

NORMAN KÖSTER

AN EXTENSIBLE GRAPH QUERY LANGUAGE
FOR MODEL-BASED INFORMATION RETRIEVAL
IN INTELLIGENT ENVIRONMENTS

AN EXTENSIBLE GRAPH QUERY LANGUAGE
FOR MODEL-BASED INFORMATION RETRIEVAL
IN INTELLIGENT ENVIRONMENTS

NORMAN KÖSTER

Domain analysis, conceptualization, implementation, and empirical evaluation

A doctoral thesis presented for the degree of
Doctor of Engineering (Dr.-Ing.) at

Faculty of Technology
Bielefeld University
Inspiration 1
33619 Bielefeld
Germany

REVIEWERS

Prof. Dr. Philipp Cimiano
Dr. Sebastian Wrede
Prof. Dr. Nico Hochgeschwender

BOARD

Prof. Dr. Ulrich Rückert
Dr. Malte Schilling

DEFENDED AND APPROVED

April 29th, 2020

Printed on permanent paper as per ISO 9706.

ABSTRACT

Research on human behavior and the (non-)verbal interactions executed in these situations makes increasing use of intelligent systems, such as robot companions or *smart environments*. To allow for valuable and robust communication in these socially involved scenarios, systems executing *human-robot interaction (HRI)* are strongly tied to and dependent on the data and knowledge provided by the various sensors and software components of the system. This data ranges from low-level raw sensor data to higher-level domain-specific knowledge derived by components applying, for example, machine learning techniques. Additionally, these systems and environments are characteristically extensively heterogeneous and highly complex, yielding large amounts of data following different schemata at diverse granularities. State-of-the-art systems therefore often structure and store available knowledge using graph-based structures, which represent the domain-specific entities and their relations. This raises the question on *how* to provide access to the full range of data and domain-specific knowledge of the intelligent systems in a manageable and (ideally) supportive manner.

In this thesis I investigate the applicability of a *Model-driven Software Engineering (MDSE)* approach to assist *behavior developers* of intelligent systems and environments by supporting the information retrieval process. I analyze how extensive modeling of the domain can support the retrieval and query creation process already at query design time. Therefore, I examine questions on *domain-specific language (DSL)* design, semantics, and composition to identify what the necessary conceptualizations are for providing an extensible graph query language, which exploits the available model-based knowledge. I further describe my efforts implementing a *vertical prototype* which realizes a functional slice of the proposed system. Within a detailed evaluation, which tests the implementation on users in a real world application context, I analyze the viability and advantages of my approach compared to a baseline condition making use of state-of-the-art tools. I measure cognitive load of users, task solving duration, and additionally multiple usability metrics. The results show that in terms of *usability*, the presented *vertical prototype* does not reach the professional tooling of the baseline condition and is perceived as less *usable* by users when designing *graph database queries (GDQs)*. However, once the users overcame the initial learning curve, they require less *effort* and are more *effective* when designing domain-specific *GDQs* using the implemented conceptualizations.

ACKNOWLEDGMENTS

This journey towards a PhD was an incredible experience and naturally I would not have been able to achieve this monumental task without the support from the ones around me.

I am the most grateful to Marlena Dorniak. Ever since our paths crossed you provided me with nothing short of the greatest support. You encouraged me to discover, challenge, push, and accept the boundaries of my own potential at every step of the way – even on long and busy days. Especially during the finalizing phase when I could not return the time and efforts as much as I would have liked to. You also provide perspective when my own tunnel vision hinders me to see the bigger picture. Thank you so very much.

I also extend great gratitude to my parents Doris and Edward Köster. You have always supported me during my life and provided me all options to finish this endeavor. Without your help I would not have reached this point in my life. Thank you very much.

Academically, I am grateful to my doctoral supervisors Prof. Dr. Philipp Cimiano and Dr. Sebastian Wrede. Thank you very much for providing me with this opportunity and for supporting me through this process with ideas, feedback, and guidance. Our discussions and your views had great impact on form and shape of this work. Also, many thanks to Prof. Dr. Nico Hochgeschwender for agreeing to join this process as a reviewer. Further, great help and discussions were always provided by all colleagues in my work group, the overarching projects, and all other PhD candidates and students I crossed paths with in the past. Thank you all very much.

Last but absolutely not least, I am very thankful to my friends. You are the social net which catches me on good and on difficult days. Thank you for being there and allowing us to laugh and cry together. Special thanks to Phillip Lücking for always thinking outside of the technocrat-box with me, in which we found ourselves in ever so often. Without your perspective I would have been without prospects more than once. Also special appreciation and thanks to Dr. Sebastian Schneider. Our joint journey ever since we started studying in 2006 has enriched me on so many levels. Your encouragement and your own curiosity kept also me going and searching for more. It is difficult to find the correct words to express how thankful I am to have you as a close friend: Thank you very much.

Thanks to everyone who gave me feedback: Sebastian, Marlena, Jan, Michael, Hendrik, Dennis, Vera, Marco, and everyone else I was unable to name here personally: Thank you!

CONTENTS

I	RESEARCH TOPIC	1
1	INTRODUCTION	3
1.1	Research questions and contribution	6
1.2	Outline	7
II	PRELIMINARIES	9
2	GRAPH-BASED KNOWLEDGE REPRESENTATION AND MANAGEMENT	11
2.1	Data, information, and knowledge modeling	12
2.2	Graphs and their role in intelligent systems	14
2.3	Graph-based Knowledge management	17
2.3.1	NoSQL: Graph databases	18
2.3.2	Graph query languages	22
2.4	Summary	28
3	MODEL-DRIVEN SOFTWARE ENGINEERING	29
3.1	Foundations and introduction	29
3.1.1	Models and transformations	29
3.1.2	Domain-specific languages	31
3.1.3	Benefits of MDSE	42
3.2	Application of MDSE in adjacent domains	43
3.3	MDSE development process	44
3.4	Summary	47
III	MODELING INTERACTION RELEVANT KNOWLEDGE IN SMART ENVIRONMENTS	49
4	A MODEL OF INTERACTION RELEVANT DATA	51
4.1	Embodied interaction in smart environments	52
4.2	Domain analysis	53
4.2.1	The CSRA Project	53
4.2.2	Roles, responsibilities, and required knowledge	57
4.2.3	Knowledge queries in the EISE domain	59
4.3	Related work	61
4.4	A multi-modal interaction corpus	64
4.5	An ontology of interaction relevant knowledge	65
4.5.1	Smart environment ontologies	66
4.5.2	Ontologies in robotics	67
4.5.3	Graph-based approaches	69
4.5.4	The EISE ontology	70
4.6	Summary	72

5	CONCEPTUALIZATIONS FOR MODEL-BASED QUERY COMPOSITION	75
5.1	Objectives and requirements	75
5.1.1	Requirements	77
5.1.2	Functional requirements	77
5.1.3	Non-functional requirements	77
5.2	Related work	78
5.3	System architecture	79
5.4	Extensible graph query language composition	81
5.4.1	Representation of graphs	84
5.4.2	Representation of pattern matching queries	84
5.4.3	Representation of domain descriptions	88
5.4.4	Representation of time	91
5.4.5	Plug-ins and implementation modules	100
5.5	Technology mapping	101
5.6	Summary	102
	IV MODEL-BASED SUPPORT FOR BEHAVIOR DEVELOPERS	105
6	IMPLEMENTATION AND PRACTICAL CONCERNS	107
6.1	Language implementation	107
6.1.1	Language composition	109
6.1.2	Graphs and graph query languages	112
6.1.3	Domain description language	114
6.1.4	Time languages	115
6.1.5	Transformations and generation of queries	116
6.1.6	Language pragmatics	118
6.2	Automation aspects in applied MDSE research	123
6.2.1	Continuous integration of DSLs	124
6.2.2	Language deployment: A DSL plug-in server	126
6.3	User perspective: The EISE Query Designer	128
6.4	Summary	130
	V EVALUATION OF MDSE APPROACHES	133
7	EVALUATION AND APPLICATION	135
7.1	Introduction to MDSD evaluation	136
7.2	Evaluation metrics	138
7.3	Evaluation of the EISE Query Designer	140
7.3.1	Methods and study design	142
7.3.2	Measurements	146
7.3.3	Study results	147
7.3.4	Discussion	149
7.4	Summary	152

	VI PERSPECTIVES	155
8	OUTLOOK	157
9	CONCLUSION	161
	VII APPENDIX	163
A	EVALUATION APPENDIX	165
	A.1 Full questionnaire	165
	A.2 Study information material	167
	A.3 Ethics documents	184
	A.4 Ethics committee application	184
	A.5 Consent form	185
	A.6 Questionnaire results	186
	ACRONYMS	189
	GLOSSARY	193
	BIBLIOGRAPHY	199
	Involved and own publications	199
	General	200
	Online resources	215
	Software packages	216

LIST OF FIGURES

Figure 2.1	<i>DIKW</i> pyramid model mapping	12
Figure 2.2	<i>GDB</i> popularity trends	21
Figure 3.1	<i>MOF</i> abstraction model of the <i>OMG</i>	31
Figure 3.2	Four types of language composition	36
Figure 3.3	Screenshot of the <i>Xtext language workbench</i>	39
Figure 3.4	Screenshot of the <i>MPS language workbench</i>	40
Figure 3.5	Applied <i>MDSE</i> development process	46
Figure 4.1	The <i>CSRA</i> living room	54
Figure 4.2	Map view of the <i>CSRA</i> laboratory	55
Figure 4.3	Exemplary interaction scenario	59
Figure 4.4	The <i>EISE</i> interaction ontology	71
Figure 5.1	System structure diagram	80
Figure 5.2	Language modularization and composition	82
Figure 5.3	<i>Meta-model</i> of the Graph language	84
Figure 5.4	Reduced <i>meta-model</i> of the Graph Query language	85
Figure 5.5	<i>Meta-model</i> of the Domain Description language	88
Figure 5.6	The <i>meta-model</i> of the Time language.	92
Figure 5.7	The <i>meta-model</i> of the Relative Time language.	93
Figure 5.8	Absolute temporal expansion examples	94
Figure 5.9	Relative temporal expansion examples	94
Figure 5.10	Temporal Graph Query language <i>meta-model</i>	95
Figure 5.11	Two approaches for time abstraction within a graph	97
Figure 5.12	Technology mapping diagram	102
Figure 6.1	Implementation language composition diagram	108
Figure 6.2	<i>Devkit module</i> composition	109
Figure 6.3	Alluvial diagram of the language dependencies	111
Figure 6.4	Cypher language <i>concrete syntax</i>	113
Figure 6.5	Domain Graph Description language syntax (1)	115
Figure 6.6	Domain Graph Description language syntax (2)	115
Figure 6.7	<i>Concrete syntax</i> of the temporal languages	116
Figure 6.8	<i>MPS</i> based Cypher generator example	117
Figure 6.9	Temporal query constraint generator example	118
Figure 6.10	Graph visualization projection implementation	120
Figure 6.11	Editor aspects of visualization projection	121
Figure 6.12	Example of the query explain feature	122
Figure 6.13	<i>EISEQD solution</i> dependency diagram	123
Figure 6.14	<i>MPS module</i> build dependency graph	126
Figure 6.15	Screenshot of the <i>EISEQD</i> interface	128
Figure 6.16	Screenshot of the query editor in the <i>EISEQD</i>	130
Figure 7.1	Screenshot of the Neo4j web interface	141

Figure 7.2	The executed in-between study design	143
Figure 7.3	Execution time results boxplot per set	148
Figure 7.4	Cognitive load results boxplot per set	148
Figure 7.5	Syntactical error rates results per task	149
Figure 7.6	Categorized keystrokes results per set	149
Figure 7.7	<i>UEQ</i> and <i>SUS</i> usability questionnaire results	150
Figure A.1	<i>SUS</i> questionnaire	165
Figure A.2	<i>TLX</i> questionnaire	165
Figure A.3	<i>UEQ</i> questionnaire	166

LIST OF TABLES

Table 2.1	Overview of existing graph databases	20
Table 4.1	Estimation of sensory data amount in the <i>CSRA</i> . . .	57
Table 6.1	Aspect statistics of implemented languages	110

LIST OF CODE LISTINGS

Listing 2.1	Example Cypher query	24
Listing 2.2	Example <i>SPARQL</i>	25
Listing 2.3	Example Gremlin query	27
Listing 5.1	<i>EBNF</i> excerpt from the openCypher language	86
Listing 6.2	Exemplary project file for the Cypher <i>DSL</i>	126
Listing 6.3	Example entry within a <i>updatePlugins.xml</i>	127
Listing 6.4	Generated Cypher query code	129

NOTATION

MARGIN NOTES

- Key point
-  Definition

LANGUAGES AND CONCEPTS

The names of *domain-specific languages* and concepts of these are written in a non-proportional font, e.g. Relative Time. Concepts of languages are additionally written using medial capitals, e.g. DomainDescriptionGraph.

DENOTATIONAL SEMANTICS

The *denotational semantics* of element E of the *abstract syntax* of a language L is described by $\llbracket E \rrbracket_L$. The natural association of any language construct to its denotational meaning (its natural identity) is accordingly expressed by $\llbracket E \rrbracket_L$. *Concrete syntax* of the target language within the *denotational semantics* are written as green characters. We assume there exists a distinct empty element ϵ with the following properties. Further, the operator \oplus_t is used for element concatenation into target semantic of type t such that for any given left expression L , right expression R , and empty element ϵ the following holds:

$$\begin{aligned} \llbracket L \oplus_t \epsilon \rrbracket_L &= \llbracket L \rrbracket_L & \llbracket L \oplus_t R \rrbracket_L &= \llbracket L \rrbracket_L \oplus_t \llbracket R \rrbracket_L \\ \llbracket \epsilon \oplus_t R \rrbracket_L &= \llbracket R \rrbracket_L & \llbracket \epsilon \oplus_t \epsilon \rrbracket_L &= \epsilon \end{aligned}$$

For example:

$$\llbracket L \oplus_{\text{AND}} R \rrbracket_{\text{Cypher}} = \llbracket L \rrbracket_{\text{Cypher}} \text{ AND } \llbracket R \rrbracket_{\text{Cypher}}$$

ATTRIBUTION OF AUTHORSHIP

I will speak of myself using *I* in case of work originally done by myself alone. In case the results of a collaboration with others are presented, I will use *we*. The respective collaborators are indicated by the co-authors of the publication the results are based on.

Part I

RESEARCH TOPIC

Introduction to the research topic, the adjacent domains, and formulation of the research questions investigated in this thesis.

INTRODUCTION

“All sorts of things can happen when you’re open to new ideas and playing around with things.”

—Stephanie Kwolek
chemist who invented Kevlar and winner of
the Lavoisier Medal for technical achievements

Research investigating humans, their behavior, and (non-)verbal interaction makes increasingly use of artificial intelligent systems such as robot companions or intelligent environments [Gar+07; Atk+00; FND03]. Any *human–robot interaction (HRI)* executed in these systems is strongly dependent on the data and knowledge available as to accomplishing valuable and robust communication [GS07]. It is thus imperative for software components orchestrating the interactive scenarios to have access to the relevant data and derived knowledge which is available in the system. However, these systems and environments are characteristically extensively heterogeneous and highly complex, yielding large amounts of data at diverse granularities [CD05]. This raises the question on *how* to provide access to the full range of low-level data to high-level domain-specific information of the intelligent systems in a manageable and (ideally) supportive manner.

The aforementioned shared systems incorporate the domains of intelligent/*smart environments* and embodied cognition in *HRI* into a joint environment. This combined environment houses both, an autonomous embodied robot (e.g. a companion robot) and an ubiquitous system (e.g. a smart home) operating together. A common shared space as such is referred to as the *Embodied Interaction in Smart Environments (EISE)* domain in which the systems jointly support humans in their daily lives [Hol+16a]. While a physically distributed system commonly only provides an overview on the complete environment, it lacks on the one hand sensors for searching tasks (e.g. finding misplaced keys in a smart home), on the other hand actuators for manipulation (e.g. picking up and handing over found keys) which are both necessary for rich embodied interactions with the environment. In contrast to this, embodied agents are often equipped with high quality local sensors required for navigation or pick-and-place tasks and can thus conduct a such a detailed local analysis.

The emergent behavior of these systems can be separated into low-level functionalities (e.g. autonomous navigation, object detection, or manipulation tasks) and high-level behavioral functionalities (e.g. cooperative task solving, verbal interaction, or other forms of *HRI*). To

realize higher level behaviors, such as increasingly complex *HRI* scenarios involving multimodal data sources and interaction partners, the available raw sensor data is commonly enriched yielding derived information and knowledge (e.g. via machine learning or sensor data fusion). Research objectives in these domains for example cover questions regarding the creation and design of robust interaction [Ric+16], recovering from weak or eventually broken interactions [CSW16], safe *HRI* and trust humans place into robots [BES19], or requirements towards software/hardware infrastructure and architecture [Wre+17]. The creation and execution of such complex *HRI* scenarios is thus a non-trivial task and poses an interdisciplinary challenge including other distinct domains such as linguistics and psychology. Only few researchers from these adjacent non-engineering domains have a computer science background or the accompanying programming expertise which is commonly necessary to program these extensive *HRI* behaviors for the above mentioned systems. In addition to this, the (non-)functional requirements towards systems in research settings strongly differ to requirements towards commercial products and projects. Researchers reside in a more volatile setting and the development process needs to be adapted accordingly. They are for example exposed to constant development and system training efforts, exploratory (re-implementation efforts, volatile software, exceedingly complex systems, or drastically changing systems and environments within short time spans.

In this thesis I attend the perspective of *behavior developers* who create complex system behaviors. I am concerned with topics regarding *which* data is actually relevant for natural interaction and *how* the developers can be supported in their efforts to create interaction behavior by exploiting available domain-specific knowledge. This involves in detail two perspectives:

1. *What data is relevant for interaction and how can it be abstracted?*
2. *How can behavior developers be supported in the process of interaction relevant data retrieval within complex interactive systems?*

The first perspective attends to the adjacent domain of graph-based knowledge representation. In recent years there is a rise of applications which represent the knowledge and information of the domain at hand in graph structures. Renowned examples for the application of graphs are a) trees in the *Robot Operating System (ROS)* coordinate transform library TF2 [Foo13], b) knowledge bases such as ontologies for the organization of task specific knowledge as *ORO* or *KnowRob* [TB13; Bee+18; Lem+10], or also c) the in recent years advancing use of *Graph Database Management System (GDB)* as a storage backend for relational knowledge [Ne007]. Their application often targets the use of graphs as modeling frameworks used to abstract and represent the real world. In contrast to structures in classic *Database*

Management System (DBMS), the graph structure inherently is concerned with connections between the nodes of the graph. This eases data retrieval as it provides sub-graph pattern matching, graph traversal functionalities, and specialized graph algorithms operating on the graph structure and consequently the structure of the data itself.

The second perspective is concerned with the software engineering side of interaction design and creation. Depending on the research goals and changing study system configurations, *behavior developers* adapt their programs requiring an extensive knowledge of low-level system properties (e.g. system architecture, data structures, or storage properties). Commonly they fall back on *General Purpose Languages (GPLs)* such as Java, Python, or C++ and the full set of generic language features to create the intended interactions in individual software components interconnected via the common middleware [ES12; Fer15; Fis+18]. These applications further increase complexity with respect to the system design and its usage. To be able to create optimal retrieval queries from the available data sources, *behavior developers* need to be aware where data is stored, what schema it follows, and how to access the data. Following a *Model-driven Software Engineering (MDSE)* approach is increasingly popular to manage similar (accidental) complexity of artificial systems [Rod15a]. The key idea is to provide *domain-specific languages (DSLs)* and tools to describe the real world and its details in a language close to the domain and known by its experts. As a result, the created formalizations such as *meta-models* allow its users to capture and model the domain, consequently reducing the semantic gap between original and actual implementation. Additional tooling can then provide the developers with analysis and *model* checking at the time of query composition. Helpful functionality emerges, such as static analysis, domain-specific code completion, or extensive debugging support. Additionally, the ability to generate code automatically based on the created user *models* positively impacts productivity, quality, validation, and verification [Fow10; Völ13a; vKV00; Com17].

In this thesis I combine these two perspectives with the goal to provide *behavior developers* with domain-specific support during information retrieval. Considering both perspectives, it becomes apparent that an extensible and unified query interface is required for the developers. As with the storage systems in the first perspective, state-of-the-art systems commonly rely on the *DBMS* accompanying query languages as the information retrieval interface – with various abstraction and expressive capabilities, for example *Structured Query Language (SQL)* or Prolog [Bee+18]. When put in the context of frequently adapting research settings, the creation of a domain-specific query language requires further special attention. Especially the application of a *MDSE* approach underlines the need for an extensible query language to avoid recurrent *meta-model* (and consequently user

model) changes. This aspect is thus managed twofold in this thesis: First, domain model modifications need to be possible, not only by the hand of a language engineer, but also by researchers themselves. The language compositions for this domain hence requires designs and mechanisms on the *meta-model* layer which enable domain model changes in the model space by the domain experts themselves. Second, the language composition needs to respect possible future and present extensions which are within the considered *EISE* domain. As such, extension points and language adaption opportunities needs to be realized to allow the addition of orthogonal or adjacent domain with manageable effort. Languages consequently need to realize appropriate differentiation and separation regarding language cohesion and coupling.

I present four contributions realizing this goal. First, I analyze the *EISE* domain and its stakeholders to finally derive a *model* of interaction relevant data. This *model* is the foundational abstraction of the domain knowledge and serves as the basis for the second contribution of this thesis: Implementation-independent conceptualizations describing a system, which realizes an extensible graph query language. This involves the identification of suitable languages and domain-specific extensions (e.g. temporal expansion or user domain data schemata) to provide model-based graph knowledge retrieval to *behavior developers* of the *EISE* domain. Third, I implement a *vertical prototype integrated development environment (IDE)* realizing the theoretical considerations as a proof-of-concept. Fourth and last, I evaluate this prototype in an extensive user study. The evaluation covers metrics such as the *cognitive load* and error rate during query design to analyze the impact of such a system and show benefits and potential limitations of the application.

1.1 RESEARCH QUESTIONS AND CONTRIBUTION

- goal and re-
search questions* ○ The overarching top-level goal of this thesis distills to:

Provide DSL conceptualizations that support developers of HRI scenarios in the process of query design, execution, and maintenance of interaction relevant information. Follow a MDSE approach that incorporates an extensive empirical evaluation on a vertical prototype with a high level of evidence.

In more detail, the individual research questions I investigate in this work are derived from the above mentioned challenges. These questions are:

RQ1 What are central elements and relations of interaction in the *EISE* domain and how can these be modeled such that commonly occurring questions can be answered?

- RQ2** What is a suitable *MDSE* development process and what are the (non-)functional requirements, which allow to create an extensible query language?
- RQ3** What are implementation-independent *meta-models*, semantics, and compositions strategies of *DSLs*, which provide a suitable approach to reach the identified requirements?
- RQ4** How can the implementation-independent abstraction be realized? What are language *pragmatics* resulting in a working *vertical prototype* that provides the identified functionalities?
- RQ5** Is an integrated graph data query support solution such as an *IDE* viable and does it provide an advantage to the usual workflow with state-of-the-art tools? How does it perform in terms of usability, complexity reduction and feature support?

1.2 OUTLINE

This thesis is separated into six parts which investigate the above stated research questions. This initial Part **I** opens up this thesis and introduces the research topic as well as the involved domains. It motivates the application of a *MDSE* approach for supported query design in the *EISE* domain. Further, this introduction presents the five research questions which are investigated in the remainder of the thesis. Lastly, I explicitly comprise my contributions included in this thesis.

The second Part **II** summarizes the theoretical background for this work. These preliminaries are split into two sections which introduce the two adjacent domains presented in the afore named perspectives. The first domain of graph-based knowledge representation focuses on knowledge representation, storage, and retrieval using graphs structures. State-of-the-art graph-based knowledge storage systems are discussed in this section and available *graph query languages (GQLs)* and their characteristics are analyzed. The second part initially examines *MDSE* in artificial systems and the graph domain. Foundations of *models*, *DSLs* (including semantics, language composition, and *language workbenches*), and the *MDSE* process are presented. This part concludes in the presentation of the overarching iterative *MDSE* development process, which I applied during my research (**RQ2**). The six individual phases of the process are used to frame the presented work in the remaining sections.

The third Part **III** presents the initial two contributions of this thesis. First, a domain analysis of the *EISE* domain alongside the im-

plementation example of the *Cognitive Service Robotics Apartment as Ambient Host (CSRA)* project is described in Chapter 4. This analysis extracts the existing roles, responsibilities, and competency questions (knowledge queries) of the domain. The information is used to compose a *model* of interaction relevant knowledge in the *EISE* domain (RQ1). The second contribution in Chapter 5 presents the results of the application of the *MDSE* approach. Extracted requirements, corresponding languages, their composition, and *denotational semantics* are discussed implementation-independent and in-depth (RQ2 and RQ3). This part closes with a technology mapping which grounds these theoretical considerations into specific technical choices as used in the reference project.

Part IV describes the implementation of the concepts presented in the analysis phase. The central contribution is a *vertical prototype* developed using the *language workbench* MPS that realizes the abstract *models* (RQ4). Pragmatics of the language development are shown, especially how a domain description and temporal constraints are included into the graph query design process.

The prototype implementation is the basis for the evaluation presented in Part V. My contributions presented in this section amount to study design, execution, and result discussion (RQ5). I measure usability and application benefits via multiple metrics, as I consider the perspective of users to be highly relevant for model-based applications.

The last Part VI concludes this thesis by opening perspectives. I present possible future work extending the presented contributions, before closing with a brief discussion of the results of this work with respect to to the initially stated research questions.

Part II

PRELIMINARIES

Theoretical background of graph-based knowledge representation and *Model-driven Software Engineering*.

