28
102 NODE_PROP_ADD, node_28, type, Class
103 EDGE_ELEM_ADD, edge_55, node_28, node_27
104 EDGE_PROP_ADD, edge_55, type, CONTAINS_METHOD
105 NODE_PROP_ADD, node_27, stream, null
106 NODE_PROP_SET, node_27, stream, 
107 NODE_PROP_DEL, node_27, stream, 
108 NODE_PROP_ADD, node_27, name, stream
109 NODE_PROP_ADD, node_28, name, Arrays
110 NODE_ELEM_ADD, node_29
111 NODE_PROP_ADD, node_29, type, Package
112 NODE_PROP_ADD, node_29, name, java.util
113 EDGE_ELEM_ADD, edge_56, node_29, node_28
114 EDGE_PROP_ADD, edge_56, type, CONTAINS_TYPE
115 NODE_ELEM_ADD, node_30
116 NODE_PROP_ADD, node_30, type, Package
117 NODE_PROP_ADD, node_30, name, java.util.stream
118 NODE_ELEM_ADD, node_31
119 NODE_PROP_ADD, node_31, type, Class
120 NODE_PROP_ADD, node_31, name, IntStream
```
A.2. Parallelization Pattern

Listing A.5. The Cypher query for Listing A.4

MATCH (node_1)-[edge_1:DATA_FLOW]->(node_2),
  (node_1)-[edge_2:CONTROL_FLOW]->(node_3),
  (node_1)-[edge_18:DATA_FLOW]->(node_13),
  (node_2)-[edge_9:DATA_FLOW]->(node_8),
  (node_2)-[edge_10:CONTROL_FLOW]->(node_8),
  (node_3)-[edge_4:CONTROL_FLOW]->(node_5),
  (node_3)-[edge_31:DATA_FLOW]->(node_18),
  (node_5)-[edge_5:CONTROL_FLOW]->(node_6),
  (node_5)-[edge_6:CONTAINS_UNIT]->(node_6),
  (node_6)-[edge_7:CONTROL_FLOW]->(node_7),
  (node_6)-[edge_11:DATA_FLOW]->(node_8),
  (node_6)-[edge_20:DATA_FLOW]->(node_14),
  (node_6)-[edge_34:DATA_FLOW]->(node_20),
  (node_7)-[edge_8:CONTROL_FLOW]->(node_2),
  (node_8)-[edge_14:CONTROL_FLOW]->(node_10),
  (node_8)-[edge_15:CONTROL_FLOW]->(node_11),
  (node_10)-[edge_43:CONTROL_FLOW]->(node_23),
  (node_11)-[edge_16:CONTROL_FLOW]->(node_12),
  (node_12)-[edge_17:CONTROL_FLOW]->(node_13),
  (node_12)-[edge_44:DATA_FLOW]->(node_16),
A. Appendix

\[
\begin{align*}
(n_13) \xrightarrow{\text{edge}_19: \text{CONTROL}_F\text{LOW}} & (n_14), \\
(n_13) \xrightarrow{\text{edge}_23: \text{DATA}_F\text{LOW}} & (n_15), \\
(n_14) \xrightarrow{\text{edge}_21: \text{DATA}_F\text{LOW}} & (n_15), \\
(n_14) \xrightarrow{\text{edge}_22: \text{CONTROL}_F\text{LOW}} & (n_15), \\
(n_15) \xrightarrow{\text{edge}_24: \text{DATA}_F\text{LOW}} & (n_16), \\
(n_15) \xrightarrow{\text{edge}_25: \text{CONTROL}_F\text{LOW}} & (n_16), \\
(n_16) \xrightarrow{\text{edge}_27: \text{DATA}_F\text{LOW}} & (n_17), \\
(n_16) \xrightarrow{\text{edge}_28: \text{CONTROL}_F\text{LOW}} & (n_17), \\
(n_17) \xrightarrow{\text{edge}_29: \text{DATA}_F\text{LOW}} & (n_12), \\
(n_17) \xrightarrow{\text{edge}_30: \text{DATA}_F\text{LOW}} & (n_18), \\
(n_19) \xrightarrow{\text{edge}_32: \text{CONTROL}_F\text{LOW}} & (n_19), \\
(n_19) \xrightarrow{\text{edge}_33: \text{CONTROL}_F\text{LOW}} & (n_20), \\
(n_20) \xrightarrow{\text{edge}_35: \text{DATA}_F\text{LOW}} & (n_21), \\
(n_20) \xrightarrow{\text{edge}_36: \text{CONTROL}_F\text{LOW}} & (n_21), \\
(n_21) \xrightarrow{\text{edge}_37: \text{DATA}_F\text{LOW}} & (n_22), \\
(n_21) \xrightarrow{\text{edge}_38: \text{CONTROL}_F\text{LOW}} & (n_22), \\
(n_22) \xrightarrow{\text{edge}_39: \text{DATA}_F\text{LOW}} & (n_20), \\
(n_22) \xrightarrow{\text{edge}_40: \text{DATA}_F\text{LOW}} & (n_22), \\
(n_22) \xrightarrow{\text{edge}_41: \text{DATA}_F\text{LOW}} & (n_8), \\
(n_22) \xrightarrow{\text{edge}_42: \text{CONTROL}_F\text{LOW}} & (n_7).