GRAPH-BASED KNOWLEDGE REPRESENTATION AND RETRIEVAL IN INTERACTIVE INTELLIGENT SYSTEMS

“I didn’t want to just know the names of things. I remember really wanting to know how it all worked.”

—Elizabeth Blackburn [Bla09],
Biological Researcher at the University of California
awarded the 2009 Nobel Prize in Physiology/Medicine.

Graphs are a central component of interactive intelligent systems. Their conceptual abstractions provide properties which can successfully be used at different data processing layers of the system. Most prominently, graphs are often used as the foundation for abstractions, which create a reduced representation of the real world, for example, in state machines, markov models, or neural networks. Specialized algorithms on graphs such as shortest/longest path or cycle detection allow subsequently to exploit the graph structure of the stored data efficiently and intuitively. With an increased interest in graph data management in the last decade, graph-based data storage, query, and analysis tools have been developed. Also, the use in academic fields such as research on robotics or *human–robot interaction (HRI)* increased and graph databases are employed more frequently in experimental systems [Hoc+16; Fou+17; HRH16]. Further, graphs are a foundational element in the *Model-driven Software Engineering (MDSE)* process, for example, the *abstract syntax tree (AST)* of a *model* is represented using a graph structure. Thus, I investigate the application of *Graph Database Management Systems (GDBs)* over traditional *Database Management Systems (DBMSs)* for interaction relevant data in the domain of *Embodied Interaction in Smart Environments (EISE)*. This chapter presents an introduction to graph-based knowledge representation and retrieval in interactive intelligent systems.

The remainder of this chapter is thus organized as follows. To contextualize the usage of graph-based knowledge within interactive intelligent systems I will firstly recap clear conceptualization of data, information, and knowledge via the *data-information-knowledge-wisdom (DIKW)* hierarchy. I map the elements of this hierarchy to common components of interactive artificial systems, showing the system’s connection between the *DIKW* concepts. In the then following [Section 2.2](#), I give a concise introduction to graphs, their representations, and their role in the different layers of the *DIKW* hierarchy. The remaining sections define the requirements for successful information

and knowledge management in the context of interactive artificial systems. This includes an analysis of existing *GDBs* as well as suitable *graph query languages (GQLs)* for the application in interactive intelligent systems.

2.1 DATA, INFORMATION, AND KNOWLEDGE MODELING

The *DIKW* hierarchy represents a central abstraction for information management, information systems and knowledge management in general. With no clear origin of its first presentation, I follow the revisited analysis of the *DIKW* hierarchy as presented by Rowley [Row07]. The model pyramid is depicted in Figure 2.1 and shows the increasing structure, meaning, and transferability of its elements from bottom to top. In contrast to this gain in value, the size of the pyramid parts decrease from bottom to top reflecting the respectively available amount.

Within this hierarchy, *data* lies at the lowest level and is characterized by being discrete observations which are unprocessed collected signals. For an intelligent system, this layer transfers to the actual data recorded by the low level sensors of the system (e.g. camera images, microphone recordings, or laser scanner data).

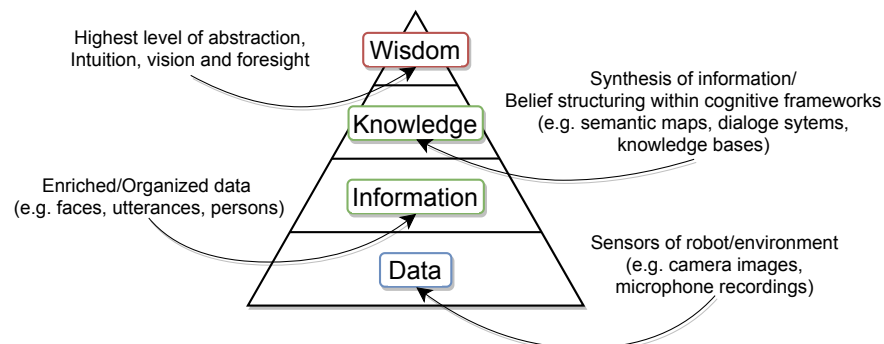


Figure 2.1: The *DIKW* pyramid and its mapping to interactive intelligent systems.

The second layer on top of data represents *information* and is commonly defined as structured data [Row07]. This layer has been organized to give the data relevance for a specific context, thus enriching its value and relevance for a given application. Aggregating low level sensor data of an intelligent system this way can, for example, yield information such as spoken utterances, face detection, or even person percepts.

Knowledge is the third layer in the *DIKW* pyramid. Depending on the perspective (philosophical, biological or entirely technical), definitions vary strongly. However, a common key distinction is made between tacit knowledge and embedded knowledge. The former is embedded in the individual, whilst the latter is recorded, and in turn,

explicitly defined for sharing. More generally, information is a concept that is commonly described as being a reference to underlying data [Row07]. Further structuring of information and beliefs creates a synthesis over a certain time span. Knowledge is thus an expert opinion composed of information, experience understanding, and skills – either obtained from previous information (tacit) or previously dormant within the individual (embedded). The transfer of the concept of knowledge to intelligent systems can best be represented via large systems which contain system goals, semantic maps, or knowledge bases. A common goal of those systems is to rely on tacit knowledge, rather than embedding all required knowledge into a system at its creation time.

The top layer of *wisdom* is often omitted from the pyramid and as such it is rarely a clearly defined concept. Available definitions point out that wisdom is the highest level of abstraction, which can enable an entity to predict and provide a form of foresight [Row07]. Thus, wisdom can allow to transfer and apply concepts from one domain to previously unseen situations in different domains. Exhibiting generalizing viable wisdom in an artificial system remains the most difficult unsolved task, as it requires all lower tiers of the pyramid to be considered and available.

When viewed from the perspective of robotics and artificial intelligent systems, all layers of the *DIKW* have their own unique requirements for storage and retrieval of relevant information. Systems need to manage the high complexity, whilst providing solutions for low level drivers, the perception, abstraction, reasoning, and further behavior building upon these. The question on *how* to extract and provide perception data is considered solved in current state-of-the-art applications for artificial systems and robotic systems by the general utilization of event-based architectures [Qui+09; MFN06; WW11]. Systems using this architecture can cope with the heterogeneous hardware and software uses of the domain while still allowing a growing scale and scope. Developers separate the functionality to collect and provide data into individual software packages that communicate over the common middleware. This approach, in turn, breaks down the overall system complexity into small manageable chunks allowing the reuse of highly specialized software by other developers (e.g. in the *Robot Operating System (ROS)* eco-system). This facilitates rapid prototyping of hard- and software experiments (with helpful features such as debugging, collaboration support, monitoring/introspection) while allowing large-scale integrative robotics research. Higher layers of the *DIKW* are build on top into individual components of the underlying system. Information is extracted in these components and shared into the event-based architecture. Popular approaches incorporate, for example, knowledge bases (cf. Section 2.3 on page 17), which allow systems to infer new information over previous information of

(interaction) episodes. The *KnowRob* project presents such a knowledge based approach. The project aims to provide a computational resource to bridge the gap between vague task description and low level information needed for robotic manual task execution [TB13; Bee+18]. Similarly, Lemaignan et al. presented the *Open Robot Ontology (ORO)*, an ontology based knowledge processing framework supporting agents with cognition in *HRI* environments [Lem+10].

As stated in research question **RQ3**, my goal is to make use of model-based domain knowledge to support the information retrieval process on layers above the data layer. Developers already have dedicated access to the data layer via the middleware and programming of behavioral components also uses this transport layer. In contrast to the data level, this thesis therefore targets retrieval of information and knowledge (green elements of Figure 2.1) from the developer perspective.

2.2 GRAPHS AND THEIR ROLE IN INTELLIGENT SYSTEMS

The concepts, terminology, and structure of graphs suggest their application as a modeling framework for abstractions of the real world. In these cases, the graphs provide mechanisms to manage knowledge in a contextually appropriate form. The following (sub-)domains of artificial intelligent systems are, for example, well represented using graphs:

- (a) (Finite-)state machines for system coordination or task execution [BC10; Lüt+11; SGK17],
- (b) Ontologies, or knowledge graphs, used as a tool for the formal definition and representation of real world entities or knowledge [TB13; Bee+18; Lem+10],
- (c) Machine learning (e.g. markov models or artificial neural networks) used to process complex data inputs in *HRI* and allow task execution, such as, image recognition, speaker identification, or gesture recognition [NLK12],
- (d) Trees to appropriately represent domain-specific knowledge, for example, the transform library as a part of *ROS*, which is used for the representation of coordinate frames, their relations, and respective transformations [Foo13]
- (e) Graphs to represent the structure of *meta-models* as used in the *MDSE* workflow (cf. Chapter 3 on page 29) [Rod15a]

The final example is of great importance to this thesis and the four-layered abstraction model in Figure 3.1 on page 31 shows an exemplary application, which makes heavy use of connected graphs featuring labels, properties, direction, and their overall structures to allow (meta)

modeling in the context of *MDSE*. In all cases graphs are used to represent or hold knowledge relevant to the overall system and/or (sub-)system parts and are of existential importance to their applications.

Claude [Cla66] initially presented the topic of graph theory and today numerous detailed introductions into graphs and graph theory exist in literature [Wil99; Wes01; Gro08]. As graphs are a central underlying concept for my work, I will briefly introduce (non-simple) graphs, a common notation used in the remaining thesis.

Fundamentally, a graph G is represented by the ordered pair

$$G = (V, E) \quad (2.1)$$

consisting of a finite set of nodes (also called vertices)¹

$$V = V(G) = \{n_0, \dots, n_m\} \quad (2.2)$$

and a finite set of edges $E(G)$ containing ordered pairs of elements of $V(G)$. An edge

$$E = E(G) = \{(n_x, n_y) \mid n_x, n_y \in V(G)\} \quad (2.3)$$

connects (or joins) the two nodes n_x and n_y and thus expresses the relationship between them. When using ordered pairs of elements within an edge, one can express directed edges between nodes of a graph, thus expressing source and target of the relationship. The graph is then called a *directed* graph or *digraph* opposed to the otherwise *undirected* graph.

A *loop* can exist within a graph which is an edge composed of an equal pair of nodes, i.e. connecting the node to itself:

$$e_{\text{loop}} = \{n_x, n_x\} \mid n_x \in V(G) \quad (2.4)$$

Further, a *multidigraph* refers to a graph which allows connecting any node with a directed edge to any other node. Edges of the same type can, as a result, connect the same nodes multiple times.

A graph H is called a subgraph of a graph G (i.e. $H \subseteq G$) if the set of nodes and edges of H are a subset of the nodes and edges of G , that is

$$V(H) \subseteq V(G) \quad \text{and} \quad E(H) \subseteq E(G) \quad (2.5)$$

Alternatively to the previously mentioned connected graphs, disconnected graphs G_A and G_B can coexist and are defined by having no connection between any nodes. A common operation on connected graphs involves walks alongside its nodes and edges. A connection

¹ In the remainder of this thesis the words node and vertex, as well as edge and relationship are used interchangeably.

walk along a sequence of nodes $P = (n_0, \dots, n_k)$ is also called a *path* between two nodes. In the particular case that $n_0 = n_k$ the path is considered to be closed and called a *cycle*. For example, paths play an important role in the context of tree structures, which in turn are connected graphs with only one path between each pair of nodes.

graph model ○ A further extension of graphs adds more descriptive aspects to its entities via labels and properties. Labeled graphs allow to attach different labels from a finite set Lab to nodes and edges, while property graphs allow to assign multiple properties from finite sets Prop and constants Const to nodes and edges respectively.

$$\begin{aligned}
 G &= (V, E, \rho, \lambda, \sigma) \\
 V &= \{v_0, \dots, v_j\} \\
 E &= \{e_0, \dots, e_k\} && | \text{ where } V \cap E = \emptyset \\
 \rho : E &\rightarrow (V \times V) && | \rho(e_x) = (v_a, v_b) \\
 \lambda : (V \cup E) &\rightarrow \mathcal{P}(\text{Lab}) && (2.6) \\
 \sigma : (V \cup E) \times \text{Prop} &\rightarrow \text{Val} \\
 \text{Lab} &= \{l_0, \dots, l_l\} \\
 \text{Prop} &= \{p_0, \dots, p_m\} \\
 \text{Const} &= \{c_0, \dots, c_n\}
 \end{aligned}$$

With E being a finite set of edges, the function $\rho(e_x) = (v_a, v_b)$ consequently defines that e_x is a directed edge from node v_a to node v_b in graph G .

To further allow the graph G to represent labeled *multidigraphs* (i.e. multiple directed edges between nodes with identical source, target, and label(s)), λ projects onto a powerset of the finite set of labels Lab .

labeled property multidigraph ☞ These graph properties combined are referred to as a *labeled property multidigraph* (in the following simply referred to as a graph) and can be represented by current state-of-the-art *GDBs*, such as Neo4j. There are more graphs properties with less importance to the application in this thesis which are not discussed in depth at this point; for further details I suggest reading common literature on graphs [Wil99; Wes01].

Recent research continues evolve graphs to support large-scale data management formally. For example, Shinavier and Wisnesky recently presented a formal lingua for algebraic property graphs and an exemplary implementation [SW19]. The authors define a *labeled property multidigraph* similar to Equation (2.6) and also include a notion of graph schema as well as the concept of hypergraphs². Their description of graphs is a fundamental perspective and contains similar definitions as presented in this section.

² Hypergraphs are a graph in which an edge can connect any number of nodes and in which edges are also vertexes that can be connected by further edges

2.3 GRAPH-BASED KNOWLEDGE MANAGEMENT

The term knowledge management term was first coined in 1974 by Henry and refers to the entire process of crating, sharing, storing, retrieving, and managing knowledge [Hen74]. This scopes the processes of collecting and storing data, information, and knowledge in *DBMS* for increased accessibility. The classic relational model was introduced by Codd in 1970 proposing to represent data as sets of tuples (relations) alongside a suitable first-order predicate logic to describe queries (i.e. *Structured Query Language (SQL)*) [Cod70]. The relations are a collection of tables consisting of sets of rows and columns. Modifications on these tables are realized via relational operators for tabular manipulations.

Consequently, the underlying model imposes strong constraints on inserted data. The data model core advantage is its uniformity and as such, the problem domain is always mapped to this model [Mai83]. As a result, since its wide application no considerations were made whether or not the relational model is appropriate for a particular set of data or information. However, while the relational model is widely adopted and used, model shortcomings are increasingly documented [Ang12]. One central limitation is that data model with high coupling (i.e. the degree of interdependence between elements) often drastically impacts querying performance negatively [SF13]. Additionally, the relational model is not well suited for data containing uncertainty. Each row of a table is considered a true proposition in this model. Analytical processing, statistical data, and fundamentally changing data is thus not easily represented.

Similarly, the query language *SQL* [CB76] has been heavily criticized – especially in its early years by its author [Dat84; Dat87; Dat12; Atz+13]. The most common issues with *SQL* refer to its

- Lack of consistency (abstract and concrete syntax are inconsistent),
- Lack of compactness (large and growing language),
- Lack of orthogonality (hard to compose),
- Host language mismatch (low system cohesion; no integration with application languages and protocols).

As the relational model contains limitations that do not cover the requirements of current applications, alternating databases increased in popularity [Ang12]. *NoSQL* databases emerged which address the shortcomings by providing alternative data models and appropriate query languages. The most popular approaches provide models such as key-value stores, document stores, graph, triple stores, or multi-model stores. These databases exploit the structure of the data and

information to be stored, providing higher performance or accessibility for appropriate domains. Consequently, *NoSQL* databases and their performance and application suitability are researched intensively; multiple comparisons of *NoSQL* databases discuss the individual properties [TB11; MK14]. In the context of this work, *GDBs* and the suitable *GQLs* are analyzed in detail in the following sections.

2.3.1 *NoSQL: Graph databases*

GDBs are one of the widely adopted *NoSQL* storage types which employ a fundamentally different data model compared to the traditional relational model for knowledge storage. This type of storage utilizes a structured graph model with similar model definitions to the previously presented *labeled property multidigraph* in Equation (2.6) on page 16. Besides the exploitation of data structure in the model, *GDBs* mitigate the aforementioned limitations of relational *DBMSs* by providing a (computationally) cheap traversal alongside the edges of the graph. The costs of edge traversal are in turn shifted to the insertion time and higher investments (e.g. via more complex statements) are necessary when inserting data into the *GDB*. With the importance of graphs for computer science and knowledge management, multiple *GDB* implementations are available addressing individual functional and non-functional requirements of developers and system users [Ang12]. They can be categorized into two categories: a) single-node platforms providing high efficiency with limited scalability (e.g. Neo4j, OrientDB), and b) large-scale distributed systems (including cloud solutions) with efficiency impacts due to distribution and overhead (e.g. AragonDB, Hadoop). Each implementation supports a wide range of different graph specific algorithms, including graph traversing, path (sub-graph pattern) finding, graph metric calculation, shortest path calculations, cluster detection, or graph similarity calculation. Table 2.1 on page 20 presents a condensed list of state-of-the-art graph databases and their core features and properties. In the following I compare the different implementations along the following categories to show their adequacy for my work to fit the previously introduced research question RQ4 (Section 1.1 on page 6) and the upcoming implementation requirements FR1, FR7, NFR1, NFR2, NFR5 (Section 5.1 on page 75):

- 1) Access (the application in research requires to consider source code access and potential licensing)
- 2) Specialization (the degree of language specialization opposed to generalization of the query language)
- 3) Query interface (users require an intuitive interface for effective query composition)

- 4) Ranking (overall popularity impacts the choice as further development and use of the query language benefits its usage)
- 5) Overall applicability (does the language provide all features necessary for the domain application).

With the application of the target *GDB* in public funded research, (source) access and licensing impact my *GDB* choice. Besides downsides such as vendor lock-in, reduced innovative approaches, and high complexity of corporate applications, factors such as the availability (especially for paid only licensing models), adaptability and overall mindset of closed source systems do not align with my research requirements. However, some companies offer a dual-license model, as for example the *Neo Technology* organization does for their *GDB* Neo4j. In this example, the community edition is open source and freely available, while enterprise edition primarily provides exclusive support and specialized features. Unfortunately, few *GDB* (i.e. Sparksee and GraphDB) invoke deal-breaking constraints on the community editions – such as upper limits of nodes or no parallel query execution – rendering them unusable for the application in my work.

Most of the databases provide implementations of specialized features for narrow domains and employ different graph database models that correspond to their intended application. These models include graph models tailored for spatial and geographical data (e.g. Oracle Spatial) and *Resource Description Framework (RDF)* triplestores (e.g. Apache Jena). Only three of the selected solutions use the detailed and for my work best suitable *labeled property multidigraph* (cf. [Equation \(2.6\)](#) on page 16) as their underlying graph model and the remaining examples use a multi-model approach that combines different models.

The *RDF* based databases (e.g. Apache Jena or AllegroGraph) primarily provide retrieval interfaces using (extended) *SPARQL Protocol and RDF Query Language (SPARQL)* or *SQL* query languages due to the underlying triple store model. The remaining databases use specialized query languages such as Cypher, Gremlin, AQL, or provide access via specialized APIs. *GDBs* with support for the query languages openCypher and Gremlin are a good fit for the application in the *EISE* domain as shown in [Section 2.3.2](#) on page 22. These include the databases Neo4j, JanusGraph, and AnzoGraph from [Table 2.1](#) on the next page. While Cypher was initially presented as a part of Neo4j in 2011, it is pushing for a broader adoption since 2015 via a full and open specification in the openCypher project. As a result, other platforms (recently) added suitable support, for example, AgensGraph, CAPS (Cypher for Apache Spark), Apache TinkerPop (via Cypher for Gremlin), Memgraph, RedisGraph, or SAP HANA Graph.

SOURCE	ID	NAME	GRAPH MODEL TYPE	IMPL. LANGUAGE	QUERY LANGUAGE	RANKING ^R	ACID	LICENSE	SCALABILITY
ACCESS	1	Neo4j ¹	property graph	Java	Cypher	1	✓	dual-license	✓([Dom+10])
	2	OrientDB	multi-model	Java	SQL	3	✓	Apache License 2.0	-
open source	3	JanusGraph ²	graph database	Java	Gremlin (TinkerPop)	7	✓	Apache License 2.0	✓
	4	AragonDB ³	multi-model	multiple	AQL	4	✓	dual-licensed	-
	5	Apache Jena ⁴	RDF triplestore	Java	SPARQL, OWL	>30	✓	Apache License 2.0	- ([Dom+10])
	6	Couchbase ⁵ (Membase)	multi-model	C, C++	NiQL	>30	✗	dual-license	-
Proprietary	7	Oracle Spatial ⁶	triplestore	Java	SQL & schema func.	>30	(✓)	closed	-
	8	AllegroGraph	triple-/documentstore	CL	SPARQL	11	✓	closed	-
	9	AnzoGraph ⁷	triplestore	C, C++	SPARQL; Cypher	22	✓	closed	✓
	10	Sparksee ⁸ (DEX)	property graph	C++	API only	24	✓	Dual-licensed	(✓) ^P ([Dom+10])
	11	GraphDB ⁹	triplestore	Java	SPARQL	10	✓	closed/dual	(✓) ^P

Table 2.1: Overview of existing graph databases, their properties, and the used query languages; Comments: ^RRanking obtained from [sol19]^P payed subscription required ¹open specification of openCypher exists [Neo15]²various backends usable (Cassandra, HBase, Bigtable, and others); backend impacts scalability ³key/value, document, graph; AragonDB Query Language ⁴various internal reasoners ⁵document oriented; community edition without recent bug fixes with Apache 2.0 license; NiQL is a SQL extension targeting JSON data⁶tailored to geographic and location; ACID compliance unclear ⁷no community edition; actual storage type not disclosed ⁸community version limited to 1 million nodes ⁹W3C compliant; free version: no more than two queries in parallel

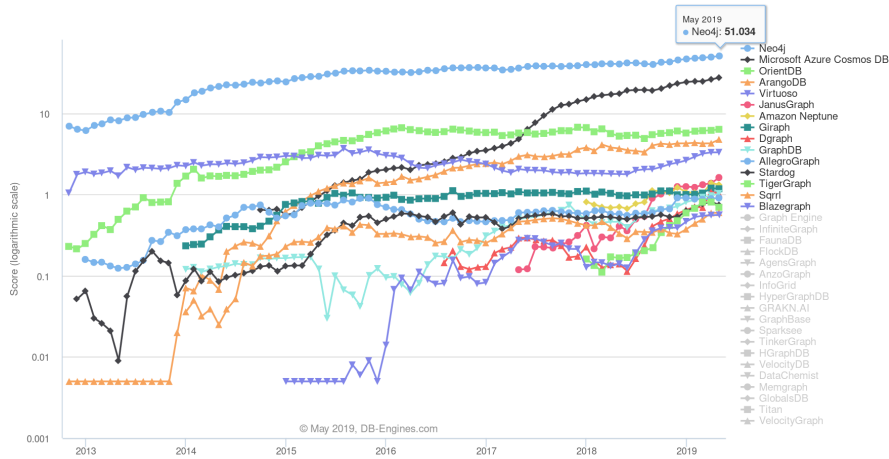


Figure 2.2: Trends of *GDB* popularity as presented by [sol19].

The individual graph database rankings (cf. Figure 2.2) are obtained from *DB-Engines Ranking* [sol19] and while they do not reflect a formally complete analysis of actual usage of the databases, they allow to identify trends and serve as an indication for current usage statistics, which are rarely officially reported by the developers themselves. According to the authors, rankings represent a popularity score including (for a given *DBMS*) the number of mentions on websites, a Google Trends³ analysis, the frequency of technical discussions on common related support forums, mentions in job offers and profiles of professionals, and lastly relevance in social networks⁴. Within this *GDB* ranking the databases Neo4j, OrientDB, and ArangoDB are placed amongst the top five candidates. The popularity and ranking of the Neo4j database has been consistently the highest among all graph databases since 2013. Within recent years cloud based alternatives (e.g. Microsoft Azure Cosmos DB or Amazon Neptune) have gained popularity, possibly due to the stronger adoption of NoSQL based databases within professional fields.

The overall applicability of *GDBs* to my work is a superset of individual properties of each platform; various features impact the applicability metric decision, including a) possible graph operations, b) supported graph size, c) storage and retrieval efficiency, d) horizontal scalability (i.e. cluster size), e) vertical scalability (i.e. number of cores), f) data ingestion time, g) indexing performance, and h) job execution time. Evaluating these metrics for the listed *GDB* is done with respect to the *EISE* application domain. In terms of scalability, Dominguez-Sal et al. conducted the HPC Scalable Graph Analysis Benchmark [Bad+09] which consists of multiple analysis techniques (via multiple kernels), which implement access to *GDB* [Dom+10].

³ <https://trends.google.com/trends/>

⁴ See https://db-engines.com/en/ranking_definition for further details on the score calculation.

Their experimentation identified that for small graphs all tested databases are capable to achieve a reasonable performance, but only Neo4j and Sparksee were able to deal with the largest benchmark sizes and are thus considered the most efficient. Guo et al. also evaluate *GDBs* using multiple different experiments [Guo+14]. The authors measure basic performance via job execution time, which shows that there is no overall winner (with Hadoop being the worst performer in all cases) and the individual *GDBs* perform stable with maximal variances of 10%. Further scalability experiments show that an increase of computation cores can lead to regressing performance, especially for small graphs while normalized performance per computing unit generally decreases when scaling horizontally and vertically. The last metric of overhead evaluation exhibits diverse results across the *GDBs*, chosen algorithms, and graphs. There exist many more evaluations of graph databases available which compare platforms directly or investigate the difference between relational based *DBMS* and *GDB* [McC+14; Vic+10; Mpi+15; Lou+15; TB11]. However, with the fast development life-cycle many tools have often improved previous shortcomings and extended further support for previously missing features (e.g. user management within Neo4j).

Summarizing, I conclude that Neo4j is the optimal choice as a platform for the development of a model-driven query support environment. With a *labeled property multidigraph* as the underlying graph model, Neo4j provides free access to a popular and high-performance solution. Even though Neo4j has no native *sharding* support to battle potential horizontal scalability, this missing feature is of low importance, as the *EISE* use-case primarily requires a high efficiency solution rather than high horizontal scalability. Additionally, Neo4j uses Cypher as its main query interface, which further backs it as an optimal platform due to my decision to use Cypher as the underlying query language for my approach (cf. Section 2.3.2). This ensures that a potential transfer or extension to other *GDB* platforms which support the openCypher standardization will be possible with little to no overhead. Neo4j's plug-in support also allows further customization of the overall database, query evaluation mechanisms, and the query access (including graph access via a Java *Object Graph Mapping (OGM)* interface).

2.3.2 Graph query languages

With possible applications of *GDBs* in various domains (e.g. biology, chemistry, machine learning, and robotics), the interest has strongly increased within the last decade showing their importance to many fields. With this development, appropriate query interfaces are required to facilitate data, knowledge, and information handling. These *graph query languages (GQLs)* are commonly external *domain-specific*

languages (DSLs) for information retrieval. Existing languages such as *SQL* (used for relational *DBMS*) can not sufficiently provide access to data stored in graph models and is unable to make use of algorithms which exploit the graph properties. As a result, new language definitions and implementations emerged next to already well established *GQLs*, such as *SPARQL*, *Prolog*, *XPath*, or *XQuery*. These new languages provide support for efficient and easier access to data in extended graph models (such as the *labeled property multidigraph*, cf. [Section 2.2](#)) and further focus on individual (niche) domain requirements; examples are *Cypher*, *Gremlin*, *GraphQL*, *G-Core*, and *PGQL*

All graph query languages share the conceptual core that they utilize either a) graph pattern matching and/or b) graph navigation as operations for graph querying [[Ang+17](#)]. The pattern matching method matches a user supplied subgraph pattern against the present graph in the *GDB* and returns all matches. The latter method alternatively navigates the topology of the graph and extracts suitable results based on the provided user query. Additionally, navigational queries allow users to check path existence or path lengths and hence directly exploit the graph model structure. In both cases, a query on a graph can return either constants (i.e. labels, types, properties, or values), nodes, relationships, individual paths, or entire (sub-)graphs. Further result augmentation, filtering and match restrictions are additionally possible by appending operations such as projection, union, optional, or difference.

At the same time, different semantics for query language execution are followed (either in the *GQL* or by the executing query engine). These are categorized into the following (sub-)types [[Ang+17](#)]:

- a) Pattern matching
 - Homomorphism-based
 - Isomorphism-based
 - Simulation-based
- a) Graph navigation
 - Arbitrary path
 - Shortest path
 - No-repeated-node
 - No-repeated-edge

Query designers thus need to precisely understand the underlying semantics of the different *GQLs* and execution engines to formulate correct and effective queries. One can easily overlook (sometimes subtle) differences in different semantical interpretations and thus construct valid queries with unintended behavior (e.g. duplicated or missing results).

```

1 MATCH (lisa:Person {name: "Lisa"})-[:Friend]->()-[:Friend]->(fof)
2 WHERE fof.name = lisa.name
3 RETURN lisa.name, fof.name

```

Listing 2.1: Example of a simple Cypher query extracting the pair of names of persons who know each other via exactly two other friends and share the same name.

Discussions in the literature about graph query languages and their strengths and weaknesses are primarily informal and present the most prominent features of different languages [Mah17]. However, more in-depth discussion of this topic are gaining traction in recent years. Angles et al. recently composed a detailed survey on the foundational features of modern graph query languages with a focus on the importance of their formalization [Ang+17]. They focus their efforts on the three representative languages *SPARQL*, Cypher and Gremlin and address them along the three dimensions of 1) data models to encode data, 2) (sub) graph patterns search, and 3) navigational expressions for path matching. Similarly, Shinavier and Wisnesky recently presented a detailed analysis of the graph model and include semantic descriptions of model transformations and query algorithms [SW19].

A *GQL* for my application domain needs to fulfill three requirements. The query language needs to provide both types of language querying – pattern matching and navigational querying. At the same time, *behavior developers* (cf. Section 4.2.2 on page 57) of the *EISE* domain who implement queries need to have an easy entry and a good grasp of the language capabilities. With most languages being *GDB* specific (cf. Table 2.1 on page 20), only few candidates remain which are transferable and support more than one storage back-end. Further, language customizations (via language internal keywords, functions, or operations) are ideal. More experienced developers can use these features to customize and extend their queries and reduce complexity. In the following, I describe Cypher, *SPARQL*, and Gremlin in greater detail alongside uncomplicated query examples as the representatives of the most appropriate *GQLs* and identify their applicability for my scenario.

2.3.2.1 Cypher

The Cypher graph query language [Fra+18; Neo11] has initially been introduced by the Neo4j, Inc. as a part of the graph database *Neo4j* [Neo07] and was extracted to its own project called openCypher in 2015 [Neo15]. Currently, Cypher within Neo4j is a specialization of the specification of openCypher, extending it with only few features with future plans to migrate to the pure openCypher specification. Cypher is a declarative *GQL* designed for graph data retrieval and storage whilst maintaining a high expressiveness and efficiency.

```

1 PREFIX foaf: <http://xmlns.com/foaf/0.1/>
2 SELECT ?name ?mbox
3 WHERE
4 { ?x foaf:name ?name .
5   ?x foaf:mbox ?mbox }

```

Listing 2.2: Example of a simple *SPARQL* query extracting the name and email address of persons.

The language design is strongly inspired by the popular SQL (Structured Query Language) query language commonly used in relational *DBMSs*; Listing 2.1 on the facing page shows a query example matching a subgraph and filtering the results. This similarity to *SQL* shows in the clause syntax which is using a subgraph pattern matching clause (*MATCH*), a filtering clause (*WHERE*) and a final transformation clause (*RETURN*). Additional clauses exist for the creation, modification, and deletion of nodes, relationships, and properties. Cypher's *concrete syntax* visually encodes nodes as circles with surrounding braces and relationships between them as boxes with directed arrows. Additionally, the properties of locally bound variables can be accessed via common dot notation. Build-in functions allow to execute common operations, such as *ID()* and *TYPE()* to obtain the node id and type respectively, *COUNT()* to count elements, or various mathematical operations⁵. Cypher is based on the *labeled property multidigraph* model (cf. Section 2.2 on page 14) and its evaluation follows the isomorphism-based non-repeated edge semantics: the same edge is not mapped twice within a single match statement [Ang+17]. This property ensures for example that the query in Listing 2.1 on the facing page does match a circular graph between the same two nodes.

In summary, Cypher is a suitable candidate as the query language for queries towards the *EISE* domain. The language holds a) an intuitive interface close to existing query languages, b) allows to be extended and customized via user functions, c) uses the extensive *labeled property multidigraph* model with intuitive isomorphism-based non-repeated edge semantics, and d) the Neo4j back-end is a strong and scalable competitor pushing adoption further via the openCypher initiative.

2.3.2.2 SPARQL

The *SPARQL* has initially been standardized in 2008 by the *World Wide Web Consortium (W3C)* and is considered as a key semantic web technology [W3Co8]. It is a declarative semantic triplet query language intended for the design of retrieval and storage queries of *RDF* formatted data [Hebo9]. The language supports query triplet patterns,

⁵ Refer to <https://neo4j.com/docs/cypher-refcard/current/> for the full reference of the current Cypher features shipped with Neo4j

conjunctions, disjunctions, and optional patterns⁶ (cf. [Listing 2.2](#) on the previous page for a simple *SPARQL* query example). The core clause syntax of *SPARQL* allows to locally bind external namespaces (PREFIX), a matching clause used to identify individual elements or graph patterns to be returned (SELECT), a filter clause holding the triplets representing the graph pattern to match (WHERE), and further optional elements to refine the query (e.g. UNION, CONSTRUCT, or ASK). It is a rich and expressive querying language allowing users to write complex and extensive queries. In contrast to the Cypher language, the semantics of *SPARQL* evaluation are homomorphism-based, matching of identical nodes within a graph needs to be manually avoided with appropriate filters [[Ang+17](#)].

The application of *SPARQL* within the *EISE* domain is a possible choice. The language provides extension mechanisms and is highly expressive. However, the expressiveness also implicates high costs of efficient query design. Further, the description of subgraphs in *SPARQL* via triple patterns is perceived as not as intuitive as in other languages such as Cypher [[SW19](#)]. The central backends with direct *SPARQL* support are RDF knowledge bases due to the underlying knowledge base structure, opening potential issues of transferability and user adoption.

2.3.2.3 Gremlin

Gremlin is a property graph query language introduced in 2016 and is a central element of the *Apache TinkerPop3* graph framework [[Rod15b](#)] (cf. [Listing 2.3](#) on the facing page). Unlike Cypher and *SPARQL*, it does not draw inspiration from *SQL*, but rather functional query languages, such as XPath [[RDS17](#)]. As a result, Gremlin is also a functional language that focuses on navigational over pattern matching based queries. However, is still possible to define subgraphs for matching queries using declarative queries. Gremlin evaluation follows the homomorphism based bag semantics [[Ang+17](#)]. With the functional language design, Gremlin decisively differentiates as it is an embedded language in any host language, which supports function composition and nesting. This results in queries being written alongside the application code and thus tightly integrated into the application logic. The exemplary Gremlin query shown in [Listing 2.3](#) on the next page operates on `G.V()` which provides the set of all nodes (called vertexes within Gremlin) in the graph. Instead of individual clauses as implemented in Cypher and *SPARQL*, queries in the Gremlin language traverses the graph by chaining individual functions (`hasLabel()` to match nodes, `out()` to follow relations). Result filtering happens along this traversal via corresponding functions

⁶ Refer to <https://www.w3.org/TR/2006/WD-rdf-sparql-query-20061004/> and <https://www.w3.org/TR/rdf-sparql-query/> for a full list of features in *SPARQL*

```

1 G.V().hasLabel("Person").has("name","Lisa")
  ↪ .out("friend").hasLabel("Friend")
  ↪ .out("friend").hasLabel("Friend").has("name","Lisa")
  ↪ .values("name","email")

```

Listing 2.3: Example of a simple Gremlin query extracting a traversal over three nodes extracting the name and email address of persons.

(.has()). Possible results are iterable traversals, nodes, paths or extractions from the graph (.values()).

With no inspiration from popular languages such as *SQL*, the Gremlin language requires users to get to know the language's traversal based approach in depth to effectively formulate queries. The tight integration of queries in application code further results in difficulties for code generation within *MDSE* approaches. Gremlin is primarily used within *JanusGraph* and all current efforts to standardize graph query languages do not include any functional query language [ISO19]. Its transferability is consequently unclear and an application for the *EISE* domain thus not ideal.

2.3.2.4 Other

Besides Cypher, two other state-of-the-art property graph query languages exist, namely the research language proposal *G-CORE* and the *Property Graph Query Language (PGQL)*. While *PGQL* is bundled and bound to the *Oracle Spatial and Graph database*, *G-CORE* represents a research language proposal. With largely overlapping feature sets in these three languages, the *GQL Manifesto* has been established to call for a fuse of these three languages into comprehensive query language for graph data called "GQL", to fill a role similar as *SQL* in the context of relational databases [Neo19]. The applicability of each of these language for my work is thus similar and decided based on their interoperability with graph storages. Only recently the call for a unified graph query language is gaining further traction. The ISO/IEC's Joint Technical Committee decided to work on a unified *Graph Query Language (GQL)*⁷ [ISO19]⁸. Cypher is the core inspiration for the new developments but also other languages proposed in the *GQL Manifesto* influence the upcoming development.

Besides these generalizing query languages for (property) graphs, query languages for niche applications and highly specialized languages are available, for example, *GraphQL*, *NIQL*, or *AQL*. While *NIQL* and *AQL* are domain-specific specializations of existing languages, *GraphQL* provides a conceptual framework as an alternative to REST frameworks [Fac16; HP18]. It is an open-source data manipu-

⁷ Note that the new *GQL* language name collides with the already existing query language but is favored due to its closeness to the standardized *SQL*.

⁸ See <https://neo4j.com/blog/gql-standard-query-language-property-graphs/> for further information

lation language for *Application programming interfaces (APIs)* released in 2015 and now part of the *GraphQL Foundation*. Since 2018, the *GraphQL Schema Definition Language* was added to the specification adding schema and type support to the framework. In contrast to the previously mentioned query languages, *GraphQL* requires an existing code and data environment for its application as the language is not part of a *GDB* and does not provide any database or storage engine itself⁹). Semantics of *GraphQL* are only implicit and subject to current research with the implementation and application relying on an already present back-end environment [HP18]. The query language design is inspired by the *JSON* syntax and query results returned are *JSON* objects. It is different to usual *GQLs* and the application is intended for the goal to provide a framework for web based services. With these properties, *GraphQL* is thus not a valid choice for the application in the *EISE* domain.

2.4 SUMMARY

This chapter presents the context of graph-based knowledge representation alongside the *DIKW* pyramid model. With such a prominent role of graph structures I present the *labeled property multidigraph* model which is the most versatile data representation model. It allows to represent graphs with nodes and relationships which both hold multiple labels, properties, and values. Additionally, the graph is a directed graph that can represent multiple relationships of the same type between individual nodes. The increased adoption of *NoSQL* databases also shows an increased use of *GDBs*. Consequently, I present an introduction and analysis of available *GDBs* and *GQLs*. The analysis shows that the usage of Neo4j as a *GDB* in combination with Cypher provides a viable technology foundation to implement upon. Neo4j is a popular *GDB* choice which scales appropriately and allows for unrestricted free access. Cypher provides pattern based matching and path queries while providing high language familiarity due to its closeness to *SQL*. Additionally, the non-repeated edge bag semantics reduce the query design complexity further. With the openCypher initiative and recent standardization efforts, the language also prospectively will be able to provide maximal storage independence [Neo19; ISO19].

⁹ With only experimental support in *GDBs*, for example within Neo4j: <https://neo4j.com/developer/graphql/>


“When I first started using the phrase software engineering, it was considered to be quite amusing. They used to kid me about my radical ideas. Software eventually and necessarily gained the same respect as any other discipline.”

—Margaret Hamilton
Mathematician and Pioneering Computer Scientist,
co-authored the notion of *software engineering*

As I apply a *Model-driven Software Engineering (MDSE)* approach in this work, this chapter provides an overview to the relevant fundamental concepts by introducing general definitions of core keywords as a common ground. Within the relevant subdomains there exist numerous interpretations of the relevant definitions and their application in a *MDSE* approach [Völ+13; Rod15a]. I will summarize these points of views and present their usage in relevant recent literature of the individual (sub-)domains. Lastly, based on these foundations, [Section 3.3](#) on page 44 presents the development process applied in this thesis to develop a model-driven graph query interface for interactive smart environments.

3.1 FOUNDATIONS AND INTRODUCTION

3.1.1 *Models and transformations*

At the core of *MDSE* (also called *Model-driven Engineering (MDE)*) stands the task of abstraction from a certain domain and its entities by creating a *models*. According to Bevin and Gerbe [BG01] a *model*  *model* is defined as follows:

A model is a simplification of a system built with an intended goal in mind [...]. The model should be able to answer questions in place of the actual system. The answers provided by the model should be the same as those given by the system itself, on the condition that questions are within the domain defined by the general goal of the system. [BG01, p. 2]

Similarly, Combemale [Com17] define the term *model* as follows:

A model is an abstraction of an aspect of reality (as-is or to-be) that is built for a given purpose. [Com17, p. 5]

Thus, a *model* is an abstraction of a system and gains its usefulness from being easier to use than the original, which is visualized in [Figure 3.1](#) on the next page, where layer *Mo* represents the real-world system and layer *M1* represents the *model* of this system [Omgo8]. While this definition of a *model* considers the corresponding core concepts (i.e., a targeted simplification of a system allowing to answer questions towards the system itself), it is rather broad and allows for a wide interpretation of what can be a *model*. More recently, Rodrigues da Silva [Rod15a] summarizes the definition of a *model* as:

[A] model [is] a system that helps to define and to give answers of the system under study without the need to consider it directly. [Rod15a, p. 141]

However, a more concise definition (and more relevant for *MDSE*) is given by Kleppe et al. [KWBo3]:

A model is a description of a (part of) systems written in a well-defined language. A well-defined language is a language with well-defined form (syntax), and meaning (semantics), which is suitable for automated interpretation by a computer [KWBo3, p. 52]

This definition includes languages – along with the important property of being well-defined (i.e., in the mathematical sense to be unique and unambiguous to allow automated interpretation) – with both of their central parts: the syntax, referred to as the *concrete syntax*, and most importantly the semantics, referred to as the *abstract syntax*. This definition of a *model* thus describes *prescriptive models* (i.e., more rigorous formal, complete, and consistent), rather than solely *descriptive models* [Völ13a]. In the context of software engineering this difference is key, because this enables one to use a defined *model* and transform it as required within the same medium (i.e., a digital representation of the system). Depending on anticipated final target *artifacts*, an example result can be generated *General Purpose Language (GPL)* source code to be executed in production.

Re-applying the step of modeling to modeling itself results in the corresponding *meta-model* (see [Figure 3.1](#) on the facing page, layer *M1*), which is composed of the concepts required to write down the *model* itself. *Meta-models* in the *M2* layer are thus the languages created in the *MDSE* process with the goal to be used by the domain experts in the *M1* layer to model the real-world system. In my work I will focus on the layers *Mo* to *M2*, but for completeness it is important to note that re-applying the modeling step again on the *M2* layer provides one with a *meta-meta-model* (*M3*) which describes the concepts required to write down a *meta-model*. As a result, the real-world system is an instance of a *model* which conforms to the *meta-model* which in turn conforms to the *meta-meta-model*.

concrete syntax ☞
abstract syntax ☞

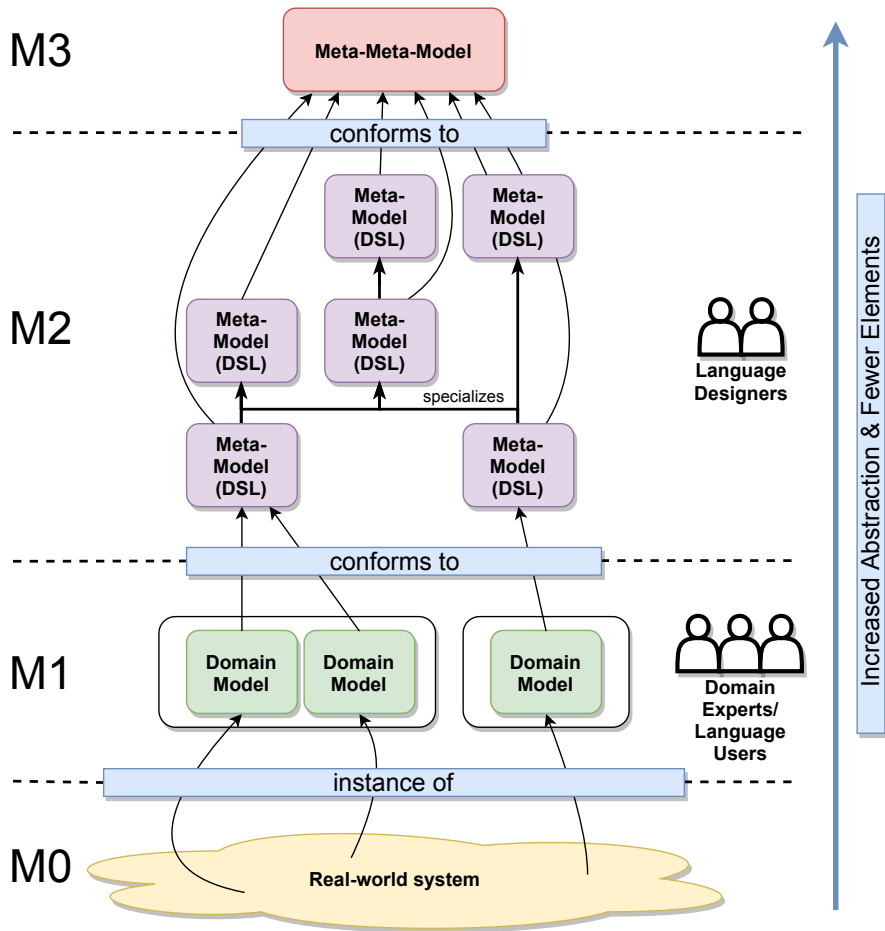


Figure 3.1: The four layered abstraction model (*Meta-Object Facility (MOF)*) as defined by the *Object Management Group (OMG)* along with the language user and language designer areas of involvement [Omgo8].

3.1.2 Domain-specific languages

The previously mentioned language – a central element of *MDSE* – is referred to as a *domain-specific language (DSL)*. van Deursen et al. [vKV00] define *DSLs* as: ✍ *domain-specific language (DSL)*

A domain-specific language is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain. [vKV00, p. 1]

More recently and prominently Fowler [Fow10] defined:

domain-specific language (noun): a computer programming language of limited expressiveness focused on a particular domain. [Fow10, p. 33]

The following sections describe *DSLs* and their effect in further detail.

3.1.2.1 Benefits and challenges of DSLs

With the reduced expressiveness of a new language, *DSLs* offer a wide range of benefits [vKV00; Völ13a]. Most importantly, the expression of a given problem at the level of abstraction of the problem domain is a strong argument. This has a positive impact on software production and will increase productivity, as problems are expressed and solved quicker with the actual language of the domain. To identify and realize these concepts in a new language, direct involvement of domain experts is necessary and – in turn – will assist the communication about the problem and domain between the involved parties. For each language concept individual (provably) correct *artifacts* can be generated which allows to proof the overall correctness and thus increase the product quality. This generation step can also individually be changed or extended to any given target platform, ensuring reuse and portability. With the *model* knowledge about concepts and relationships of the domain, the corresponding validation and verification of *models* becomes available. This feature increases the testability as it allows for validation and verification of user inputs at *model* design time, rather than at later stages of the development (e.g. at compile time). Potential input errors are avoided which increases the reliability of the product. With *models* specifically describing their domain, their maintenance and evolution is straight forward and thus reliability, maintainability, and data longevity increases.

The use of a *DSL* also comes with challenges, which developers need to overcome [vKV00; Völ13a]. *MDSE* processes require the necessary language engineering skills to reach high quality *DSLs* and results in at first increased development costs (i.e. higher effort for design and implementation). The target users of *DSLs* are also required to understand and effectively use the created languages, which can be mitigated by involving them early in the development cycle. This involvement can help to also reduce the difficulty of outweighing the ideal level of specificness and keeping a balanced *DSL* scope. Further, the integration into other tools needs to be addressed to avoid the general danger of (tool) lock-in. Thus, current research attends reusable language (domain-specific) building blocks for common *DSL* tools as well as language modularization [Wig+17; Völ18]. Lastly, a specific issue is described by Völter called the “DSL Hell” [Völ13a, p. 44], which describes the issue of re-implementation and development of unfinished languages instead of using existing compatible, well engineered and extendable languages.

3.1.2.2 DSL variants

DSLs can be divided into two different variants [Völ13a; FBoo].

First, internal (or embedded) *DSLs* are written within the language which they are intended to be used with/in. In this case, the host

language itself is transformed and extended into a *DSL*. As a result, the full feature set of the host language (often also called the base language) is available to the language creator and (if wanted) to the language end-user. However, this can also impose limitations on the internal *DSL* as any reduced expressiveness of the host language will also be present in the final language. With the overall goal of *DSLs* in mind – to reduce the expressiveness for the target audience of domain experts – language creators thus must reduce available features so that there are unique ways to express problems of the domain at hand. This is especially difficult as the host language is not necessarily designed to be used for an embedded *DSL*. Without appropriate abstraction, non-programmers will potentially struggle to use a finished language if they do lack basic knowledge of the host language itself. Prominent examples for this approach are often created within functional programming languages (e.g. Lisp).

Second, external *DSLs* are written in a different language than the targeted host language. This removes the burdens that come with an internal host language (e.g. the host syntax, or features) and allows to define any free form for the target *DSL*. External *DSLs* thus lack a link into the target language; this mapping must consequently be defined by the language designer. Further, tools and features that commonly are provided for host languages and are considered the bare minimum for a language (e.g. an editor, compiler, or fully featured *integrated development environment (IDE)*) have to be created and provided for external *DSLs*. This requires the language developer to find appropriate abstractions and limits for their external *DSL*. Lastly, a new external language needs to be transformed into the target language with the help of defined *model-to-model (M2M)* and/or *model-to-text (M2T)* generators.


Summed up, from the perspective of the language end-user external languages mitigate issues internal languages often struggle with. But in turn the *DSL* creator will have a more demanding task as detailed care of has to be put into language design.

3.1.2.3 *DSL semantics*

Next to the *abstract syntax* and the *concrete syntax*, it is necessary to describe the behavior of a *DSL* – the language *semantics* [Com17; Hen90; NN91]. The syntax is initially defined either by using a minimized notation representation such as *Extended Backus–Naur Form (EBNF)* or by graphical representations such as *meta-models*. While the syntax describes the valid and allowed form of expressions within a language, the semantics are concerned with the effects of the evaluation of correct expressions [HR00]. The semantics fulfill the three roles of 1) a basis to prove the semantical correctness of a language, 2) a machine- and compiler-independent standard, and 3) a formal way to ensure that implementation are correct and conform to the concepts created

by the language designers. There are two distinct types of semantics for programming languages [Com17; Hen90; NN91]: a) operational semantics, and b) denotational semantics.