\end{align*}
\]

WHERE ID(edge_37)=62

AND ID(edge_38)=15

AND ID(edge_39)=66

AND ID(edge_40)=65

AND ID(edge_41)=64

AND ID(edge_42)=14

AND ID(edge_9)=47

AND ID(edge_10)=26

AND ID(edge_11)=48

AND ID(edge_14)=13

AND ID(edge_15)=25

AND ID(node_8)=23

AND ID(edge_43)=12

AND ID(node_10)=24

AND ID(edge_16)=24

AND ID(edge_29)=67

AND ID(edge_17)=23

AND ID(edge_44)=86

AND ID(node_12)=34

AND ID(edge_4)=31

AND ID(edge_5)=30
A.2. Parallelization Pattern

```
AND ID(edge_6)=29
AND ID(node_5)=19
AND ID(edge_21)=51
AND ID(edge_22)=21
AND ID(edge_23)=60
AND ID(edge_24)=52
AND ID(edge_25)=20
AND ID(node_15)=37
AND ID(edge_18)=73
AND ID(edge_19)=22
AND ID(node_13)=35
AND ID(edge_32)=18
AND ID(edge_33)=17
AND ID(node_19)=40
AND ID(edge_7)=28
AND ID(edge_8)=27
AND ID(node_7)=21
AND ID(edge_27)=63
AND ID(edge_28)=19
AND ID(edge_30)=68
AND ID(node_17)=39
AND ID(edge_31)=70
AND ID(edge_34)=50
AND ID(edge_35)=71
AND ID(edge_36)=16
AND ID(edge_1)=72
AND ID(node_2)=22
AND ID(edge_20)=49
AND ID(node_14)=36
AND ID(node_21)=42
AND ID(node_16)=38
AND ID(node_6)=20
AND ID(node_11)=33
AND ID(edge_2)=32
DELETE edge_37
DELETE edge_38
DELETE edge_39
DELETE edge_40
DELETE edge_41
DELETE edge_42
DELETE edge_9
```
A. Appendix

103 DELETE edge_10
104 DELETE edge_11
105 DELETE edge_14
106 DELETE edge_15
107 DETACH DELETE node_8
108 DELETE edge_43
109 DETACH DELETE node_10
110 DELETE edge_16
111 DELETE edge_29
112 DELETE edge_17
113 DELETE edge_44
114 DETACH DELETE node_12
115 DELETE edge_4
116 DELETE edge_5
117 DELETE edge_6
118 DETACH DELETE node_5
119 DELETE edge_21
120 DELETE edge_22
121 DELETE edge_23
122 DELETE edge_24
123 DELETE edge_25
124 DETACH DELETE node_15
125 DELETE edge_18
126 DELETE edge_19
127 DETACH DELETE node_13
128 DELETE edge_32
129 DELETE edge_33
130 DETACH DELETE node_19
131 DELETE edge_7
132 DELETE edge_8
133 DETACH DELETE node_7
134 DELETE edge_27
135 DELETE edge_28
136 DELETE edge_30
137 DETACH DELETE node_17
138 DELETE edge_31
139 DELETE edge_34
140 DELETE edge_35
141 DELETE edge_36
142 DELETE edge_1
143 DETACH DELETE node_2
A.2. Parallelization Pattern

DELETE edge_20
DELETE node_14
DETACH DETACH DELETE node_21
DETACH DETACH DELETE node_16
DETACH DETACH DELETE node_6
DETACH DETACH DELETE node_11
DELETE edge_2
CREATE (node_24)
SET node_24.type="MethodCallWithReturnValue"
CREATE (node_1)-[edge_45:DATA_FLOW]->(node_24)
CREATE (node_1)-[edge_46:CONTROL_FLOW]->(node_24)
SET node_24.args="array"
SET node_24.fqn="java.util.Arrays.stream(int[])"
SET node_24.argumentscount="1"
SET node_24.returntype="java.util.stream.IntStream"
SET node_24.name="stream"
SET edge_45.var="array"
CREATE (node_25)
SET node_25.type="MethodCallWithReturnValue"
CREATE (node_24)-[edge_47:DATA_FLOW]->(node_25)
CREATE (node_24)-[edge_48:CONTROL_FLOW]->(node_25)
SET edge_47.var="stream(array)"
SET node_25.fqn="java.util.stream.IntStream.parallel()"
SET node_25.name="parallel"
SET node_25.returntype="int"
CREATE (node_26)
SET node_26.type="MethodCallWithReturnValue"
CREATE (node_25)-[edge_49:CONTROL_FLOW]->(node_26)