Operational semantics provide logical statements which server to prove a language's execution and procedures. Common categories among the *operational semantics* are a) *concrete operational semantics*: This practical approach calculates the values of expressions of the language via a compiler or interpreter for the language. It requires a full implementation of the behavior that represents the language semantics and can be used to obtain the meaning of any given expression. b) *small-step semantics* or *evaluation semantics*: Language developers create formal axiomatizations of the intentions of expression evaluation. The individual inductive definitions are the resulting formalization which can lead to large sets of non-trivial axioms. c) *big-step semantics* or *computation semantics*: Result oriented semantics that show the consequence and overall results of an expression.

denotational semantics 

The *denotational semantics* in essence use mathematics for semantical descriptions to represent the interpretation of the language behavior. Thus the *denotational semantics* provide the space of meanings for all language expressions and association between symbols and actual functions. This type of semantics allows to provide concise descriptions and defines a semantical mapping \mathbb{M} from sets of languages \mathbb{L} to the semantic domain \mathbb{S} , i.e. $\mathbb{M} : \mathbb{L} \rightarrow \mathbb{S}$. The *denotational semantics* $\llbracket _ \rrbracket_{\mathbb{L}}$ of a language L (written as $\llbracket _ \rrbracket_{\mathbb{L}}$) thus describes its behavior by formalizing the meanings as mathematical constructs. This description is independent of the *concrete syntax* and provides a precise description of the individual language actions.

Semantics of *DSLs* in research literature are given mostly informal in the accompanying text – if provided at all. However, distinct denotations are required for the soundness of languages and unambiguously transport language semantics to users. Given a) the available freedom in their definition, b) the conciseness of the definitions, and c) the common understanding of definitions, *Denotational semantics* are the ideal application for describing language behavior in the scope of this thesis. For most languages of distinct domains (e.g. *graph query languages (GQLs)* as Cypher) the language semantics are well understood and documented extensively and thus omitted in this thesis¹. However, language combinations in this thesis (especially orthogonal language composition) are not intuitively defined and I thus provide their *denotational semantics*, following the notations as presented by Hennessy [Hen90; Com17]. In this thesis, I use the Cypher query language and its *concrete syntax* as the running example for *GQL* reduction. Cypher semantics are already described in detail and provide a graspable representation foundation for seman-

¹ The semantics are often not described formally but as a mixture of *concrete syntax*, *abstract syntax*, use-cases, implementations, and usage examples.

tic clarifications². The identity semantics $\llbracket \cdot \rrbracket_I$ of a given query Q are thus assumed to evaluate to Cypher semantics $\llbracket \cdot \rrbracket_C$ as follows.


$$\llbracket Q \rrbracket = \llbracket Q \rrbracket_I \quad (3.1)$$

$$\llbracket Q \rrbracket_I = \llbracket Q \rrbracket_C \quad (3.2)$$

3.1.2.4 Language composition

Language composition and suitable modularization has been identified as a central necessity for *DSL* development [Völ13a; Com17; Pic10; Erd+13; VP12; ŞvV18]. Multiple of the previously named advantages of *DSLs* – such as reuse, or extendability – require successful language dependency organization. In a recent literature survey by Méndez-Acuña et al. the authors additionally emphasize the importance of language modularization to reach acceptable separability [Mén+16]. They recommend to practice “Language Product Lines Engineering”, i.e. software product lines where the products are *DSLs*. Language features join the sets of language constructs to represent a functionality, which is provided by a *DSL*. Varying combinations of these features then can allow to produce a target *DSL*.

In detail, Völter categorizes five central types of modularization and composition approaches (compare Figure 3.2 [Völ13b]):

 language composition

REFERENCE This composition strategy allows to reference elements of a language L_B within another language L_A . A direct dependency is established between the two languages when at least one concept C_{A_1} from L_A references another concept C_{B_1} of L_B . However, the resulting fragments F_A and F_B of either language stay homogeneous as the reference is also represented in the fragments. Fragments are thus not combined. L_A consequently cannot be used without L_B

EXTENSION An extending composition allows the combination of concepts from different languages, for example to extend an existing language with additional features. The depicted example in Figure 3.2 on the next page shows a language L_A extending language L_B by providing C_{A_3} which is a specialization of the existing concept C_{B_3} . Concept C_{A_3} can thus be used as a child of C_{B_4} (additionally to C_{B_3}) and in turn provide (as shown in this example) an additional child C_{A_4} to a fragment *abstract syntax tree (AST)*. This mix of concepts allows heterogeneous fragments while creating a direct dependency (L_A depends on L_B).

REUSE The reuse composition allows to create homogeneous fragments while at the same time maintaining independent languages. To realize this independence between languages L_A and

² Refer to <https://neo4j.com/docs/cypher-refcard/current/> for a short summary.

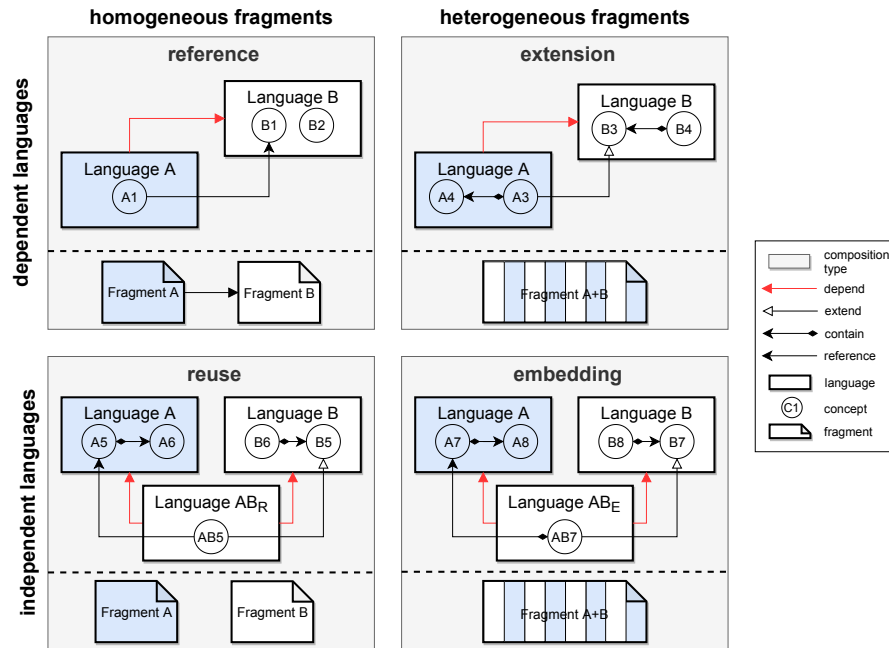


Figure 3.2: The four types of language composition and the resulting fragment structure as presented by [Völ13b, pp. 116–127].

L_B in the shown example, an adapter language L_{AB_R} is introduced. On the one hand, concept C_{AB_5} of adapter language L_{AB_R} *specializes* C_{B_5} and on the other hand, C_{AB_5} also *references* C_{A_5} . As a result, only language L_{AB_R} has a dependency to other languages while the languages L_A and L_B remain independent. This technique is very useful for *DSLs* which cover generic domains with high potential of reuse (e.g. a time domain). However, great care needs to be taken when creating reusable languages so that explicit hooks and adaption points are made available for later reuse.

EMBEDDING The embed composition is very similar to a reuse composition. Again, an adapter language L_{AB_E} is added which takes the role of holding the dependency to the involved languages L_A and L_B . These languages thus stay independent from each other. Fragments of embedding languages are, however, heterogeneous as this composition allows to mix the involved languages together. To realize this behavior, concept C_{AB_7} of the adapter language L_{AB_E} *specializes* C_{B_5} (similar to reuse). Additionally, instead of just a reference, concept C_{AB_7} *contains*³ instances of C_{A_7} (i.e. has C_{A_7} as a child). This composition strategy is especially useful when syntactically composing otherwise independent languages

³ The *Unified Modeling Language (UML)* notation defines this relation type as *compose*, but to avoid an ambiguous usage in this section, it is referred to as *contain*

ORTHOGONAL COMPOSITION Orthogonal language composition is alternatively also referred to as language *annotation*. This composition mechanism allows to attach concepts of one language to the *model AST* of another language without affecting any functionality of the target language. All following work on the *model* ignores these annotations unless they are explicitly supported by the language using the modified *AST*. Fragments are consequently heterogeneous (similar to *extension*), but the languages themselves stay independent as they are not aware of the attachment. Further no adapter language needs to be defined to handle the connection between languages. In this case, the combination of languages (being the attaching of fragments to other fragments) is a feature which the *language workbench* must explicitly support. While it is not a composition strategy in the classical sense as the others depicted in Figure 3.2 on the facing page, it allows for very clean combination of languages of different (orthogonal) domains, without introducing additional dependencies.

Similarly to Völter, Erdweg et al. also summarize comparable types of composition into the five categories of [EGR12]: a) language extension, b) language restriction, c) language unification (i.e. adapter languages), d) self-extension (i.e. embedding), and e) extension composition (i.e. incremental extension).

For example, the application of suitable composition is especially important in the context of cognitive systems and robotics research. Wigand et al. proposed an approach that contains a focus on language composition and generation for component-based robotics systems [Wig+17]. Their goal is to support extensibility and refinement of the system and split the language modules into three orthogonal dimensions: hardware platform, software platform, and capability. As a result, they successfully implemented a quad-arm object manipulation scenario on a simulated robot setup and on real robot hardware.

Look et al. presented their approach of black-box integration of heterogeneous languages in the context of cyber-physical systems [Loo+14]. Similarly to the abstract presentation by Völter, they base their approach on the three composition mechanisms provided by their *language workbench MontiCore* (a framework for the development of domain-specific languages with special consideration of language composition [KRV10]): a) aggregation, b) embedding, and c) inheritance. Unfortunately, the authors do not include language composition information or detailed *meta-model* diagrams. Also an evaluation or application to a real-world scenario would be beneficial to validate the approach.

Also using the *MontiCore* workbench, Butting et al. recently presented an approach for systematic composition of independent language features [But+19]. They use grammar-based language syntax

modules to separate concerns of language life cycle participants. Languages are decomposed into “composable language components” by the authors to realize automated language derivation. The authors argue that this step increases reuse of *abstract syntax* and tooling as the decomposition decouples language development and composition. The language composition is hence based on developer defined grammar and rules.

Another example closely related to this thesis, is the robot knowledge query *DSL* by Balint-Benczedi et al. [Bal+17]. The authors provide a language which is part of *KnowRob*'s perceptual episodic memory storage and retrieval system. The presented language is an internal *DSL* created via a grammar definitions and serves the purpose of a description language of the stored information. Their language description, however, does not contain any language composition information or other language design details.

3.1.2.5 Language workbenches

There exist numerous tools that assist language developers and facilitate the design and usage of *DSLs*. These so called *language workbenches* [Fow05] “alter the relationship between editing and compiling the program. Essentially they shift from editing text files to editing the abstract representation of the program” [Fow05, p. 12]. They provide an integrated language development environment providing the tools to edit different language aspects (e.g. *concrete syntax*, *abstract syntax*, or semantics), as well as the language *pragmatics* [Rod15a] (i.e. the practical concerns of a language, such as its application in the real-world). *Language workbenches* vary in the set of provided features and design approaches and are hence analyzed and evaluated with respect to features and completeness [Erd+15]. This section introduces an overview of state-of-the-art tools providing *language workbenches* features along with their advantages and challenges.

XTEXT

Xtext [Ecl] is an Eclipse-based open-source software framework which provides the means to develop *DSLs* (c.f. Figure 3.3 on the next page). Development of *Xtext* takes place within the *Eclipse Modeling Framework (EMF)* and provides *Ecore* as the central (meta-)model, which in turn implements the *OMG's MOF* shown in Figure 3.1 on page 31. *Xtext* is based on a notation close to *EBNF* and the framework provides features for meta modeling, constraint checking, code generation and *M2M* generation [EV06]. The *AST model* is derived (parsed) from the textual syntax and can support multiple *concrete syntaxes* by manually defining additional representations once the textual syntax was parsed. The internal project structure within Eclipse is used for *Xtext* languages and individual parts of the language is separated into individual projects.

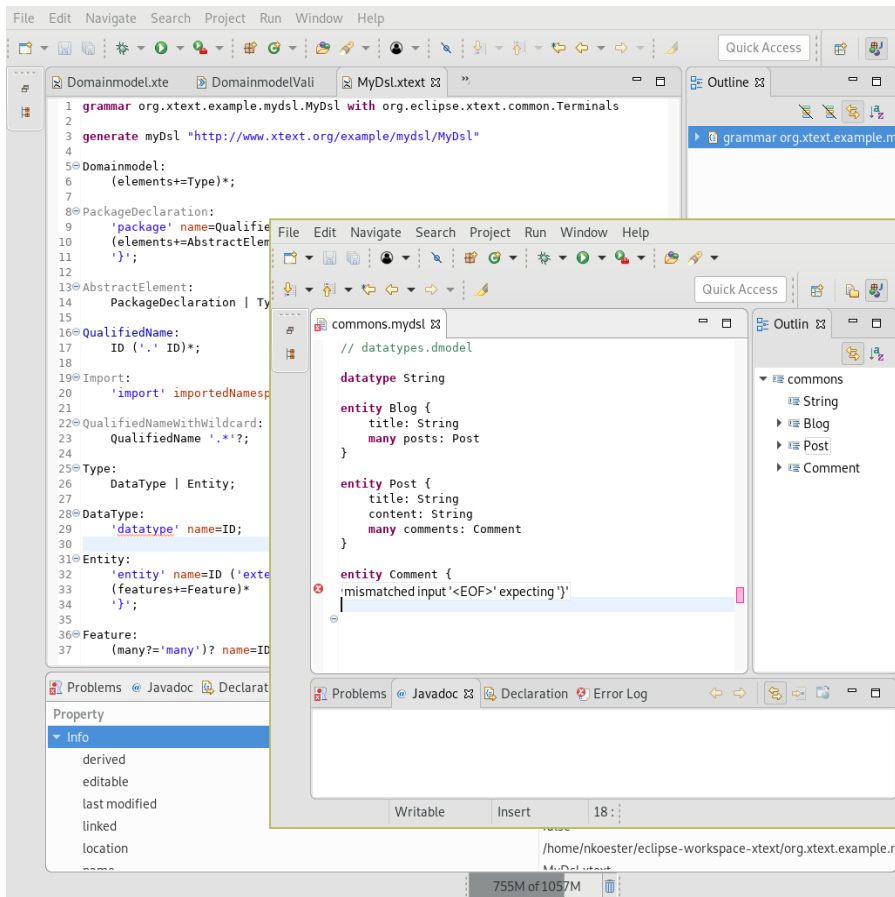


Figure 3.3: Exemplary screen shot of a running *Xtext* instance with a domain model example language. The created *DSL* is launched as a new Eclipse instance (window on top) in which the newly created concepts are available.

Developers can use the general-purpose high-level programming language *Xtend* as the common grounding language, which is implemented using the *Xtext* framework [Typ11]. The implementation aims to provide a less verbose language which is syntactically close to Java while maintaining maximum compatibility. The core usage of *Xtext* requires language developers to also use other plug-ins provided by the *openArchitectureWare* project, which in turn is a part of the Eclipse *Generative Modeling Technologies* project. Consequently, the integration of all parts and plug-ins required for effective domain modeling with *Xtext* is not a seamless experience which can in turn also impact the language end users.

JETBRAINS META PROGRAMMING SYSTEM

JetBrains Meta Programming System (MPS) [Jet; Jet18] is an open-source tool developed by JetBrains. It facilitates the creation of external *DSLs* and in contrast to most other parser based *language workbenches*, *MPS* provides a projectional editing environment. The language *concrete*

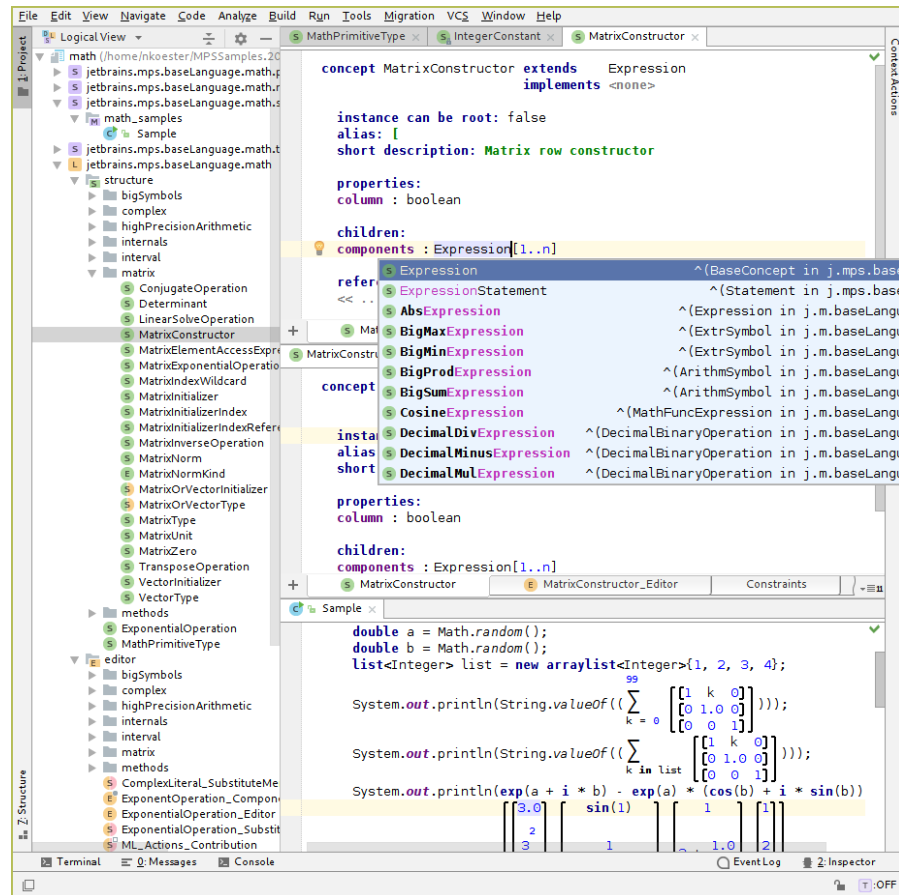


Figure 3.4: Exemplary screen shot of a running *MPS* instance. The shown math language is embedded into the provided Java base language and allows seamless editing of Java code and common math constructs.

syntax is therefore separated from the *abstract syntax* and thus the *AST*, consequently removing the need of limited language parsing [VP12]. As a result, the information of a *model* can be displayed in different formats such as textual, tables or within other custom graphical representations.

Language definition in *MPS* is separated into nine different language aspects for each language concept: structure, editor, actions, constraints, behavior, typesystem, intentions, dataflow, and generator. These individual aspects represent the central elements required for the definition of a *DSL*. The structure models the *abstract syntax*; the editor model the *concrete syntax*; action and behavior aspects model a concepts behavior; constraints, typesystem, and dataflow model various restrictions of the concepts; and lastly the generator models the necessary *M2M* and *M2T* transformations via a template engine, which are required for *artifact* generation. Each aspect can be individually changed for each concept to implement the necessary language features.

Further, *MPS* also includes numerous user-facing features which modern *IDEs* provide, for example, syntax highlighting, code completion, error checking, or runtime debugging [Völ13b]. This also includes extensive language modularization capabilities facilitating language composition for language combination, extension, reuse, embedding, and orthogonal language composition. With extendability, maintainability, and low coupling as goals, these features are essential for large *MDSE* projects.

Unlike the previously presented *Xtext*, JetBrains provides core concepts and a base language which serve the purpose of providing foundational and extendable concepts. The *MPS* base language is referred to as the *Java Base Language* and implements the complete Java API as *DSL* constructs. This language overcomes the lack of a link into a target language with which external *DSLs* often struggle with. As a result, this allows easy integration of other languages within Java programming code, as shown in Figure 3.4 on the preceding page. In this example math language concepts are directly embedded and accessible within Java code statements. The math language generator defines the *M2M* transformations required to transform its concepts into base language concepts, which are then generated in a later generation step into valid Java statements using the *M2T* generators of the base language.

ANTLR

ANother Tool for Language Recognition (ANTLR) is a tool used to create parser based languages [Ter]. Parsers and lexers are generated based on a given grammar specifying the language expressed in *EBNF*. Generated tools are used for reading, processing, executing, or translating structured text or binary files. The supported target languages for generation contain most prominent *GPLs* such as Java, C++ and Python. As a primarily text based approach using parsing, *ANTLR* provides no capabilities for language composition or management. Though *ANTLR* allows to create external *DSLs*, the *AST* is represented by *EBNF* definitions and language *pragmatics* and further tooling needs to be build around the generated parsers and lexers.

OTHER LANGUAGE WORKBENCHES Further, there exist a wide variety of other *language workbenches*, for example Melange & Kermeta: A language composition oriented approach that provides a bridge towards the Eclipse ECore formalism as the underlying *meta-model* conforms to the *MOF* standard⁴, Metaedit: A commercial modeling tool used in industry applications focusing on graphical language creation ⁵, Spoofox: A workbench allowing to generate parsers, type checkers, compilers, and plug-ins for common *IDEs*⁶. For detailed de-

⁴ <http://www.kermeta.org/>

⁵ <http://www.metacase.com/>

⁶ <http://www.metaborg.org>

scriptions and information and analysis of the existing workbenches, I refer to the *language workbench* evaluation presented by Erdweg et al. [Erd+15].

3.1.3 Benefits of MDSE

MDSE has proved to be an effective approach in the development and maintenance of large scale and embedded systems [Hut+11; Lie+14a; Rod15a]. Empirical assessment in industry shows that most apparent benefits of *MDSE* are the increased communication and reduced time to respond to quickly changing surroundings. Models are used for all parts of the development cycle, such as domain modeling, documentation, refactoring, transformation, static analysis, code generation, or automated testing.

In contrast to traditional software engineering the *model* creation process holds special properties. At the core, *models* and their counterpart are in the same eco-system: they are both software and as such automatic processing of the *models* is possible. The formalization in the modeling process documents the structure of valid *models* via the involved aspects *abstract syntax*, *concrete syntax*, semantics, and *pragmatics*.

The *MDSE* process is practical and a good addition to the software engineering practices already in place as modeling is a common task in computer science. Nevertheless, formalizing modeling is a task beyond creating class diagrams which requires an initial investment concealing a lot of *MDSE* success. When overcoming these challenges, *MDSE* can yield strong advantages over traditional approaches [Völ13a; Rod15a]. Due to formalization the developers reach an (implementation) independent *meta-model* of the domain which reduces the semantic gap between original and actual implementation. This *meta-model* allows to execute analysis and checking on *models* while programming/writing statements in the created language. Developers are provided helpful functionality, such as static analysis, domain-specific code completion, debugging at design time, or most importantly, the ability to generate code automatically based on the created user *models*. In turn, these features positively impact productivity, quality, validation, and verification.

For the application in the *Embodied Interaction in Smart Environments (EISE)* domain (cf. Chapter 4) a *MDSE* approach provides productive tooling to support developers in the query design. The accompanying advantages result in direct feedback for developers at query design time rather than at execution time. The query creation process can further be stripped from complexity, for example by providing model-based completion suggestions or special features for temporally constraint queries. The developers are thus closer to the problem domain allowing to specify queries easier, understand, and maintain during

system evolution – thus acting as domain experts rather than broad system and technology specialists.

3.2 APPLICATION OF MDSE IN ADJACENT DOMAINS

The combination and incorporation of graphs in *MDSE* approaches is often highly application domain-specific.

For example, direct applications such as *Green-Marl* provide high-level languages that provide features for algorithms on graphs and graph structure analysis [Hon+12]. Hong et al. present an external *DSL* which is targeted for developers that is capable to generate C++ code. The basis for the proposed analysis language is a directed property graph which is not modified during the execution of developer defined analysis. The provided approach includes a compiler for parsing, type checking, and *model* transformation. Though the authors do not provide *meta-models* or any other *abstract syntax* definition, the presented evaluation shows reduced (*Source*) *Lines Of Code (LOC)* and reduced algorithmic execution duration.

Similarly, *GraphIT* is a performance oriented graph *DSL* for algorithmic applications on graphs [Zha+18]. Its scope is close to *Green-Marl* as the implemented *DSL* is a high-level language describing computations on graphs. Optimized algorithm implementations are generated by the proposed compiler, focusing on performance characteristics. The approach separates computation of the algorithm from how it is computed via an algorithm language for programmers and a second scheduling language for performance optimizations. The authors also do not describe the *abstract syntax* of the languages and the behavior of the languages are explained solely example driven with no concise semantics. However, a detailed quantitative evaluation is presented comparing similar state-of-the-art frameworks and *DSLs* on graphs on multiple datasets showing its increased performance over the alternatives.

DSLs research in the domain of artificial systems and especially robotics strongly increased in recent years. The literature survey by Nordmann et al. presents a detailed analysis of uses of *DSLs* in the robotics domain [NHW14]. The authors discuss the use of specific languages for design, simulation, and programming of robotic systems. While their investigation of quantitative measures and the temporal distribution of publications show that increased research interest is present, the authors also identify the missing reuse of languages.

A recent application of *MDSE* closer to the domains of artificial systems and graphs is presented by Hochgeschwender et al. [Hoc+16]. The authors present results of a *MDSE* approach which incorporates domain *models* at runtime of robotic applications. Their research investigates the roles of graph-based knowledge retrieval and query languages in robotic applications within multiple application scenar-

ios. Neo4j is used as the *Graph Database Management System (GDB)* in the presented implementation and Cypher is used as the *GQL*. The authors make use of a similar graph *model* to Equation (2.6) on page 16 but extend the *model* via specific pre-defined labels. These set labels represent individual domain-specific elements of their domain. However, the authors do not discuss how changes in the domain are executed in this extended *model*. Further, the robot uses the knowledge in the graph at runtime by applying developer designed Cypher queries. Detailed knowledge of the domain and the underlying graph schema is required to allow developers to design these Cypher queries, as the proposed application does not provide query design support to the developers. The authors acknowledge the difficulty to quantitatively evaluate their approach and all of its facets and consequently present a use-case driven evaluation on real world systems. This application at run-time steps beyond previous *MDSE* approaches where *models* are solely used as a design tool for developers. As a result, the performed knowledge access is comparable and close to approaches such as *KnowRob* and *ORO* [TB13; Bee+18; Lem+10]: Formalized knowledge is organized in a graph structure and primarily used by the system to improve the robot behavior (cf. Section 4.3 on page 61).

Similarly, the *Robmosys* project strives for a composable set of *models*, also considering *model* application at runtime [EU17]. In this context graphs are also identified as a central core and are used represent the highest abstraction level. Their graph *model* extends the introduced *labeled property multidigraph* to hierarchical hypergraphs⁷, i.e. a graph in which an edge can connect any number of nodes and in which edges are also vertexes that can be connected by further edges [SLS17]. With the recent start of this project, no further details are available how these extended graphs are practically used in robotic applications.

3.3 MDSE DEVELOPMENT PROCESS

Industry and research commonly recommends language engineers to execute the *MDSE* development process iteratively and in close cooperation with domain experts [Völ06; Völ09; Völ13a; Com17; Obj14; Nor16; BAG18; Bar+12]. Völter suggests an iterative process and distinguishes three categories of *DSLs* which language engineers usually develop [Völ13a]. First, *technical DSLs* factor present knowledge from known existing frameworks, systems, or architectures into a reduced and targeted set of languages. Second, *business domain DSLs* extract (tacit) knowledge from domain experts or given abstractions such as ontologies and bundle information into *DSL*. Third and most difficult, *fragmented domain DSLs* work on domains with no clear given ab-

⁷ See also <https://robmosys.eu/wiki/modeling:hypergraph-er>

stractions and possible split domain knowledge. The core abstractions are unknown, detailed analysis is required, and especially difficult in fragmented domains. The *Cognitive Service Robotics Apartment as Ambient Host (CSRA)* project (cf. Section 4.2 on page 53 for a detailed domain description and analysis) equals such a fragmented domain: An interdisciplinary set of researchers works towards an implementation of an *EISE* domain, which is embedded in similar iterative development process. Völter additionally proposes an iterative language development process towards a stable product which is composed of the three distinct phases of a) elaboration, b) iteration, and c) automation. Further, co-evolving of languages alongside the analysis is a must to avoid an uncontrolled fragmentation of languages. These recommendations are reflected in my proposed process as shown in Figure 3.5 by the following phases: Phase **P1. Domain Analysis** (requirements and system architecture), Phase **P2. Language Design** (implementation-independent architecture and technology mapping), Phase **P4. Automation** (tool generation, deployment, and integration).

Combemale recommends complementary to distinguish the process of meta-modeling and modeling (cf. M_2 and M_1 in Figure 3.1 on page 31) [Com17]. Each of these parts involves different persons and roles at different stages of the development cycles: the *meta-modeling (M_2) team*, the *the modeling (M_1) team*, and the *final user*. While theoretically distinct, these roles can contain potential overlap. Within a varying research setting such as the CSRA, this distinction supports the MDSE process. The separation allows an advanced composition of language and domain concepts as the complexity is decomposed iteratively in the application domain.

Due to the neglect of systematic evaluation within MDSE processes (cf. Chapter 7 on page 135), I propose to additionally include a detailed dedicated evaluation phase in the process [KBM16; GGA10]. This phase contains both, validation via application and detailed analysis of a *vertical prototype*. The goal of the evaluation phase is to reach a tangible evaluation with a high level of evidence, thus improving the language and tool quality [Net+08]. Literature suggests the development process to be tightly intertwined with the evaluation [BAG18; Bar+12]. This involves the evaluation of *DSL usability* which is executed during the development life cycle and executed on pre-defined metrics [Weg+13; Bar+12; Bar13].

Additionally to the mentioned elements, the proposed process is further inspired by recommendations from the *MDA guide* [Obj14], extensions proposed by Nordmann [Nor16], as well as *Language-oriented Programming (LOP)* where the development of a *DSL* is started at a high-level layer in which a well-suited language for the target domain is developed [War95].

Figure 3.5 on the next page shows the iterative development process I propose and apply in this thesis, which contributes towards

○ *development process*

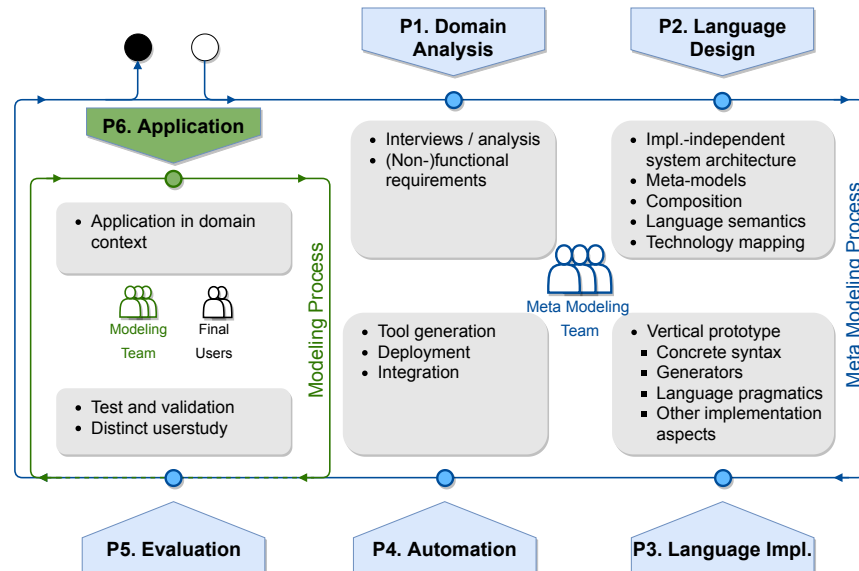


Figure 3.5: The proposed and applied development *MDSE* process. Combining recommendations from Völter; Combemale; Ward; Nordmann [Völ13a; Com17; War95; Nor16]

the overarching research question **RQ2**. This process incorporates the roles and responsibilities of the three involved actors in the applied *MDSE* development cycle. The process is primarily inspired by Völter [Völ13a], Combemale [Com17], and Barišić et al. [BAG18]. The figure depicts the two nested iterative processes consisting of the distinct *meta-modeling process* and *modeling process*. The meta-modeling process, which is executed by the meta-modeling team, is separated into five core parts: 1) domain analysis, 2) language design, 3) language implementation, 4) automation, and 5) evaluation. The latter is connected to the nested modeling process which involves the modeling team as well as the final users and targets the application of the developed languages and tools to the domain.

Each of the six phases (Phase **P1. Domain Analysis** to Phase **P6. Application**) shown in Figure 3.5 has multiple steps and produces phase specific artifacts:

p1. DOMAIN ANALYSIS

The first phase, the *Domain Analysis*, targets to increase the understanding of the domain and all involved concepts. From each execution step of the analysis phase a set of functional and non-functional requirements is defined. These requirements are extracted from the domain assessment done via analyzing documented domain knowledge or consulting domain experts via interviews.

p2. LANGUAGE DESIGN

The requirements and results of the domain analysis are then used in the second *Language Design* phase to extract a *technol-*

ogy-independent system architecture. This architecture is platform agnostic and is mapped to particular technology items in the following *technology mapping* step. The mapping to technology items is explicit and makes clear statements about used standards and platforms.

p3. LANGUAGE IMPLEMENTATION

Phase three, the *Language Implementation*, realizes the previous specifications into a *vertical prototype*, thus implementing the architecture definition and technology mapping. The *vertical prototype* provides a minimal viable product and is extended with each iteration. This step in itself is an iterative process centered around the implementation of the *abstract syntax*, *concrete syntax*, language semantics, and lastly suitable generators for the language(s) [Com17].

p4. AUTOMATION

The fourth *Automation* phase targets supplemental automation. This includes the generation of the final tools (often an *IDE* or set of programs), integration in the domain (i.e. implementation of static glue code and adapter *artifacts*), and handing of their deployment to the final users (including updates in following iterations).

p5. EVALUATION

Phase five contains a set of detailed tests and validations. This step incorporates recommendations of extensive testing of *DSLs*. The *Evaluation* phase represents a dedicated evaluation via distinct user studies within a controlled environment using the previously created *vertical prototype*.

p6. APPLICATION

Besides the previously mentioned distinct evaluation, the last phase executes the application of the *vertical prototype* directly in the domain context (i.e. the *CSRA* project). This phase involves the final users as well as a modeling team who are jointly responsible for the creation and maintenance of *models* for the target domain. This application thus ensures that the *meta-models* are capable to represent the domain via test and validation.

3.4 SUMMARY

This chapter presents the foundations of *MDSE*, the creations of *DSLs*, and *language workbenches*. This includes the different aspects that are relevant in this context to create languages: *abstract syntax*, *concrete syntax*, semantics, and language *pragmatics*. Additionally, the commonly emerging challenges are discussed and what expected benefits one can yield. Related work is presented via the role of *MDSE*

for graphs and intelligent systems. These foundations are used to define the *MDSE* process applied in the remainder of this thesis: an agile and iterative development process that takes a strong emphasis in the language design and evaluation. This process is separated into six phases handling analysis, design, implementation, automation, evaluation, and application. Lastly, the process also identifies the different roles active in the process and maps these roles to the individual phases.

Part III

MODELING INTERACTION RELEVANT KNOWLEDGE IN SMART ENVIRONMENTS

The third part analyzes the concepts and relations of interaction within *smart environments* alongside the *Cognitive Service Robotics Apartment as Ambient Host* project. It further proposes a *model* of interaction relevant knowledge as an ontology. Based on this analysis implementation-independent *domain-specific languages*, their semantics and composition are presented.

A MODEL OF INTERACTION RELEVANT DATA FOR INTELLIGENT SYSTEMS

“A human must turn information into intelligence or knowledge. We’ve tended to forget that no computer will ever ask a new question.”

—Grace Murray Hopper
developer of the first compiler for
a computer programming language

This chapter presents a domain analysis of the *Embodied Interaction in Smart Environments (EISE)* domain. I examine the domain alongside the *Cognitive Service Robotics Apartment as Ambient Host (CSRA)* project, an application scenario implementation representing an exemplary *EISE* domain. From this scenario I present a domain description as well as the roles and responsibilities present within the project. Based on interviews with the developers and based on the scenarios implemented in this project, I identify representative questions asked towards the domain. From the perspective of the developers, a query system in this domain needs to be able to fully answer these questions. As a last contribution I present an ontological *model* capturing the interaction relevant concepts.

This chapter aligns within Phase **P2. Language Design** of the development process applied in this thesis. The presented analysis and domain description are an artifact of this process investigating research question **RQ1**. The derived *model* of interaction relevant data serves as an underlying artifact used for the following Phase **P3. Language Implementation**.

The presented analysis and *model* are the result of multiple iterations executed within the *CSRA* project and parts have also been published previously. This primarily includes the two publications “An Ontology for Modelling Human Machine Interaction in Smart Environments” (presented during the Proceedings of SAI Intelligent Systems Conference (IntelliSys) 2016) and “How to Address Smart Homes with a Social Robot? A Multi-modal Corpus of User Interactions with an Intelligent Environment” (published in the Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016) [KWC18b; Hol+16b]).

4.1 EMBODIED INTERACTION IN SMART ENVIRONMENTS

A *smart environment* is an extension of the concept of ubiquitous computing (the idea of omnipresent computing capabilities) and according to Cook and Das [CD05] it is defined as follows.

Smart environments combine perceptual and reasoning capabilities with the other elements of ubiquitous computing in an attempt to create a human-centered system that is embedded in physical spaces. [...] [It] is a small world where all kinds of smart devices are continuously working to make inhabitants' lives more comfortable. [CD05]

In contrast to these ubiquitous systems, the theory of embodied cognition considers the shape of the entire body of an organism as a first class citizen and a central component necessary for cognition and cognitive tasks. With the field of *human-robot interaction (HRI)* research this theory is investigated by the design and explicit use of robotic companions with similar or close to identical human physical sensors and actuators.

Nouvelle AI is based on the physical grounding hypothesis. This hypothesis states that to build a system that is intelligent it is necessary to have its representations grounded in the physical world. [Bro90]

*Embodied Interaction
in Smart Environments (EISE)*

When combined, the two ideas of a *smart environment* and embodied cognition in *HRI* merge to the concept of *Embodied Interaction in Smart Environments (EISE)* in which both, an embodied robot and an ubiquitous system, share a common space in which they support humans in their daily lives. The agents (e.g. a robot companion and a smart environment system) can make use of each of their individual strengths and overcome individual shortcomings to further support the environments inhabitants. For example, while an ubiquitous system often has a broad view on the complete environment, it lacks sensors for searching tasks which are necessary for a detailed analysis of an area (e.g. finding misplaced keys in the apartment). In contrast to this, embodied agents are often equipped with high quality local sensors required for navigation or pick-and-place tasks. These robots can provide their additional sensing capabilities to conduct a detailed environment analysis.

Prominent examples of *smart environments* are smart homes, as they are composed of many sensors, or sensor networks, capturing relevant variables of the environment. Additionally, various actuators commonly provide multiple actuation capabilities allowing the environment to engage with and influence its surrounding. The rising popularity of smart home solutions and smart home technology results in increased availability and application [Ric+06]. The most established implementations target support for private households and

are available in various complexities. The offers range from full (often ubiquitous) systems, such as systems based on the *KNX* standard [ISO14543] or less intrusive systems which allow for effortless installation, such as the *appleHomeKit*¹, to rather simple personal devices and assistants, such as *Alexa*² or the *Google Home*³ system. Further than the application in private homes, one can observe an increased adoption of smart home technology in elderly care scenarios research [Mor+13; Cav+14]. These approaches are examples for applications of the *EISE* domain, as work in this area additionally incorporates personal robots to support humans in their daily living and provides an embodied interaction.

From a technological perspective a significant challenge of systems in the *EISE* domain is posed by the overall system complexity and its heterogeneous nature. Integration of hardware and software components in such a joint environment requires developers to take into account domain-specific characteristics of both domains. Developers of individual interaction components need to access and incorporate interaction relevant data, information, and knowledge from all modalities (i.e. different data sources, storage properties, schema, etc.). Any support in this information retrieval process will reduce the required knowledge about all these modalities and in turn reduce the query design complexity. As a first step to identify the central elements and concepts of high importance to the domain, the following section presents a domain analysis alongside the *CSRA* project.

4.2 DOMAIN ANALYSIS

To reach this chapter's goal of developing a *model* for interaction relevant data within the *EISE* domain, I conduct an analysis of the domain and its involved actors. The analysis is executed alongside the implementation of an application scenario, namely the *CSRA* project, in which I participated during my research [Bie13]. The system is presented by Wrede et al. in detail [Wre+17], but to provide appropriate context and relevance for my *model* of interaction relevant data, I first present the core elements of the system and secondly the domain analysis and its results in the following.

4.2.1 The *CSRA* Project

The *Cognitive Service Robotics Apartment as Ambient Host (CSRA)* is a large-scale project of the *Cluster of Excellence Cognitive Interactive Technology at Bielefeld University (CITEC)* aiming to provide an *EISE* domain laboratory as described in Section 4.1 on the preceding page.

¹ <https://www.apple.com/ios/home/>

² <https://alexa.amazon.de>

³ https://store.google.com/product/google_home

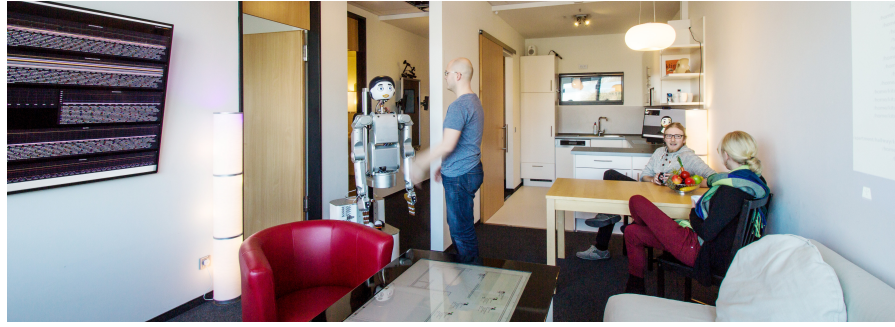


Figure 4.1: An example picture from within the CSRA from the living room showing an interaction with Floka in the apartment.

Its laboratory space is build up to accommodate an apartment-like area composed of three rooms and a connecting hallway which incorporate a total space of 60 m². Figure 4.1 shows a picture taken within the apartment during a handover interaction. Additionally, Figure 4.2 provides an overview of the rooms and the *smart environment* elements, such as sensors and actuators. The laboratory area includes a large multipurpose area which contains a fully functional kitchen, a dining area, an open living room, and a functional bathroom⁴. Further, a robot room provides space dedicated *HRI* with an autonomous robot inhabiting the apartment at all times. Adjacent to the robot room exists a control room which allows researchers as well as developers to work on the laboratory system or study conductors to supervise and observe running experiments. The control room is not a part of the CSRA laboratory space in terms of scenario execution and is therefore excluded in further descriptions. The overall goal of the CSRA is to provide a research platform which allows to investigate questions regarding cognitive interaction in daily scenarios. The environment serves as a basis for quantitative and qualitative research within a controlled environment that provides reproducible conditions joined with automated recording and post-processing of the gathered experimental data [Hol+16b; Ric+16; BE18; RK18]. Exemplary questions of relevance being addressed in this laboratory setup are

- Which interfaces are ideal to support specific functions of a smart home or a mobile robot?
- How do users of this environment address the available functions?
- Which information and knowledge is of high interest or even required for individual software components?

Overall, the research topics reach from *smart environments* (containing ambient intelligence and ubiquitous systems) to social robotics

⁴ Due to ethical and privacy reasons the bathroom only includes simple motion sensors unable to record audio or video



Figure 4.2: Map view of the CSRA laboratory.

(including virtual and embodied agents). This vision is realized by providing an *EISE* domain which is composed of extended smart home features as well as a cognitive social robot with advanced manipulation capabilities. As such, it offers a dense sensor- and actuator network which embeds virtual agents and a mobile robot, providing embodied and personalized interaction with humans. The system is designed to operate 24/7 so that any interaction episode in this environment can contribute to interaction components. This is specially helpful for adapting software components using for example machine learning approaches. As a result, access to higher level information and knowledge of past episodes beyond the raw sensor data is a necessary functionality for the running interaction software components.

Compared to related approaches, the *CSRA* differs in its conceptualization and research areas [LLM15]. The project prioritizes the role, interplay, and usage of devices used for embodied interaction with the smart environment. An anthropomorphic mobile service robot provides the designated role as preferred agent for interaction with the environment. We consider the capabilities of the semi-autonomous robot as more extensive than the isolated smart environment. Together, these weakly coupled systems operate independently with the possibility for bilateral cooperation and support (e.g. exchange of data, knowledge, or information).

THE SMART ENVIRONMENT ELEMENTS OF THE CSRA

The CSRA contains multiple sensors and actuators to facilitate interaction with humans. This includes common smart home sensors, such as movement, temperature, power usage, present devices, capacitive floors or programmable wall switches, as well as actuators, such as controllable lights, heating, blinds, or wall plugs. Additionally, there are various displays, projection areas, audio interfaces as part of the experimental research setup. These include multiple home automation systems and standards, such as KNX⁵ [ISO14543], zigbee⁶, and openHab⁷. Technically, the system uses a homogeneous service-oriented software architecture. All systems are integrated together via the common middleware *Robotics Service Bus (RSB)* to provide access to available data for all involved software components.

THE (EMBODIED) INTERACTION ELEMENTS OF THE CSRA

Beyond the usual smart environment interfaces, additional elements fostering interaction are present in the CSRA environment. For example, the ceiling is exhaustively equipped with depth sensors facing downward, capturing the entire apartment space. This setup is primarily used to provide a global overview over the entire apartment. In conjunction with dedicated software components, this setup allows us to provide a consistent apartment wide person tracking system. Similarly, systems for situation recognition, speaker detection, and interaction group detection are implemented and researched. Additionally, the bi-manual mobile *Floka* robot is present in the CSRA (a *Meka* robot base [Gui11] modified for optimal interaction capabilities via the *Flobi* head, which replaces the MEKA M1 default head [BE17]). It operates as an autonomous agent which is able to interact physically with the environment and serves as an embodied interaction partner for end-users. With its mobile platform it can complement the otherwise stationary sensors and actuators of the apartment in scenarios such as, clean up or search and find. Further, two individual virtual *Flobi* agents are present in the entrance area and in the kitchen as interaction points for task specific interaction and support (i.e. welcoming or kitchen cleaning). Similar to the smart environment elements, all above listed additional interaction elements are also integrated into *RSB* as a source for interactive applications.

DATA, INFORMATION, AND KNOWLEDGE OF THE CSRA

As shown, the system and its elements manifest a heterogeneous, complex, and highly diverse software and hardware ecosystem. Besides the base software, interactive systems or software components

5 <https://www.knx.org/>

6 <https://www.zigbee.org/>

7 <https://www.openhab.org/>

can also access the available (sensor) data via the common middleware. In Table 4.1 I present an estimation of the accumulating data within this system (not including data the mobile robot can supply) It shows that recording of raw and un-abstracted data in this scenario is not feasible. Consequently, higher level abstractions (i.e. knowledge

SENSOR TYPE	AMOUNT	FRAMERATE (HZ)	DATA (MIB/S)
Depth sensors	14	30	615,23
HD cameras	5	25	741,58
Microphones	12	16000	0,366
Various Small Sensors	100	30	5,0
Total			1362.176

Table 4.1: Estimation of sensory data of the CSRA smart environment. This estimation excludes data extracted by a mobile robot.

and information as presented in Section 2.1 on page 12) such as concepts of persons, their physical movement trajectories, conversations, or conversation topics need to be extracted and made available for dependent software components.

4.2.2 Roles, responsibilities, and required knowledge

Four central groups of actors are involved within the CSRA environment. Each of these groups holds individual roles, responsibilities, and knowledge in the domain. Consequently, participants of the groups have different requirements for model-driven information retrieval support.

The most basic and direct interaction recipients are *naïve users* who represent the target audience or end-users for the overall system and environments. They are the recipients of services offered by the system (e.g. as demonstration attendees or study participants) and are thus not directly involved in the design and implementation of services or programs. Nevertheless, depending on the influence users are allowed to have on the automation, *naïve users* need to be enabled to control the systems provided by appropriate means. Current state-of-the-art *smart environments* thus give users controls to allow coordination of activities via trigger-action programming [Jon+14]. These are commonly in the form of rule based *domain-specific languages (DSLs)*⁸, which allow to express rules such as “disable the heaters if any of the windows are opened”, or “turn off the lights if no person is present”. The appropriate *DSL* concepts are commonly part of the environment and are bound to real world entities, such as heating, lights, or windows.

⁸ For example the *openHAB* rule *DSL* [ope13]; See <https://www.openhab.org/docs/configuration/rules-dsl.html>

With the CSRA being a laboratory setup, *study conductors* are a separated role in the domain. They oversee the execution and observation of studies or demonstrations of the environment that are conducted within this *EISE* domain. Their knowledge thus needs to cover the basic usage of the system as well as all required elements to start and stop the system or individual components of it.

In contrast to this, *behavior developers* are closely involved in the system development and make use of available capabilities. They are domain experts on how to create suitable interactions for the *naïve users* via the available actuation and interaction mechanisms. Interaction design and creation is their core task in this role and thus this task requires them to have appropriate domain-specific knowledge and programming skills. As a result of their specificity, *behavior developers* do not necessarily know a) which interaction relevant data, information, or knowledge is stored (cf. [Section 2.2](#) on page 14) and b) in what format the data is stored and accessed optimally (cf. [Section 2.3](#) on page 17). This role specifically does not require to have this detailed knowledge, nevertheless the developers need to have access to all layers of the *data-information-knowledge-wisdom (DIKW)* pyramid to successfully design intended interactions.

At the lowest level, *system developers* provide infrastructure and all other services of the underlying system. This includes on the one hand basic tasks such as mounting and maintenance of sensors, actuators, computation machines, cables, and all other infrastructure; this requires detailed knowledge about technological requirements for the successful execution of all software components (e.g. required computing power, storage or connection speeds). On the other hand, the overall system architecture needs to be defined and maintained by *system developers*, including the communication protocols, utilized middleware, key software components for sensor/actuator data provisioning, knowledge and information extraction, *DIKW* storage and querying; these services are considered to provide the base functionality. Additionally, *system developers* need to define the interfaces and access points for *behavior developers*, which is a non-trivial task. Especially in research environments the system composition can change quickly based on study goals, resulting in varying hardware and software setups. Additionally, interaction relevant data is highly volatile; information exchanged within an interaction is strongly context dependent and varies between users. Given these unusual properties, the storage of higher level knowledge or information in this domain can thus not easily be achieved with traditional storage mechanisms (cf. [Section 2.3](#) on page 17).

From the perspective of this thesis, the *system developers* are responsible for the application of a *Model-driven Software Engineering (MDSE)* approach to support the information retrieval process. In turn, the recipients are primarily *behavior developers* who are domain experts of

adjacent (potentially non-technical) domains and require the access to domain-specific information and knowledge. Given the aforementioned number of sensors and actuators employed, there is hence a large amount of data comprising of various modalities available. This data needs to be stored in a fashion so that it can be readily queried within the interactive scenarios. While *system developers* are familiar with the required *Database Management System (DBMS)* details (i.e. data schema and query languages), *behavior developers* typically lack the specific knowledge to efficiently work with complex database management systems. While the group of *naïve users* are not the intended recipient, they can still be questioned to provide further domain insights. This group can for example provide their perspective of how they prefer the interaction with the system, impacting the domain *model* and queries.