CREATE (node_25)-[edge_50:DATA_FLOW]->(node_26)
SET edge_50.var="stream(array).parallel()"
SET node_26.fqn="java.util.stream.IntStream.sum()"
SET node_26.name="sum"
CREATE (node_26)-[edge_51:DATA_FLOW]->(node_3)
SET edge_51.var="stream(array).parallel().sum()"
CREATE (node_26)-[edge_52:CONTROL_FLOW]->(node_23)
CREATE (node_3)-[edge_53:DATA_FLOW]->(node_18)
SET edge_53.var="sum"
CREATE (node_27)
SET node_27.type="Method"
CREATE (node_24)-[edge_54:CALLS]->(node_27)
CREATE (node_28)
A. Appendix

```
SET node_28.type="Class"
CREATE (node_28)-[edge_55:CONTAINS_METHOD]->(node_27)
SET node_27.name="stream"
CREATE (node_29)
SET node_29.type="Package"
SET node_29.name="java.util"
CREATE (node_29)-[edge_56:CONTAINS_TYPE]->(node_28)
CREATE (node_30)
SET node_30.type="Package"
SET node_30.name="java.util.stream"
CREATE (node_31)
SET node_31.type="Class"
SET node_31.name="IntStream"
CREATE (node_30)-[edge_57:CONTAINS_TYPE]->(node_31)
CREATE (node_32)
SET node_32.type="Method"
SET node_32.name="parallel"
CREATE (node_31)-[edge_58:CONTAINS_METHOD]->(node_32)
CREATE (node_25)-[edge_59:CALLS]->(node_32)
CREATE (node_33)
SET node_33.type="Method"
SET node_33.name="sum"
CREATE (node_31)-[edge_60:CONTAINS_METHOD]->(node_33)
CREATE (node_26)-[edge_61:CALLS]->(node_33)
```
Appendix B

Manual for the Creation of Patterns
B. Manual for the Creation of Patterns

- Manual -
Pattern Creation
with the Graph Editor

Erik Koppenhagen


SOFTWARE ENGINEERING GROUP
DEPARTMENT OF COMPUTER SCIENCE
KIEL UNIVERSITY
# Table of Contents

I  First Steps .............................. 1
   I.1  Preconditions .......................... 1
   I.2  Creating a New Graph .................... 1
   I.3  Setting up the GUI ...................... 3

II  GUI Overview ......................... 4
   II.1  The Tree View ......................... 4
   II.2  Important Tabs ...................... 5
      II.2.1  The Properties Tab ................ 5
      II.2.2  The Layout Tab .................... 5
   II.3  The Diagram Panel ................... 6
   II.4  The Context Menu .................... 6
   II.5  The Menu Bar .......................... 7
      II.5.1  The Color Choice Dialog .......... 7
      II.5.2  The Recorder ...................... 7
      II.5.3  The Translator .................... 8

III  Creating Patterns .................... 9
   III.1  Working with the Context Menu ....... 9
   III.2  Creating a Candidate Pattern ........ 10
      III.2.1  Creating a Graph ................ 10
      III.2.2  Translating the Graph .......... 10
   III.3  Creating a Parallelization Pattern .... 10
      III.3.1  Creating a Graph ................. 10
      III.3.2  Translating the Graph .......... 10
I First Steps

I.1 Preconditions

This manual provides instructions on how to create Candidate Patterns and Parallelization Patterns with the graph editor. This requires the graph editor to be set up correctly. Therefore, please follow the instructions of the installation manual provided by L. E. Blühnke.