4.2.3 Knowledge queries in the EISE domain

This section discusses common questions stated in the domain before an abstraction of the domain can be derived. As an example, consider the following interaction sequence. It consists of a greeting scenario as depicted in [Figure 4.3](#) and a subsequent cleanup task, which both incorporate knowledge of previous interactions. Two persons enter the apartment together upon which they are greeted by the virtual agent *Flobi* in the entrance interaction island (cf. [Figure 4.2](#)). Before the start of the interaction, the agent (i.e. the interaction behavior component) needs to inquire whether or not the currently targeted person has been seen before. If the persons have been there before, the agent accesses the stored information (e.g. their names) and personal preferences regarding environmental settings (e.g. lighting or heating set-



Figure 4.3: Exemplary interaction scenario in the entrance interaction island from the user perspective.

tings). Following, the settings are applied and the users are greeted by name. Depending on content and topics of previous interactions, the agent enters a conversation regarding open matters, such as left messages by other interaction partners. Otherwise, the conversation starts anew and the agent introduces itself, introduces the environment, the environment capabilities, and lastly, asks the new users whether to remember them via face identification and name. Once entered, the agent could offer refreshments and in case the persons have not been in the apartment before, corresponding information needs to be supplied by the agent. In a second part of the interaction the persons are prompted to support a cleaning task within the kitchen. Objects such as cutlery, glasses, and plates lie on the kitchen counter and are detected by the agent to be removed by storing them in the corresponding drawers. In this cooperative task the virtual agent follows the goal to obtain help from the present persons. This interaction requires access to similar information as described for the greeting scenario, as the agent is independent of the one at the entrance interaction island. Additional to information about the entered persons, new queries need to be answered whether or not the persons have executed a similar cleaning scenario before. The virtual agent can further provide feedback regarding the designated drawer and cupboards for the individual items upon request, thus fulfilling the cooperative aspect of this task.

A second example for an interaction scenario involving the autonomous embodied agent is a search scenario. In this example, a person in the apartment is unable to find the keys within the apartment. As a result, the person asks the environment's virtual agent if they can supply help in the search. If the apartment has information on previous locations of the keys, it can provide the relevant information. With the overlooking perspective via the installed cameras, the apartment only has a low resolution image of the rooms. It can thus not directly participate in this search scenario and requests the embodied agent to engage in a detailed exploration using its mobile sensors. The mobile agent then participates in the search together with the person and provides the location of the item in case of success.

competency ○ As the above described scenarios show, the *behavior developers* questions questions require access to a large and diverse set of questions. Exemplary questions lifted from these scenarios include questions such as:

- Have I seen the person in the wardrobe/hallway/kitchen before and if so when?
- What is the name of the person P in the room R?
- What were topics of the last conversation with person P and should I explain how to do task T?
- Does the person P know where to find drinks/cups/object O in the apartment?

- What are names of persons that entered the apartment within the last 5 minutes?
- Have previous interactions with the present persons P involved the cleaning scenario?
- Where has the object O called 'Keys' been seen within the last hour?
- Do known persons in the apartment know each other and what were previous conversation topics?

The questions allow to identify central aspects a *model* of this domain needs to cover (RQ₁):

- Sensor and actuator information,
- Objects information,
- Agent and person information, and
- Spatiotemporal information of the above aspects.

The questions serve as the exemplary competency questions which a *model* of the domain needs to be able to answer.

4.3 RELATED WORK

Li et al. present a detailed overview on ambient systems which provide cognitive assisted living environments [LLM15]. From the authors' perspective, the aging population due to current demographic change imposes strong challenges with respect to healthcare, rehabilitation, and general assisted living while maintaining user independence and quality of life. The authors focus strongly on the connection of embodied interaction with a smart environment and consequently the users. In comparison, the CSRA project features these aspects explicitly via an anthropomorphic mobile service robot in combination with an extensive smart environment. Li et al. generally present a wide variety of platforms and discuss them in context of current research aspects. This includes projects covering domains such as large scale smart home environments mobility access projects, social inclusion, robotic service platforms, and human machine interfaces. A central conclusion is the missing integration of services, devices, and individual systems. Additionally, the authors encourage the execution of further studies targeting usability, user acceptance, and user expectations towards these systems.

A related robot-centric human support approach is presented by Tenorth and Beetz in the *KnowRob* project [TB13; Bee+18]. The authors introduce a knowledge representation and reasoning approach

for robotic agents. Though only partially in the *EISE* domain, this approach provides a query answering system for an autonomous agent acting in personal environments supporting humans in physical manipulation tasks. The robot can obtain information about previously captured images or motions to know *how* to reach its goal. At its core, the KnowRob system consists of multiple ontologies representing robots, their tasks, or the situational context of objects, which allows to semantically annotate the meaning of real objects to low level data in the ontology. These formalizations provide the basis for control systems and allow queries by the robot at runtime. Evaluations and experiments validate the approach on large sets of observations and real world scenarios. The project is centered on the robotic actions and does not consider smart environments or external sensors.

The underlying model used by KnowRob is encoded within an upper ontology such that the ontology provided by KnowRob can be seen as an extension of the preexisting OpenCyc ontology [Len95]. While CYC itself is an expert system with a domain that spans all everyday objects and actions - as which it served as an upper ontology targeting natural language understanding and machine learning - KnowRob extends this rather broad human knowledge base with more domain-specific concepts needed for robots. These extensions to the ontology cover the descriptions of everyday tasks, general household objects, and most crucially, robot parts. The central KnowRob upper ontology can be categorized into the following areas:

- **MathematicalObjects:** Provides all math related concepts such as vectors, coordinate systems, or matrices
- **TemporalThings:** The description of events and actions
- **SpatialThings:** Physical layout of the robots surrounding, but also the physical layout of itself
- **HumanScaleObject:** Contains the objects in the robot environment, including body parts and furniture

As a consequence the ontology allows to model

- Robots, corresponding body parts, connections of parts
- Sensor/actuator capabilities
- Objects, including individual parts, functionality, and configuration
- Robot tasks, actions, activities, and behaviors
- Contextual information about situations and the environment

Generally, the KnowRob systems follows a closed-world semantic, meaning that all unknown knowledge items are assumed to be false. As a result, the non-existence of anything does not have to be described in the underlying knowledge base. The recently published *KnowRob 2.0* extension further allows to include other ontologies into the overall system to add domain-specific knowledge into the provided core knowledge base [Bee+18]. The entire KnowRob system uses its ontology at the core but is composed of a hybrid system architecture to combine additional general- and special-purpose methods. Architecturally these methods interact with the ontology as they are build on top of the available knowledge. These additional methods combine for example features such as probabilistic inference, robot capability matching, or classification and clustering.

The hybrid approach is facilitated by the usage of powerful query language *Prolog*. Beetz et al. chose the logical programming language Prolog as the “interlingua” for robot knowledge processing. Internally within KnowRob, *OWL Web Ontology Language (OWL)* statements are consequently represented as Prolog predicates and common Prolog inference methods can be applied to these predicates. As a result, *Resource Description Framework (RDF)* triples are loaded and stored internally and *OWL* reasoning can be applied on top of these representations. On top of the systems triple store, increasingly complex *query predicates* operate and abstract step wise from lower level representations to higher conceptualizations. This extension of predicates allows programmers to create their own abstractions and chose precisely which predicate from which abstraction layer to include within their own queries. This abstraction reaches all the way up to the possibility to define simple global plans making use of multiple lower level predicates, for example such as the `ehow-make-pancakes1` plan using `pancake`, `frying-pan`, or `mixforbakedgoods2` predicates as listed by the authors in [TB13].

For the authors, this powerful abstraction mechanism provided by the Prolog language is a central reason for its choice: Prolog is more expressive than other logical dialects and allows to query more complex relations. On the one hand, comparing Prolog to *RDF* it shows that *RDF* allows for more efficient reasoning but is less expressive. On the other hand, the Prolog language is not as universal in its representations as other languages, for example *CycL* which would allow to represent nearly every natural language expression [Mat+06]. Prolog lies in between these languages and provides a well documented language to robot developers. Most prominent, the ability to inspect the knowledge base using Prolog provides the programmers with more power. For example it allows for additional logical inference on the existing knowledge at query design time. However, within KnowRob the Prolog language is primarily used as a knowledge query language rather than in the context of inference tasks. Only as a dedicated

query language, Prolog can be directly be embedded in the robot control loop. This fact represents the core challenge going along with the choice of Prolog as the query interface for the robot programmers and developers. The language uses a depth-first search with backtracking which can result in endless queries. Queries consequently need to be designed, optimized, and tested very carefully as they otherwise might block the entire system. The users of the languages (being robot programmers and graduate students) thus have to invest heavily in their queries at design time. Last but not least it is important to note, while many foreign language interfaces exist for Prolog, it lacks special-purpose reasoning mechanisms for uncertainty, temporal, or spatial reasoning. These features are covered in the KnowRob system via the aforementioned hybrid approach with dedicated methods.

4.4 A MULTI-MODAL INTERACTION CORPUS

The CSRA is used to develop smart home technology systems as well as to study human-machine interaction in the context of smart environments. Within the context of the project, Holthaus et al. and I created a multi-modal corpus of user interactions within the *EISE smart environment* [Hol+16b]. We explored multiple verbal and non-verbal interfaces by providing participants with different modalities for interaction to fulfill a given task description. To reach a given goal, such as to turn a light on/off or alter its brightness, participants were able to either chose to interact with the environment directly (e.g. using pointing gestures, talking to the environment, or clapping) or address the autonomous robot (e.g. via waving or speech) to ask for help to complete the task. The participants' choice of modality was not influenced but rather endorsed to be an exploration task in which common ways (e.g. using wall mounted light switches) were explicitly forbidden.

The conducted study was a remotely controlled study (a "Wizard-of-Oz" [Kel84] setup) which allowed a human operator to generate the correct feedback for any goal directed action of the participants. This lead the participants believe their strategy is a valid interaction mechanism and confirms their choice of interaction for subsequent tasks. During the study execution we recorded all sensor data, actuator data and other system internal events, including dedicated participant video and audio recordings. In a second step we created an integrated annotation also containing ground truth information about participants behavior and actions. The resulting published dataset is openly available for researchers to use as for evaluation and further analysis of human behavior in the *EISE* domain.

4.5 AN ONTOLOGY OF INTERACTION RELEVANT KNOWLEDGE

With the overarching goal of this work to support the querying process of interaction relevant data in interactive smart environments, a *model* of the domain at the conceptual or 'knowledge level' is required. This *model* abstracts from specific data schemata and also benefits the portability of solutions to other systems and platforms. To generalize at this conceptual level, one needs to formally specify the relevant interaction associated concepts and their relations. In an early iteration of the *MDSE* process I therefore designed an ontology that *models* the domain of interaction in interactive smart environments alongside the *CSRA* project. Knowledge management in information science is commonly executed by the creation of ontologies or knowledge graphs [Walo7]. Wallace defines an ontology as follows:

- (1) an ontology is an artificial construct that may have a link to a naturally occurring phenomenon, (2) an ontology is a tool for knowledge representation, and (3) an ontology is an explicit but abstract and simplified conceptualization. [Walo7, p. 185]

With this definition, an ontology is a similar conceptualization as *meta-models* created in the context of *MDSE*. The central difference lies in the degree of detail [PU03]. An ontology contains a set of classes, their relations, provides a fixed vocabulary of the domain. It allows to formulate meaningful statements within a domain using this vocabulary. The ontology grammar additionally defines meanings and ensures well-formedness of the statements; it is thus able to represent the data of a domain. *Meta-models* are, however, an abstraction used to describe how a domain-specific *model* is build. These abstractions describe a more formalized specification of individual domain notations with a reduced rule set. In this sense I concur with Pidcock and Uschold, who conclude that valid *meta-models* are thus also ontologies [PU03]. The inverse statement, however, is not valid and not every ontology provides a formalized model as a *meta-model*. I therefore use ontologies as a tool to analyze the domain is its structure. The ontology represent a graph where individual classes (nodes) are linked together (relationships) to form an abstraction of the real world.

As documented in literature, the creation of a domain ontology helps to accomplish several goals, most importantly [NM+01; SS09]:

- Analyze the domain
- Explicitly state domain assumptions
- Split the operational knowledge from domain knowledge
- Create a common ground and understanding of the domain information structure

- Allow to utilize domain knowledge in systems

Thus, an ontology as an analysis step supports reaching the overarching goal as introduced in [Section 1.1](#) on page 6:

- **Conceptual abstraction and interoperability:** An ontology supports an interaction with a data management system at the conceptual level, i.e. at the level how users tend to conceptualize the domain rather than how the data is modeled using specific schemata that are designed with other goals in mind (e.g. efficiency of querying). Per definition, an ontology supports integration of different data sources as well as the communication between different components that are forced to speak the same ‘language’ by the ontology with no need to know the details of how to technically access particular data.
- **Model-driven approaches:** Using an ontology, that is a declarative specification of the domain, comes with the benefit of being able to adopt model-driven approaches to system engineering which support taming the complexity of heterogeneous systems in which many constraints need to be satisfied. It provides the benefit of being able to automatically generate components from the ontology, e.g. query interfaces.
- **Reasoning:** Using an ontology comes with the benefit of being able to use inference mechanisms to check consistency of certain situations but also to infer new knowledge.

This work is inspired by the common use of ontologies in the subdomains of smart environments and robotics. The following sections hence present an overview of related work in these fields.

4.5.1 *Smart environment ontologies*

Many ontologies have been developed in the smart home/smart environment domains, which address different areas such as human behavior recognition or health monitoring [Msh+18; Rod+14]. One example ontology for the domain of smart environments includes the ontology developed by McAvoy et al. [MCD12], who present an ontology-based context management system to deal with difficulties resulting from the ambiguity of collected data. Their efforts also cover temporal reasoning, sharing, and re-using data amongst various applications. A very relevant ontology for the domain of interaction in smart environments is the *Sensor Network Ontology*, which was published in 2005 by the *World Wide Web Consortium (W3C)* and is concerned with modeling sensor networks and their properties [Com+12]. It consists of 51 concepts and 55 properties and central concepts in this scope are *Sensor*, *Observation*, *ObservationValue*, *Deployment*, and *System*. The

Sensor Network Ontology is a domain-independent ontology that can be used in domains composing high amounts of sensors. As a result, the proposed ontology for the domain of embodied interactive smart environments imports this top-level ontology and reuses its concepts.

An example of an ontology that takes into account the temporal structure of situations and their evolution over time is the contextual model developed for smart home applications by Mallik et al. [Mal+15]. Their system is able to track humans and situations occurring in smart homes, and to make predictions on the evolution of these situations based on the current observations and on ontological reasoning.

Other ontologies have been developed to support the recognition of human activity in the smart home domain, e.g. Wongpatikaseree et al. [Won+12]. They rely on a context-aware ontology to define description logic rules that can be used to infer/predict activities on the basis of location and posture information.

On a more abstract layer, the effort of the *schema.org* community tries to provide a common vocabulary jointly [Mik15]. It is developed by Google, Bing and Yahoo and includes few interaction concepts. Relevant example concepts and their properties of importance in this context are *Person*, *InteractAction* and *Place*. The central role of *schema.org* is the ability to incorporate information into web sites so that search engines can extract this structured information, e.g. by using micro formats. One downside of the models of interactions in *schema.org* is the static definition of its concepts. As a result, concept dynamics (e.g. changes on the concepts) and temporal structure are not covered.

4.5.2 *Ontologies in robotics*

In the domain of robotics, several ontologies and complex knowledge modeling frameworks have been developed to equip robots with knowledge and reasoning capabilities. The *KnowRob* system, for instance, focuses on representing task-specific and object knowledge to support robots in reasoning and planning their own actions [TB13; Bee+18]. Knowledge is encoded in *Web Ontology Language* ontologies and the system provides a Prolog based query answering system for agents. Encoding task specific information relevant for manual task execution makes *KnowRob* a robot-centric approach. The framework thus does not provide classes for representing concepts related to human machine interaction. In theory the approach is a model based system which uses ontologies as models for the world knowledge. For the applied use case the level of formalization is suitable: ontologies are sufficient for the intended modeling purposes and reasoning capabilities of the *KnowRob* system. The system consequently has difficulties to deal with modification as well as model evolution. Inconsisten-

cies and inference of new knowledge in the robot knowledge base are difficult to detect. The query capabilities of the *KnowRob* system are provided by using the *Prolog* language. Queries towards the system are thus required to be carefully designed and optimized as *Prolog*'s depth-first search is incomplete and can eventually result in infinite searches – even if possible results exist. *Behavior developers* therefore need to understand the heterogeneous domain, the ontology structure and *Prolog* optimization techniques to be able to formulate well performing queries. As already discussed by the authors, the system approach is hence difficult to combine with machine learning algorithms. A recent extension by Balint-Benczedi et al. addresses these issues and provides a dedicated interface language comparable to *SPARQL Protocol and RDF Query Language (SPARQL)* (see Section 5.2 on page 78 for more detail) [Bal+17].

The *KnowRob* system is made publicly available via the succeeding cloud robotics application and knowledge service *openEase* [BTW15]. It provides access to the information for robots and researchers with analogously semantic data access when compared to the features already present in *KnowRob*. Queries towards the platform are selectable in menu of natural language representations linking to the actual highly complex queries. These queries can be executed towards the knowledge base and the result is presented in the integrated web view. The knowledge service provided by *openEase* consists of three integrated central elements: A large database containing episodes of joint human robot manipulation tasks, an ontology that represents the underlying conceptual model of manipulation activities, and tools for querying, visualizing, and analyzing of manipulation task episodes.

The *Open Robot Ontology (ORO)* approach exhibits another example which makes use of ontologies in the domain of domestic service robotics [Lem+10]. *ORO* is an ontology based knowledge processing framework supporting agents with cognition in *HRI* environments. It acts as a central intelligent blackboard storage for robots to store or retrieve knowledge. Lemaignan et al. focus on maintaining a consistent knowledge representation by continuously updating and checking the ontology for inconsistencies. Information is mainly gathered by the robot via natural human interaction (i.e. speech or textual input) and only knowledge about objects and their location is stored in their application scenario. The approach also comprises a common sense ontology for robots which is close to the *KnowRob* upper level taxonomy. *ORO* uses an *RDF* triple store (the Jena framework) at its core and uses first-order logic formalism to represent knowledge. The querying interface of the knowledge base is realized via a dialect of *RDF* and *OWL* description logic. Evaluation is executed using synthetic exemplary task implementations such as point and learn or an object identification game. The authors identify issues regarding the ontology consistency maintenance. Consistency checks are potentially

resource intensive, especially for large ontologies. Similar to *KnowRob* the ontology is also robot centric and no explicit knowledge about the interaction is considered. While the system queries the knowledge from the triple store, no further detailed query design support is offered and existing query languages are reused.

4.5.3 *Graph-based approaches*

The large-scale knowledge engine for robots *RoboBrain* provides task execution relevant knowledge [Sax+14]. The authors merge multiple sources for knowledge, e.g. via observation, machine learning, or on-line resources analysis. Any insert into the system then triggers inference to unify the present knowledge base. Unlike *KnowRob*, the core knowledge storage of *RoboBrain* makes use of a labeled directed graph ($G = (V, E)$), which holds no properties. No common query language is reused to access the information in the system. Information retrieval is realized via a dedicated robot query library which contains retrieval functions and suitable programming constructs and thus provides traversal and pattern matching queries. The authors evaluate the system via detailed application examples.

A more practical application is provided by Fourie et al., called *SLAMinDB* [Fou+17]. The introduced system realizes a shared centralized persistence layer for memory storage and retrieval in mobile robotics. The authors combine the graph database Neo4j together with the document store *MongoDB*, to jointly store low-level data for navigation, such as *Simultaneous Localization and Mapping (SLAM)* data or obstacle information. The link between layers is realized by storing data identifiers and no details on their combination is described. Only Cypher graph queries are presented and it is unclear how developers actually obtain data from the system. The presented approach allows to attach timestamps to the stored data which is then retrievable for the *SLAM* algorithm. More complex calculations which abstract the data appropriately for the *SLAM* algorithm are realized in Java functions in the Neo4j server. With the combined databases holding no schemata of its data, Fourie et al. do not provide a model of the domain; the storage graph expresses a very specific domain model which is not introduced. Similarly, temporal aspects of queries are considered by creating temporal queries by hand. These crafted queries use timestamps in their matching clauses to filter the present data. Further, their chosen system architecture (i.e. plug-ins and functions residing in the databases) requires frequent system restarts of the centralized location of truth for the *SLAM* algorithm. Query design and developer support are not considered and developers require full domain knowledge, especially to obtain real data from document store. Approach evaluation is application and experiment driven evaluation, showing that the proposed approach is feasible.

4.5.4 The EISE ontology

A conceptualization of the *EISE* domain is an important part of the applied development process. Within the *model* of the domain, the concepts and their relations are expressed as explicitly as possible. In an early iteration, I thus created an ontology covering the *EISE* domain to gain an understanding of the involved entities which are of relevance to the *behavior developers*. The identification of competency questions is helpful to determine what the ontology needs to answer. I thus directly use the knowledge queries determined in the previous [Section 4.2.3](#) as the competency questions towards the ontology. These questions and the central aspects in them show, that an ontology of interaction for the domain of interactive smart environments needs to take into account that: a) large numbers of sensors and actuators are in the environment, b) several different objects are present in environment, c) autonomous embodied agents and persons act and interact, d) any of the above the concepts also holds spatiotemporal information.

As building blocks of the proposed ontology, I build on the following aspects: sensor-related concepts, interaction participants, spatiotemporal representation, and interaction concepts:

- **Sensor-related concepts:** Modeling sensors, their physical location, properties, schemata etc. are crucial to capture the interaction in the *EISE* domain. I reuse the *W3C Semantic Sensor Network Ontology* for this purpose [[Com+12](#)].
- **Interaction participants:** Different participants are involved in interactions, for example virtual agents, embodied agents, and persons. As the structure of persons, their properties and relations have been studied in depth, I reuse the *Friend of a Friend Ontology* [[BM14](#)].
- **Interaction concepts:** A taxonomy of interaction types is crucial in the targeted domain. I build on the HRI taxonomy introduced by Yanco and Drury that focuses on human social interaction, and extend it appropriately for my purposes. It describes the relevant categories, such as tasks, composition of interaction teams, or the possible combinations of single or multiple humans and agents [[YD04](#)].
- **Spatiotemporal concepts:** Physical objects in the environment, participants of interactions, and interactions themselves hold spatiotemporal information. These aspects of domain concepts need to be considered, for example the beginning and ending of an interaction (its temporal structure) is of high importance to the above stated competency questions. The *W3C Time Ontology* is reused for this purpose [[HP04](#)].

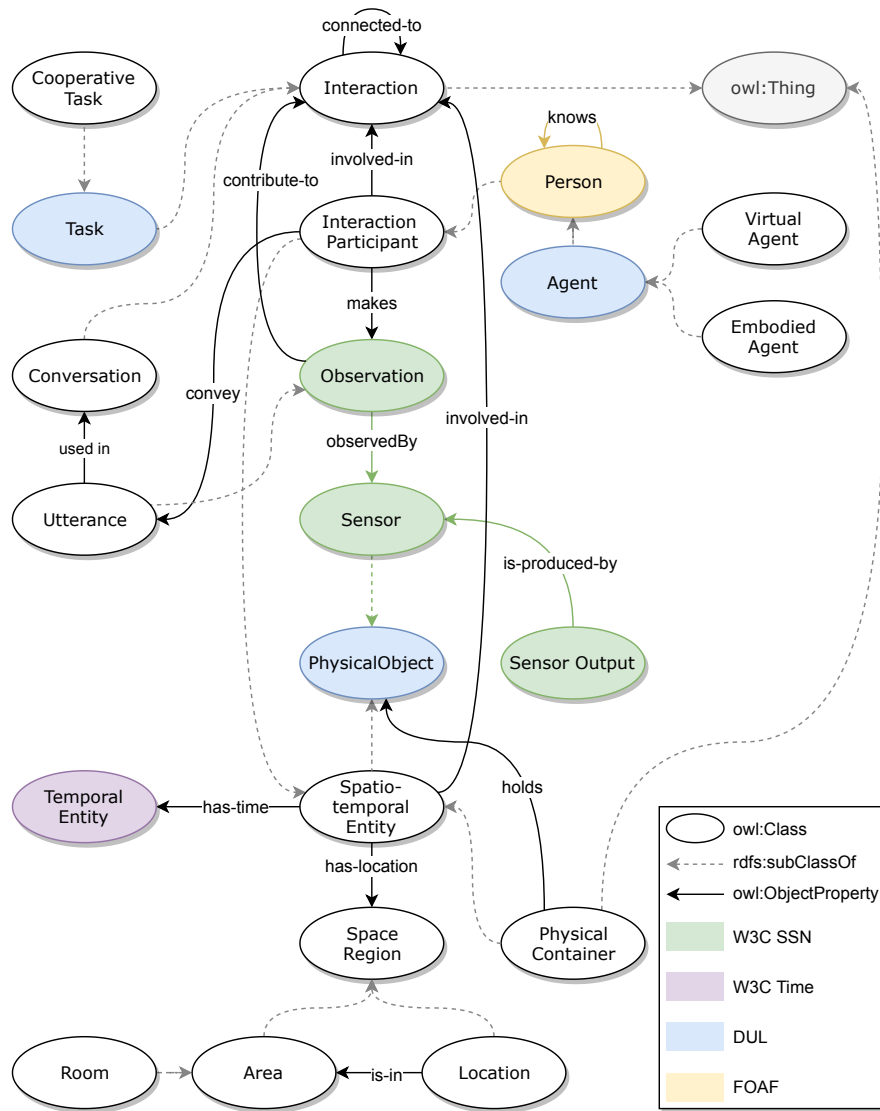


Figure 4.4: The *EISE* interaction ontology.