I.2 Creating a New Graph

To create a new graph, a graph project is required. Right-click on the Project Explorer and select New → Other... Alternatively, use the shortcut Ctrl + N. Scroll down to Propertygraph and select Propertygraph Project. Then, click Next> and enter a name for the project. (Figure 1)

![Figure 1: Creating a new propertygraph project](image)

To create a new graph, select the src folder in the created project. Add a Propertygraph File with the same menu. Then, click Next> and choose a file name. Confirm the name with the Finish button. (Figure 2)
Figure 2: Creating a new propertygraph file

The Eclipse runtime application should now be similar to Figure 3.

Figure 3: The Eclipse runtime application before changes to the layout
I.3 Setting up the GUI

The Eclipse runtime application creates a folder on the hard drive called runtime-EclipseApplication. It stores projects, files and the settings of the application. When launching the Eclipse runtime application for the first time, the GUI layout has to be configured.

At first, close Outline and Task List on the left side of the GUI. Then, double-click on the .propertygraph file in the project. A new tab with the name of the file opens on the bottom of the GUI. Select and move this tab to the middle of the screen, so that the upper half is divided in two areas. Right-click in the white space of the moved tab and select Show Layout View. In the tab on the right, expand the structure until Key Value Color Triplets forbidden type and Key Value Color Triplets forbidden edge are visible. The Eclipse runtime application should now be similar to Figure 4.

Figure 4: The Eclipse runtime application after changes to the layout
II GUI Overview

II.1 The Tree View

The Tree View tab shows the structure of the current pattern. The root node is always the propertygraph itself. Edges are children of nodes. Properties can be children of nodes or edges. (Figure 5)

![Figure 5: The Tree View containing a graph structure](image)
B. Manual for the Creation of Patterns

II.2 Important Tabs

II.2.1 The Properties Tab

The Properties tab shows the properties of the currently selected element in the Tree View. The information vary on the element. (Figure 6)

![Figure 6: The Properties tab](image)

II.2.2 The Layout Tab

The Layout tab shows various information about the current property graph. They can be modified to change the layout of the pattern. (Figure 7)

![Figure 7: The Layout tab](image)
II.3 The Diagram Panel

The Diagram Panel visualizes the pattern. Changing the pattern results in a repaint of the panel. A right-click on the panel opens the Context Menu and allows to create and design a pattern. (Figure 8)

Figure 8: The Diagram Panel visualizing a graph

II.4 The Context Menu

The Context Menu opens when right-clicking the Diagram Panel. It offers various possibilities to create and modify a pattern. (Figure 9)
The functionality is covered in the section Working with the Context Menu.