Figure 4.4 shows an overview of the top level concepts in the *EISE* ontology. The ontology is a model which allows to represent the scenarios described in Section 4.2.3 from the perspective of *behavior developers*. I emphasize on concept and representation reuse from existing ontologies to make use of existing detailed modeling efforts. The additionally added concepts, such as `InteractionParticipants` or `SpatiotemporalEntity`, embed the required classes into this structure.

The concept of `Persons` and their properties such as names and contact information are taken from the *FOAF* ontology. Artificial interaction Agents are chosen as a `subClassOf` `Persons` to enable their respective descendants `VirtualAgents` and `EmbodiedAgents` to be modeled identically to `Persons`. They thus are similarly capable to know other `InteractionParticipants`. These involved participants are in turn `involved-in` `Interactions`. their possible interconnection allows to

represent all configuration described in the *HRI* taxonomy by Yanco and Drury [YDo4]. The previously described scenarios revolve primarily around verbal interaction and thus Conversations are one exemplary lower level Interaction concept. Utterances conveyed by the individual InteractionParticipants are consequently used in such Conversations. Besides the shown Conversations, further extensions of this ontology can provide other types of Interactions describing other *HRI* scenarios. The temporal structure of PhysicalObjects (reused from the *DUL* ontology) is modeled via TemporalEntities attached to the intermediate SpatiotemporalEntity. The physical locations are abstracted by SpaceRegions more precisely Rooms, Areas, and Locations. The Sensor aspects of the *smart environment* and the EmbodiedAgent are grounded into the concepts of the *SSN* ontology. The individual Observations of these Sensors are of importance to the *behavior developers* and thus are modeled as such.

The ontology provides an initial abstraction of the concepts and relations of the domain and allows to represent the scenarios described in Section 4.2.3. Also the exemplary competency questions can be answered using this *model*. However, a central limitation of this domain *model* as an ontology lies in the eventual domain evolution and consequently the *model* evolution. With the *CSRA* domain being a research setting, it is composed of a rapidly changing hardware and software setup. One example of this change is the addition of dedicated tracking sensors to the system, which were introduced at a later stage of the project. Fundamental changes as such to the domain are problematic as they require a change to the domain *model*. This change also impacts the queries which users formulate towards the model: Changes to the ontology need to be transferred and queries need to be migrated to satisfy the new *model* layout.

4.6 SUMMARY

This chapter introduces the *EISE* domain as a combination of common *smart environments* and embodied cognition in *HRI*. As an example the *CSRA* project is described in which the contributions of this thesis are developed and used. The *CSRA* project is used for a domain analysis and the running application example for the followed *MDSE* approach. The central factors of the domain which are subsequently identified as:

- The system architecture in which tooling needs to be deployed and generate *artifacts* for,
- The data, information, and knowledge present in the system,
- The roles and responsibilities of individuals in the laboratory system,

- Required knowledge of each individual participating group,
- Exemplary knowledge queries towards the *EISE* domain.

Additionally, a brief presentation of a multi-modal interaction data corpus extracted from the *CSRA* is described. Lastly, the resulting ontology of interaction relevant knowledge is presented which was obtained in an early iteration of the development process. This ontology serves as the basis for the language engineering efforts in the following chapters: The *DSLs* and their composition need to be able to express the concepts of the ontology.

CONCEPTUALIZATIONS FOR MODEL-BASED QUERY COMPOSITION

“Nothing in life is to be feared, it is only to be understood. Now is the time to understand more, so that we may fear less.”

—Maria Skłodowska-Curie [Ben73]
Awarded the 1903 Nobel Prize in Physics

So far, the [Chapters 2 to 4](#) introduced the research topic, the adjacent domains, an analysis of the *Embodied Interaction in Smart Environments (EISE)* domain, and extracted a *model* abstracting interaction relevant knowledge of the *EISE* domain. Based on this foundational work, this chapter describes four central parts of the *Model-driven Software Engineering (MDSE)* process: 1) the underlying objectives and requirements (**RQ2**), 2) a technology-independent system architecture (**RQ3**), 3) a detailed language composition definition (**RQ3**), and 4) a suitable technology mapping (**RQ4**). I further present each language of the composition definition in detail, including individual implementation-independent *meta-models* and semantics of language intersections. Lastly, the technology mapping serves as a grounding in the application domain of the *Cognitive Service Robotics Apartment as Ambient Host (CSRA)* project. This mapping is based on the languages defined in this chapter as well as the findings of the research topic presentation in [Chapters 2 to 3](#).

The individual sections of this chapter are concerned with two different development process phases. On the one hand, [Section 5.1](#) extracts objectives as well as (non-)functional requirements and is thus part of the domain analysis in Phase **P1. Domain Analysis**. [Section 5.3](#), on the other hand, represents the results of Phase **P2. Language Design**, containing the implementation-independent system architecture and the technology mapping for the *EISE* domain.

Earlier iterations and parts of the conceptualizations in this chapter have previously been published by me and were peer-reviewed by the community. This primarily includes the publication “A Model Driven Approach for Eased Knowledge Storage and Retrieval in Interactive HRI Systems” presented during the 2018 Second IEEE International Conference on Robotic Computing (IRC) [KWC18a].

5.1 OBJECTIVES AND REQUIREMENTS

The objectives of the design process are influenced by two parts: a) the theoretical background regarding graphs and knowledge representa-

tion (cf. [Chapter 2](#) on page 11), as well as b) the background and state-of-the-art *MDSE* practices ([Chapter 3](#) on page 29). In their core, the objectives are then derived from the research questions in [Chapter 1](#) on page 3 and the domain analysis in [Chapter 4](#) on page 51. They frame the central question on *how* to provide *behavior developers* with an extensible *graph query language (GQL)* and other supporting tools. These tools are created following a *MDSE* approach, which allows developers to compose queries towards information and knowledge present in the *EISE* domain. The difficulty of this task lies in the choice of appropriate language design and composition to enable the inclusion of domain-specific user *models* and time constraints (e.g. via language patterns [[Pes+15](#)]), while keeping the individual languages extendable. The following objectives thus provide the grounds to formulate the (non-)functional requirements in the following [Section 5.1.1](#).

The first objective is *openness*, as the results of the *MDSE* approach need to integrate into the already present infrastructure and development process. This objective is based on the fact that a large portfolio of existing software, components, interconnection, and (sub-)systems are available, which need to be considered and included into the process from the beginning. On the one hand the domain's available knowledge and technical decisions, e.g. domain-specific data types or a common middleware, need to be considered in the process. On the other hand it is imperative that all generated *artifacts* can be used in the present ecosystem and infrastructure to foster developer acceptance and use. Only with this openness the usage of a graph query design tool for execution and analysis of queries is enabled.

The provided languages are required to *support the representation of domain-specific queries*. This includes an easy creation of queries grounded into the real world of the complex interactive system of the *EISE* domain, which can be constrained with respect to their temporal expansion.

Variations and changes of the *EISE* domain and the overall setup are plentiful due to the research setting and study oriented setup. This includes the addition or removal of sensors/actuators and any changes in the domain *model* or domain data types. *Extendability* and *versatility* are thus objectives to be considered during the *MDSE* process. *Domain-specific languages (DSLs)* need to provide a level of abstraction that allows to maintain, evolve, and extend user *models*. The application of appropriate language composition is central in the development process as it enables the required generalization, for example, by using language reuse, adapter languages, or orthogonal languages. Languages need to be build with support for easy future modifications and additions. The complexity of graph query design is then further reduced during the implementation phase via language *pragmatics* [[Rod15a](#)], such as query reduction/simplification of reoc-

curing patterns, domain-specific completion and suggestions (static checking), or user *model* and query analysis/profiling.

Closely tied to the versatility and query support lies the objective of *abstraction*: While the overall query constructs such as graphs, time representation, or graph pattern matching queries are part of the *M2* abstraction layer, other domain properties ideally reside in the layer of user *models M1*. As such, the domain description can either be statically implemented as a *meta-model* in *M2*, however, allowing domain experts to describe the current state of the domain themselves provides higher flexibility. Queries then depend on these user *models* and create instances of the domain concepts and their relations – grounding them into concepts of the real world.

5.1.1 Requirements

Based on the domain analysis of the project, the domain context and the identified objectives the following (non-)functional requirements are derived (**RQ2**).

5.1.2 Functional requirements

- FR1** Allow the representation of *Graph Database Management System (GDB)* queries
- FR2** Allow the execution of created queries towards a database
- FR3** Allow the creation of domain description *models* as a graph, allowing to abstract the concepts, relationships, and properties of the domain
- FR4** Allow to link the *GQL* and the domain description *model* to ground queries into the concepts of the real world
- FR5** Allow to express time constraints on *GQLs*
- FR6** Provide query feedback lifted from external analysis tools
- FR7** Interface with the *Application programming interfaces (APIs)*, middleware, and other software infrastructure of the domain
- FR8** Provide visual representations of (sub-)graphs

5.1.3 Non-functional requirements

- NFR1** Consider developer bias (e.g. system structure knowledge, preferred languages, query language knowledge)

- NFR₂** Formulate graph queries in a back-end independent query language
- NFR₃** Apply suitable language composition for easy language evolution and extension
- NFR₄** Provide a reproducible integrated language build and deployment solution
- NFR₅** Generate *artifacts* which integrate tightly with user *General Purpose Language (GPL)* code and the system of the domain

5.2 RELATED WORK

There exists only few applications in literature, which combine the domains of graphs, *GQLs*, robotics, *human-robot interaction (HRI)* and interactive environments. Proposed systems in the domain of *HRI* often utilize knowledge-based systems, knowledge processing, ontologies, or other frameworks providing reasoning capabilities, such as *KnowRob* or *ORO* [TB13; Bee+18; Lem+10]. These examples make use of inference engines to provide reasoning and logical deduction for complex problem solving, such as robot motion planning, common-sense grounding of actions, semantic annotation, or memory management. Querying in these applications is generally realized by existing query languages such as the tuple based *SPARQL Protocol and RDF Query Language (SPARQL)* or logic programming language *Prolog*.

The most prominent approach focusing on query design support for such systems is presented by Balint-Benczedi et al. who provide a storage and retrieval *DSL* for robotic episodic memories [Bal+17]. The underlying ontology *model* is concerned with data regarding robot perception and especially data relevant for long-term manipulation tasks. It is embedded and part of the *KnowRob* system and a direct reaction to the complex *Prolog* queries emerging within this system [TB13; Bee+18]. Retrieval of episodic memories is thus eased via an object and scene description language. As such, it serves as an abstraction layer between the structure of the perception of the robot and the semantic interpretation of observations. The implemented dedicated query interface is used to retrieve specific elements of the episodes and realizes two central goals: 1) enable on-line retrospection and specialized training of perception routines and 2) enable researchers to interactively explore perception results. Architecturally, the authors store the raw sensor data as unstructured information within a *MongoDB* document store. This database already provides a specific query language that follows the syntax of the data description format *JSON* and the central contribution of the authors is the addition of predicates into this language. The predicates are based

on the existing description language, which abstract from the data structure in the underlying ontology. Users of this internal *DSL* need to be familiar with the host description language and the additional predicates for successful query design: No further query design support, such as completion, query analysis, or other tooling is provided in this approach. The authors also identify temporal properties of the data as an important factor. As a result, they expose access to the annotated timestamps of data and queries can be constrained via absolute time information. Again, the complexity of proper temporal query design is left to the user, who need to directly insert timestamps into temporally constrained queries. Further, evaluation of the approach is solely anecdotal by example and implementation showing 1) reduced (*Source*) *Lines Of Code (LOC)* when compared to the usual query constructs using native database queries and 2) a realization of recognition classifier training sample collection at runtime using the *DSL*.

Dietrich et al. present another internal *DSL* which supports robotic world knowledge retrieval [DZK15]. The introduced language *SelectScript* adopts the semantics of *Structured Query Language (SQL)* and extends the language with domain-specific features. The authors provide a language with a reduced expressiveness for effective querying within developer code. The implementation was created using *ANTLR* and hence detailed descriptions on the language's grammar are given. While they provided features for continuous queries (by executing existing ones every 100 milliseconds), no detailed modeling of temporal concepts or the domain concepts were described. As a result, *SelectScript* requires developers to fall back their domain knowledge and the use of timestamps to express domain-specific temporally constraint queries.

5.3 SYSTEM ARCHITECTURE

The technology-independent system architecture defines all concepts, which are available to create a system [Völö6]. It contains all technologies and approaches useful to represent, explain, and illustrate the architecture intentions. According to Völter, there is no formalized way of representing technology-independent architectures: It is composed from box and line diagrams, state/sequence/activity charts, textual explanations, and anything helpful to communicate the architecture. Contributing towards research question **RQ3**, I present the system architecture by considering the different requirements as well as perspectives from *behavior developers* of the CSRA.

Figure 5.1 shows my proposed technology-independent structure diagram that contains a) different targeted *IDEs* and tooling which developers utilize (**NFR1**, **NFR3**, **NFR4**), b) the structural integration of the *IDE* (**FR7**, **NFR4**), c) the individual developers and their roles

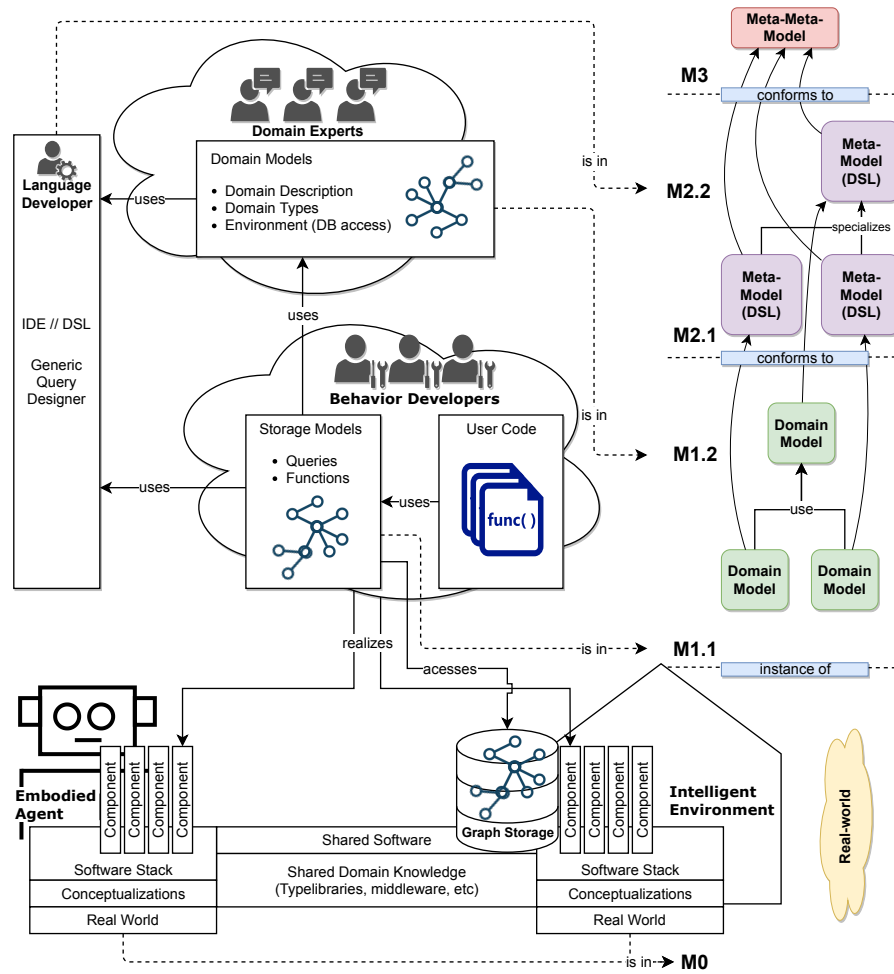


Figure 5.1: Diagram of the structural integration of the target *integrated development environment (IDE)* into the overall system, the involved user roles, user *models*, and a mapping to the underlying real world elements. Additionally, *models* and tool elements are related to the corresponding *Object Management Group (OMG)* meta-modeling layers where applicable (cf. Figure 3.1 on page 31).

in this structure (FR7), and lastly d) a mapping of user *models* to the corresponding individual layers of the *OMG* meta-modeling layers and real world entities.

At the core of the system lies the real world represented by the intelligent environment and the autonomous embodied agent acting in it (bottom). Each of these systems is composed by separated software components carrying out individual tasks. The tasks operate on various abstraction levels of the system, for example, sensor data provisioning to middleware or person tracking on available data (cf. the *data-information-knowledge-wisdom (DIKW)* hierarchy in Section 2.1). While both these systems rely on their own conceptualizations (e.g. data types, and communication patterns) and own software, they also share domain knowledge, abstractions and (connected) software.

The architecture provided in my work provides a generic query designer *IDE* for *behavior developers* (left). The languages and conceptualizations of this tool conform to the M_3 layer and are realized as *meta-models* in M_2 . With the help of this tool and the included *DSLs*, domain experts create the three different types of user *models* for the *EISE* domain: a) a domain description *model* composed of all concepts and relations of importance to the *behavior developers* (**FR3**), b) a *model* of domain types based on the shared and individual conceptualizations of the domain (**NFR2**), and c) other *models* containing required environment specific knowledge such as database access or shared middleware properties (**FR7**, **NFR1**, **NFR2**).

The created *models* in combination with the generic query design *IDE* provide a highly domain-specific query design tool: The *EISE Query Designer (EISEQD)* (center). All queries that are created in the *EISEQD* can be grounded into the domain via the dependent domain description, domain types and environment information. This tool is used by *behavior developers* who are responsible for components of the system and within its environment, each user can compose specific *models* that contain *graph database queries (GDQs)* and user functions for their specific use case. In the traditional workflow, developers compose the query strings and place a copy within their source code. In contrast to this, queries are constructed in the *IDE* and can directly be tested and profiled in the environment. *Artifacts* are generated from the *IDE* and embedded in the developer code (e.g. as library dependencies packaging the designed queries). In the last step following the query design phase, suitable deployment strategies install the resulting components and *model artifacts* into the software stack of the environment(s) (**NFR4**, **NFR5**). As a result, similarly to the domain description and domain types, the query *models* of developers exist in the M_2 layer. Changes to the domain *model* are consequently directly reflected in the user queries at design time. The developer code, however, is stable as it references the functions of the generated *artifacts* and thus does not need to be updated upon every query change. Suitable language composition is required allow this type of composition and ultimately enable evolution of the domain (**NFR3**).

5.4 EXTENSIBLE GRAPH QUERY LANGUAGE MODULARIZATION AND COMPOSITION

Language composition and modularization has been identified as a central necessity for *DSL* development [**Völ13a**; **Com17**; **Pic10**; **Erd+13**]. Suitable language composition for this work is also a requirement (**NFR3**) to implement tooling that allows to integrate in the previously presented system architecture. Additionally, many of the named advantages – such as reuse, or extendability (cf. **Chapter 3** on page 29) – require successful language dependency organization. **Figure 5.2**

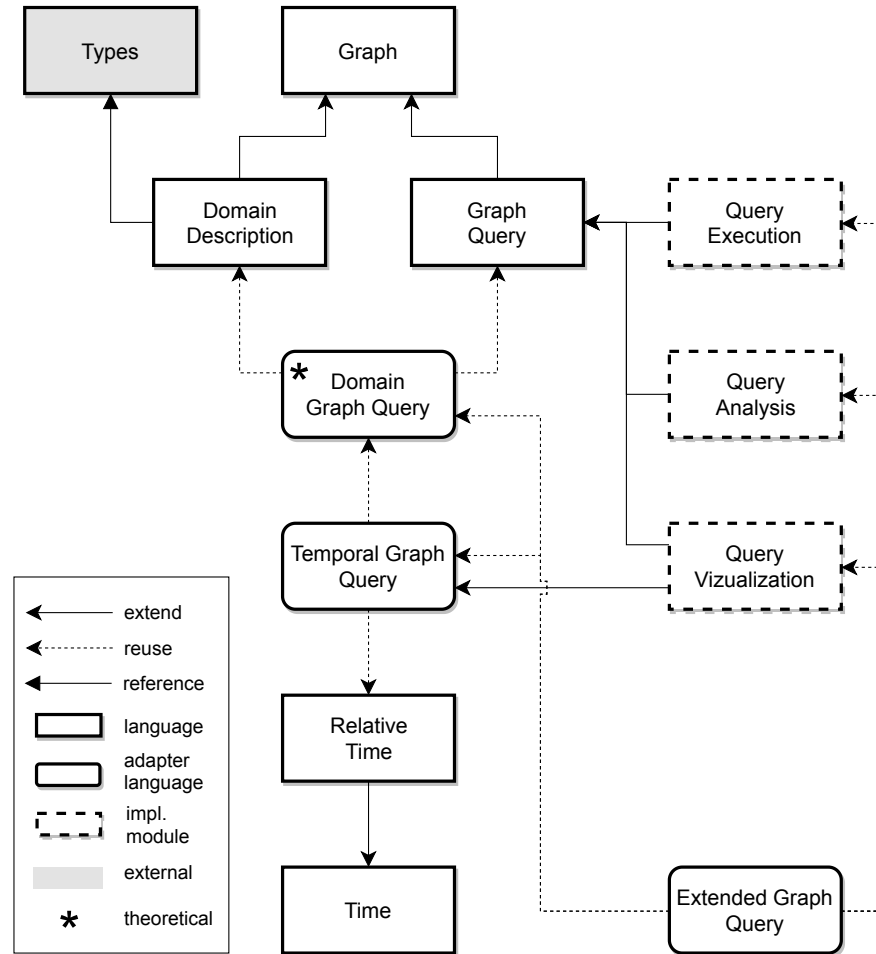


Figure 5.2: The proposed language modularization and composition.

thus shows the implementation-independent language modularization and composition applied in this thesis. At its core, the Graph language resides with no external dependencies. This language represents the abstractions required to represent *labeled property multidigraphs* as introduced in Section 2.2 on page 14. As such, it allows to represent the elements of a graph: Nodes and Relationships, as well as the respective Properties and Labels. This foundation is the basis for the *extending* Graph Query language, which *embeds* the Graph concepts and allows to describe matching pattern graph queries via the PatternQuery top-level concept. The Domain Description is the second depending language and *reuses* (i.e. depends and references) the Graph language. It abstracts from the description of a domain and allows to create two top-level concepts:

- a) DomainDescriptionGraphs: A concept to represent all elements of a domain, their relations, as well as element properties respectively.

- b) `DomainInstanceGraphs`: A Graph as defined by the Graph language that additionally allows to ground graph elements via referencing to elements of a given `DomainDescriptionGraph`¹

The Domain Description language additionally *embeds* a Type language to enable domain-specific type incorporation. The adapter language Domain Graph Query combines the features provided by the Domain Description and Graph Query languages without providing any further concepts or extensions itself. The combination of domain descriptions and graph queries in user *models* is practically enabled by the joint dependencies to the common Graph language. The Temporal Graph Query adapter language uses the conceptualizations of the domain description and provides the feature to explicitly constrain graph queries with respect to their temporal expansion (**FR5**). It therefore reuses the Relative Time language and applies orthogonal *language composition* onto the Domain Graph Query language. The Relative Time language is an extension of the Time language and enables the representation of time relative to another point in time, for example, temporal constraints relative to the query execution. Language *pragmatics*, such as Query Execution, Query Visualization, or Query Analysis primarily make use of *reference* and *extension* capabilities to enrich the language architecture with their respective features. Depending on the chosen *language workbench* and the availability of features (e.g. projection editing), the *pragmatics* can provide, for example, different *concrete syntaxes*, alternative projections, or lift analysis information back into the *IDE*. Further extensions can be created using this proposed composition mechanism (**NFR3**), even in later iterations of the process.

The following sections will present each language in detail. With the language composition in [Figure 5.2](#) showing a clear overview on how the languages are organized and related, the following individual *meta-models* explain the syntax of each individual language. I make use of *Unified Modeling Language (UML)* based *meta-model* diagrams due to the increased intuitiveness, pragmatic representation, and elegance [**HR00**]. However, this detailed view on concepts and representation of languages does not fully describe the complete language behavior: The *meta-models* are implementation-independent descriptions of the syntax, and thus it is necessary to clearly describe the intended language behavior, especially at its intersections, to fully capture the meaning behind the conceptualizations. Therefore, I present additional semantic descriptions of the language behavior (cf. [Section 3.1.2](#) on page 31). Whenever language intersections and composition cannot be explained sufficiently by the provided *abstract syntax* and *meta-models*, I provide the *denotational semantics* of the languages. These semantics are intended to denote the language

¹ Elements of the `DomainInstanceGraph` are thus “instances” of their referenced pendants in the `DomainDescriptionGraph`

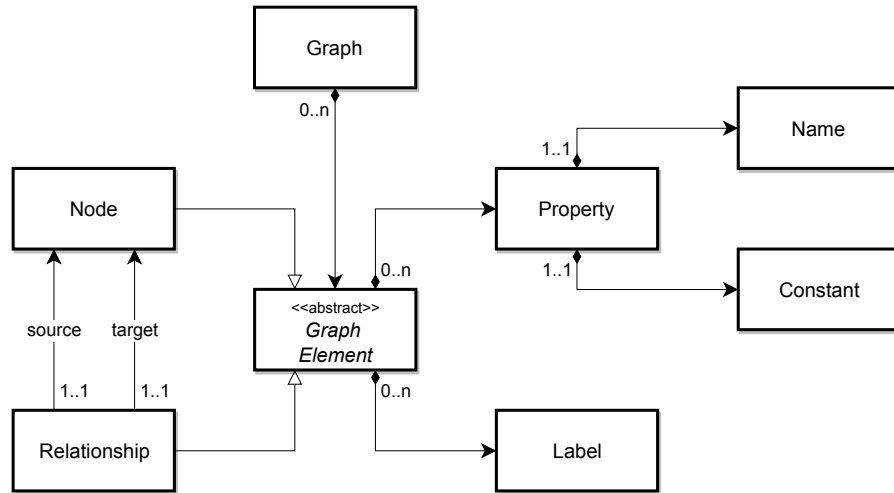


Figure 5.3: The *meta-model* of the Graph language which allows to represent a *labeled property multidigraph*.

behavior and also provide the resulting behavior applied during and after *artifact* generation.