Figure 9: The Context Menu of the Diagram Panel
II.5 The Menu Bar

The Menu Bar offers three additional buttons. The first button (from left to right) opens the Color Choice Dialog. The second button activates/deactivates the Recorder. The third button opens the Translator. (Figure 10)

Figure 10: The Menu Bar with additional buttons (red)

II.5.1 The Color Choice Dialog

The Color Choice Dialog allows to create and delete key-value-color triplets. The color has to be defined with a hexadecimal value. (Figure 11)

Figure 11: The Color Choice Dialog with two key-value-color triplets

II.5.2 The Recorder

The Recorder saves all changes made to a pattern in a file.
II.5.3 The Translator

The Translator transforms the pattern to a Neo4j Cypher query. This allows to find the designed structures in a graph database. (Figure 12)

![The Translator at initialization](image)

Figure 12: The Translator at initialization
B. Manual for the Creation of Patterns

III Creating Patterns

III.1 Working with the Context Menu

The Context Menu is essential for the creation of patterns. It provides all required operations. Patterns should only be created and modified with the Context Menu.

- **Show Layout View**
  Opens the Layout Tab shown in the subsection The Layout Tab.

- **Add Template**
  A rudimentary entry from the KLightD plugin without any functionality.

- **Clip selected**
  Fits the size of the pattern to the current size of the Diagram Panel.

- **Clip towards selection**
  Zooms on the currently selected element.

- **Clip parent**
  A rudimentary entry from the KLightD plugin without any functionality.

- **Clip root**
  A rudimentary entry from the KLightD plugin without any functionality.

- **Scale up**
  Increases the size of the currently selected element.

- **Scale down**
  Decreases the size of the currently selected element.

- **Scale 100%**
  Sets the size of the currently selected element back to default.

- **Save as Image...**
  Opens a dialog to save the pattern to an image file.

- **Export KGraphs**
  Opens a dialog to save the graph structure to a file.

- **Add New Node**
  Allows to add a new node to the pattern. A submenu offers different types of nodes, sorted by layers. Each layer represents an abstraction level.

- **Connect Nodes**
  Connects to selected nodes with an edge. A submenu offers different types of nodes, sorted by layers. Each layer represents an abstraction level.

- **Set Type**
  Sets the type of the currently selected element. A submenu offers different types of nodes, sorted by layers. Each layer represents an abstraction level.

- **Key-Value Pairs**
  Allows to add a new key-value pair to the currently selected or modify the existing ones.
• Remove Element
   Removes the currently selected element from the pattern.

• Save history
   Opens a dialog to save the .history file to a specific location.

III.2 Creating a Candidate Pattern

III.2.1 Creating a Graph

Create a new Propertygraph File. Design the pattern with the Context Menu and save the file.

III.2.2 Translating the Graph

To translate the pattern to a Neo4j Cypher query, open the Translator and click the load file button. Select the .propertygraph file of the Candidate Pattern and click the translate UIP button. Applying the query to a graph database returns a string table, which is required for the Parallelization Pattern.

III.3 Creating a Parallelization Pattern

III.3.1 Creating a Graph

Open a Candidate Pattern. To create a Parallelization Pattern for this pattern, activate the Recorder in the Menu Bar. Then, transform the pattern to the structure of a Parallelization Pattern, using the Context Menu. At last, deactivate the Recorder.

III.3.2 Translating the Graph

At first, open the Translator. Click the load file button and select the .history file of the Candidate Pattern. Then, replace the content of the code field with the string table from the Candidate Pattern query. At last, click the translate UIP button.
Appendix C

CD Content

The included CD contains multiple folders and files. In the following, we list the folders on the top level and explain their content.

Evaluation Data
Contains the files we used for the evaluation. This includes the Candidate Pattern, the Parallelization Pattern and the system dependency graphs.

Graph Editor
Contains our GIT branch of the graph editor. The graph editor includes the translator and our changes presented in Chapter 4.

Java2Neo4j
Contains the software to transform Java applications to Neo4j graph databases.

Manual
Contains the manual presented in Appendix B and all corresponding \LaTeX files. The pictures are included, too.

Sources
Contains the sources and related work of this thesis.

Thesis
Contains the thesis, all pictures and all corresponding \LaTeX files.


Bibliography