5.4.1 Representation of graphs

Figure 5.3 shows the *meta-model* of a graph, which is a central language in the composition. This representation allows to describe *labeled property multidigraphs* as introduced in Section 2.2 and as a result, a graph represented by this *meta-model* corresponds to a graph of the form $G = (V, E, \rho, \lambda, \sigma)$ as described in Equation (2.6) on page 16. At the top-level of the abstraction lies the Graph itself. It contains a number of Nodes and Relationships. Each of these GraphElements can hold multiple Properties consisting of a Name and an assigned Constant value. Labels are individually held by Nodes and Relationships alike, representing the multi-graph characteristics of the graph model. Relationship direction is expressed via two distinct references from a Relationship to a source and a target Node. This *meta-model* is the foundational abstraction to be reused by the GQL in the technology mapping (FR₁ - FR₂, NFR₂).

5.4.2 Representation of pattern matching queries

Figure 5.4 on the next page depicts a non-exhaustive *meta-model* of a pattern matching read-only query and the relations to the dependent Graph language². A PatternQuery is one possible type of graph query which consists of three core components: 1) MatchingClause, 2) FilterClause, and 3) ResultClause. The concepts Patterns and Pattern-

² This *meta-model* focuses on simplified pattern matching queries on graphs. The detailed role of graph traversal queries are not considered in this thesis.

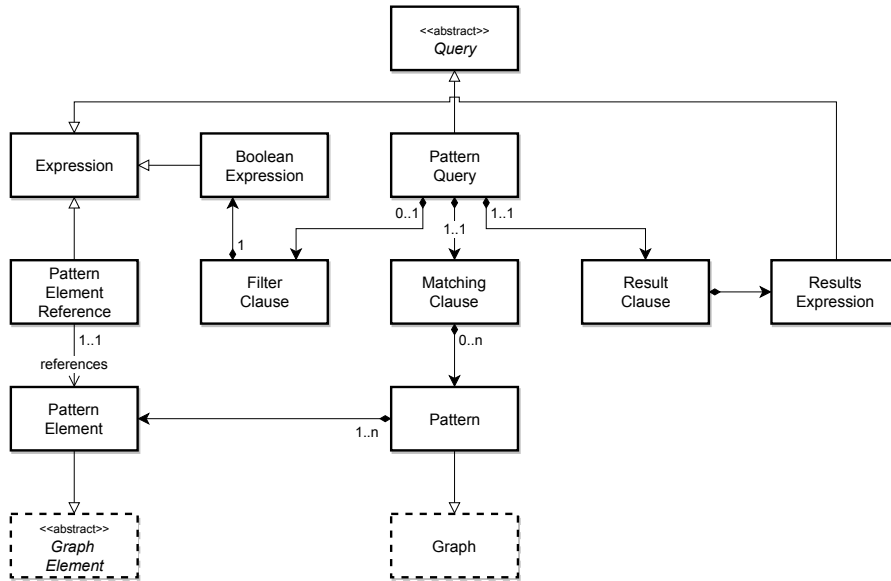


Figure 5.4: A simplified *meta-model* of a pattern based query language. Dashed concepts are part of the dependent Graph language.

Elements of Graph Query language consequently specialize the concepts Graph and GraphElements respectively. PatternElements can be referenced within the filter and result clause to restrict the query on the properties or labels of the graph. A simple pattern matching read-only graph query Q consists of the three elements matching pattern M , constraint C , and result aggregation R . Each matching pattern contains a set of Patterns P which are matched against the stored graph. An individual Pattern is composed of PatternElements PE , which individually can represent a Node or a Relationship.

$$Q = (M, C, R) \quad | \quad M \in G \quad (5.1)$$

$$M = (P_1, \dots, P_n) \quad (5.2)$$

$$P_i = (PE_1, \dots, PE_m) \quad | \quad \forall P_i \in M \quad (5.3)$$

In essence, M represents a graph that is composed of the individual contained patterns P_i , which are described as an ordered list. This choice is inspired by the Cypher semantic: Each individual Pattern of a MatchingClause is a linear graph chain (e.g. A-B-C and D-B-E) and the combination of all patterns composes the full matching graph. To further illustrate, when using Cypher as the target GQL , the semantics transforms the query to a three clause statement such that the *denotational semantics* in Equations (5.4) to (5.6) operate. This rationale is based on the *Extended Backus–Naur Form (EBNF)* grammar of the Cypher language [Neo15], as shown in the example in Listing 5.1 on the following page, taken from the official grammar³. I chose this

³ Also compare to the full resources provided at <https://github.com/opencypher/openCypher>

```

122 Where = (W,H,E,R,E), SP, Expression ;
123
124 Pattern = PatternPart, { [SP], ',', [SP], PatternPart } ;
125
126 PatternPart = (Variable, [SP], '=', [SP], AnonymousPatternPart)
127 | AnonymousPatternPart
128 ;
129
130 AnonymousPatternPart = PatternElement ;
131
132 PatternElement = (NodePattern, { [SP], PatternElementChain })
133 | ('(', PatternElement, ')')
134 ;
135
136 NodePattern = '(', [SP], [Variable, [SP]], [NodeLabels, [SP]],
137 ↪ [Properties, [SP]], ')' ;
138
139 PatternElementChain = RelationshipPattern, [SP], NodePattern ;
140
141 RelationshipPattern = (LeftArrowHead, [SP], Dash, [SP],
142 ↪ [RelationshipDetail], [SP], Dash, [SP], RightArrowHead)
143 | (LeftArrowHead, [SP], Dash, [SP], [RelationshipDetail], [SP],
144 ↪ Dash)
145 | (Dash, [SP], [RelationshipDetail], [SP], Dash, [SP],
146 ↪ RightArrowHead)
147 | (Dash, [SP], [RelationshipDetail], [SP], Dash)
148 ;
149
150 RelationshipDetail = '(', [SP], [Variable, [SP]], [RelationshipTypes,
151 ↪ [SP]], [RangeLiteral], [Properties, [SP]], ')' ;

```

Listing 5.1: Excerpt from the official openCypher *EBNF* definition showing an example on how MatchingClauses are defined in the official query language.

minimalistic example as it clearly shows the similarity of the denotational graph semantics presented in this chapter within the openCypher grammar. Additionally, it is important to note that queries – especially MatchingClauses – can possibly be more complex than what is shown here. Depending on the query details and result aggregation, the query can also yield a path as a result. For example, MatchingClauses can contain increased details such as sub-clauses, entire sub-queries, or optional keywords. Another example are PatternElements, which can be composed as complex as the desired by the users and include Nodes, Relationships, references to Relationships, references to Nodes, or references to other Match clauses. The decomposition in the *denotational semantics* provided here are chosen to keep compatibility to the aforementioned *EBNF*. Consequently, these semantical the abstractions I provide here are as direct and unambiguously as possible and are directly compatible to the implementation as presented in [Chapter 5](#) on page 75.

Matching patterns are further destructured by the Graph Query language semantics into individual pattern elements PE. Each element

of the linear pattern chain is further reduced using Cypher semantics as shown in [Equations \(5.5\) to \(5.11\)](#).

$$\begin{aligned} \llbracket (M, C, R) \rrbracket_{GQ} &= \text{MATCH } \llbracket M \rrbracket_{GQ} \\ &\quad \text{WHERE } \llbracket C \rrbracket_C \\ &\quad \text{RETURN } \llbracket R \rrbracket_C \end{aligned} \quad (5.4)$$

$$\llbracket M \rrbracket_{GQ} = \llbracket P_1 \rrbracket_{GQ} \oplus \dots \oplus \llbracket P_n \rrbracket_{GQ} \quad (5.5)$$

$$\llbracket P_i \rrbracket_{GQ} = \llbracket PE_1 \rrbracket_C \dots \llbracket PE_m \rrbracket_C \quad | \quad PE_k \in P_i \quad (5.6)$$

For the distinction of different cases in [Equation \(5.7a\)](#) each PatternElement is seen as its abstract definition, i.e. the tuple of an identifier y , the node v or edge u itself, the labels or type L , and the corresponding set of attributes A^4 .

$$PE_{k-1} \llbracket PE_k \rrbracket_C PE_{k+1} = \left\{ \begin{array}{ll} (y: \llbracket L \rrbracket_C \llbracket A \rrbracket_C) & | \text{ if } PE_k = (y, v, L, A) \wedge \\ & v \in V \quad (5.7a) \\ \text{-}\llbracket y: \llbracket L' \rrbracket_C \llbracket A \rrbracket_C \rrbracket \text{-} & | \text{ if } PE_k = (y, e, L, A) \wedge \\ & e \in E \wedge \\ & PE_{k-1} = (_, u, _) \wedge \\ & PE_{k+1} = (_, v, _) \wedge \\ & u, v \in V \wedge \\ & \rho(e) = (u, v) \wedge \\ \text{<}\llbracket y: \llbracket L' \rrbracket_C \llbracket A \rrbracket_C \rrbracket \text{-} & | \text{ if } PE_k = (y, e, L, A) \wedge \\ & e \in E \wedge \\ & PE_{k-1} = (_, u, _) \wedge \\ & PE_{k+1} = (_, v, _) \wedge \\ & u, v \in V \wedge \\ & \rho(e) = (v, u) \wedge \quad (5.7c) \end{array} \right. \quad (5.7b)$$

$$\llbracket A \rrbracket_C = \llbracket (p_1 = c_1, \dots, p_n = c_n) \rrbracket_C \quad (5.8)$$

$$\llbracket (p_1 = c_1, \dots, p_n = c_n) \rrbracket_C = \{p_1:c_1 \oplus \dots \oplus p_n:c_n\} \quad (5.9)$$

$$\llbracket L \rrbracket_C = \llbracket l_1 \dots l_m \rrbracket_C = l_1 \oplus \dots \oplus l_m \quad (5.10)$$

$$\llbracket L' \rrbracket_C = \llbracket l_1 \dots l_o \rrbracket_C = l_1 \oplus_1 \dots \oplus_1 l_o \quad (5.11)$$

Depending on the context around any present PatternElement PE_k , each individual element is constructed to Cypher concrete syntax and represents:

- Nodes in [Equation 5.7a](#) ⁵,

⁴ Attributes are properties. I use A to avoid name clashes in the equations

⁵ Includes PatternElements which have no left and/or right neighbor, i.e. $PE_{i\pm 1} \in \epsilon$

- Right directed Relationships in Equation 5.7b, or
- Left directed Relationships in Equation 5.7c.

The relationship direction is conserved by falling back to the relationship direction information in $\rho(e)$. Lastly, in Equations (5.10) to (5.11) the Labels and Attributes of each GraphElement are reduced to the list representation using the corresponding separators and syntactical extras such as brackets defined in the Cypher language.

5.4.3 Representation of domain descriptions

The Domain Description language is used to represent the *EISE* domain within the queries (FR3). This language aggregates the knowledge of the domain concepts and their relations as a user *model* instead of as a *meta-model*. The language is designed so that the domain concepts modeled in a domain *model* can be instantiated and referenced within a *GQL*, subsequently grounding the query in the domain. This grounding functions similarly to a schema known in relational databases, however the domain *model* is a *model* created by domain experts themselves and developer queries directly link to it. This allows to easily maintain and evolve domain representations according to the real world while also propagating the changes to all

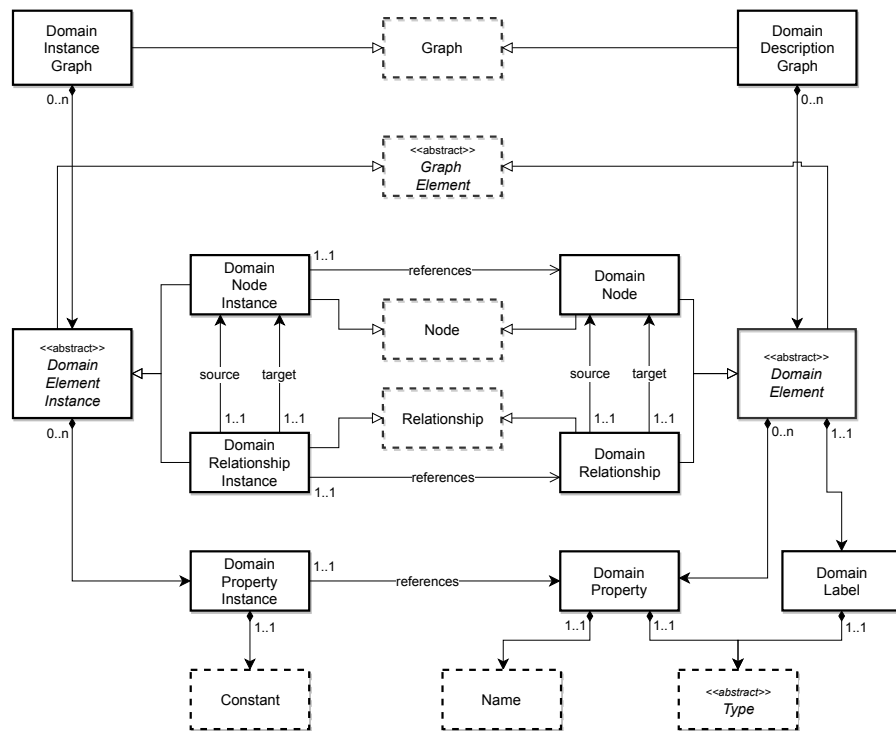


Figure 5.5: The *meta-model* of the Domain Description language. Dashed concepts are part of the dependent Graph and Type languages.

corresponding queries (without the need to update a *meta-model* by language designers).

Figure 5.5 on the facing page shows the proposed *meta-model* of the Domain Description language. The language depends on the previously presented graph language (cf. Figure 5.3). It reuses (i.e. combination of referencing and extension) the graph language and provides two top-level concepts which specialize a Graph: 1) DomainDescriptionGraph and 2) DomainInstanceGraph.

The former DomainDescriptionGraph is the specialized graph used to represent the domain concepts and their relations by using DomainNodes and DomainRelationships. DomainElements can contain any number of DomainProperties. These properties differ from the Property concept in the Graph language such that instead of a constant value, a Type from the available Type language(s) is provided. The DomainLabels also differ from the traditional *labeled property multigraph* definition as their cardinality is reduced to one, thus increasing the model specificity. The possibility to reference a Type for each DomainLabel allows to further ground the type of DomainElements to already present types in the Type language or domain-specific Type specializations available in the domain.

In contrast to this, the DomainInstanceGraph (which is technically also a graph) allows to define DomainElementInstances, which specialize the already existing GraphElements. DomainElementInstances are intended to be used within matching patterns of a GQL and conceptually represent individual instances of the anonymous abstract DomainElements. The specializations DomainNodeInstance, and DomainRelationshipInstance thus use the common GraphElement features and additionally require a reference to an existing DomainNode or DomainRelationship respectively. This required reference represents the grounding of a DomainElement via reference to the concepts defined in a DomainDescription (FR4). The DomainPropertyInstance concept held by DomainElements similarly provides the means to represent a specific instances of a DomainProperty. In contrast to the Graph language, the DomainPropertyInstances contain a constant value and their name and type is derived from the referenced DomainProperty. At the same time, non-grounded properties can be specified as the DomainElementInstance specializes a GraphElement which in turn provides the “anonymous” Property concept.

Semantically, a DomainDescriptionGraph DDG operates similarly to the previously shown graph (cf. Section 5.4.1 on page 84). However, for each edge the number of labels $|\lambda(e)|$ is limited to one for this graph.

$$\begin{aligned} \llbracket \text{DDG} \rrbracket_{\text{DD}} = G & \quad | \text{ where } G = (V, E, \rho, \lambda, \sigma) \\ & \quad \forall e \in E : |\lambda(e)| = 1 \end{aligned} \quad (5.12)$$

Analogously to the semantics for graph queries in [Equation \(5.4\)](#) on page 87, the semantics of DomainElementInstances embedded within a graph query Q^{DD} are described by the following statements:

$$Q^{DD} = (M^{DD}, C, R) \quad (5.13)$$

$$\begin{aligned} \llbracket (M^{DD}, C, R) \rrbracket_C &= \mathbf{MATCH} \llbracket M^{DD} \rrbracket_{DGQ} \\ &\quad \mathbf{WHERE} \llbracket C \rrbracket_C \\ &\quad \mathbf{RETURN} \llbracket R \rrbracket_C \end{aligned} \quad (5.14)$$

$$\llbracket M^{DD} \rrbracket_{DGQ} = \llbracket P_1^{DD} \rrbracket_{DGQ} \oplus \cdots \oplus \llbracket P_n^{DD} \rrbracket_{DGQ} \quad (5.15)$$

$$\llbracket P_i^{DD} \rrbracket_{DGQ} = \llbracket PE_1^{DD} \rrbracket_{DGQ} \cdots \llbracket PE_m^{DD} \rrbracket_{DGQ} \quad (5.16)$$

The Pattern P_i^{DD} listed in [Equation \(5.16\)](#) represents the elements of the MATCH clause M such that $P_i^{DD} \in M$. A key difference to the default semantics in [Equations \(5.4\) to \(5.6\)](#) on page 87 is the change that the PatternElements PE^{DD} of each Pattern P_i^{DD} within the set of provided Pattern in M can be a DomainElementInstance ([Equations \(5.14\) to \(5.15\)](#)).

$$PE_{k-1} \llbracket PE_k^{DD} \rrbracket_C PE_{k+1} = \left\{ \begin{array}{ll} (y:\llbracket L \rrbracket_{DGQ} \llbracket A \rrbracket_C) & | \text{ if } PE_k = (y, v, L, A) \wedge \quad (5.17a) \\ & v \in V(\llbracket DDG \rrbracket_{DGQ}) \\ \neg(y:\llbracket L \rrbracket_{DGQ} \llbracket A \rrbracket_C) \rightarrow & | \text{ if } PE_k = (y, e, L, A) \wedge \quad (5.17b) \\ & e \in E(\llbracket DDG \rrbracket_{DGQ}) \wedge \\ & PE_{k-1} = (_, u, _) \wedge \\ & PE_{k+1} = (_, v, _) \wedge \\ & u, v \in V(\llbracket DDG \rrbracket_{DGQ}) \wedge \\ & \rho(e) = (u, v) \wedge \\ \neg(y:\llbracket L \rrbracket_{DGQ} \llbracket A \rrbracket_C) \leftarrow & | \text{ if } PE_k = (y, e, L, A) \wedge \quad (5.17c) \\ & e \in E(\llbracket DDG \rrbracket_{DGQ}) \wedge \\ & PE_{k-1} = (_, u, _) \wedge \\ & PE_{k+1} = (_, v, _) \wedge \\ & u, v \in V(\llbracket DDG \rrbracket_{DGQ}) \wedge \\ & \rho(e) = (v, u) \wedge \\ \llbracket PE_k \rrbracket_C & | \text{ else} \quad (5.17d) \end{array} \right.$$

$$\llbracket L \rrbracket_{DGQ} = \llbracket l_1 \rrbracket_C = l_1 \quad (5.18)$$

As a result, the individual PatternElements PE_k^{DD} (i.e. either Node or Relationship), that make use of the domain description specializations DomainNodeInstance or DomainRelationshipInstance concepts, are reduced corresponding to [Equations 5.17a to 5.17c](#). Their refer-

ence to the corresponding `DomainNode` and `DomainRelationship` is expressed in the corresponding use of target language syntax which expresses a Cypher node or relationship respectively. Similarly to the default semantics for `PatternElements` of the Cypher language as listed in Equation (5.7a) on page 87, the semantics for queries with domain knowledge distinguish nodes and relationships. `PatternElements` are still seen as a tuple of an identifier y , the node v or relationship u , the labels or type L , and the corresponding set of attributes A ⁶. The central difference of the semantics is that domain description information (i.e. the node/relationship reference) is used as the node label or relationship type L respectively. Identifier and properties are transferred analogously, while relationship direction is conserved by falling back to the relationship direction information $\rho(e)$ of the domain description graph. In case PE_k is not a concept related to the domain description, the default Cypher semantics are applied (Equation 5.17d) as described by Equation (5.7a) on page 87.

5.4.4 Representation of time

As identified in the domain analysis, the ability to express temporal properties of queries is an important factor in the query design process (FR5). Queries on domain knowledge are often formulated with temporal properties relative to their execution time [All84; TB09; Bal+17] and are thus treated specifically in this thesis. For example, a query can target information from within the last n seconds (i.e. an interval starting n seconds ago until now), or at exactly n seconds ago (i.e. a point in time which lies exactly n seconds in the past).

The time languages are thus constructed to allow to formulate time constraints and attach them to queries or its elements. I separate this task into three languages (NFR1,NFR3): 1) Time 2) Relative Time, and 3) Temporal Graph Query.

Figure 5.6 depicts the proposed *meta-model* of the Time language, which provides the fundamental capabilities to represent any point in time or temporal expansion. I base the language design on the *Time Ontology in OWL* as presented by the *World Wide Web Consortium (W3C)*, which gathers the core temporal classes, their topology, and principles (cf. Section 4.5.4 on page 70) [W3C17; HP04]. At the language core the `TimeDescription` concept represents temporal entities, which are expressed either as `Interval` or as `Instant` concepts. An `Interval` can have two distinct forms and is either a `DurationInterval` consisting of an `Instant` and a `Duration` referencing the start and duration of the interval, or an `InstantInterval` which holds two `Instants` referencing the start and end of the interval. A time `Instant` itself is a precise `TimeDescription`, which holds all required

⁶ Attributes are properties but to avoid naming clashes in the equations the identifier A is used here

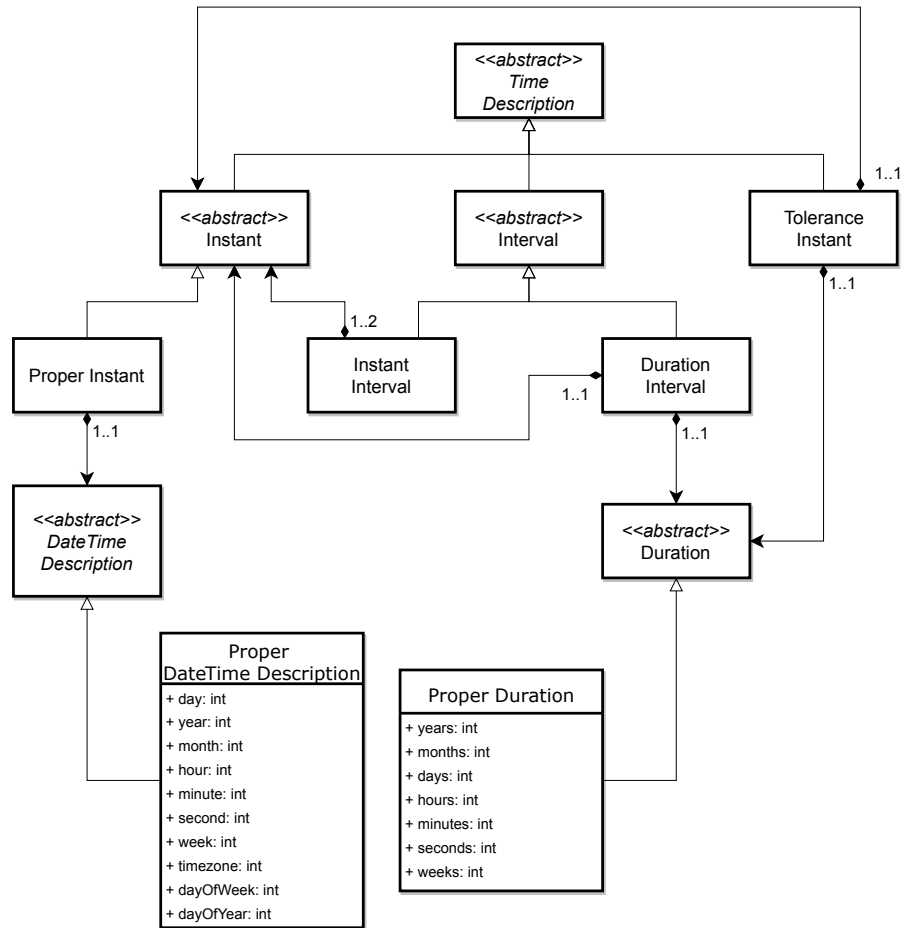


Figure 5.6: The meta-model of the Time language.

time related elements to represent a certain point in time. To allow the description of a point in time with a given tolerance, I further add the ToleranceInstant. This concept specializes a TimeDescription by holding an Instant to represent a point in time and a Duration representing the tolerance around this point (NFR₁). Though this ToleranceInstant could also be expressed with a corresponding Interval and suitable *concrete syntax* or language *pragmatics*, I chose this explicit representation to clearly formulate tolerances in the model.

The Time language is independent from the other languages of this approach (FR₄) and I further specialize temporal descriptions in the Relative Time language depicted in Figure 5.7 on the next page. The goal of this language in the overall language composition is to allow the representation of temporal constructs relative to a temporal reference point. The addition of the RelativeTimeDescription concept provides the possibility to express these relative time constructs by being an Instant specialization, which also holds an additional Anchor and Offset. This description of a point in time with a given offset

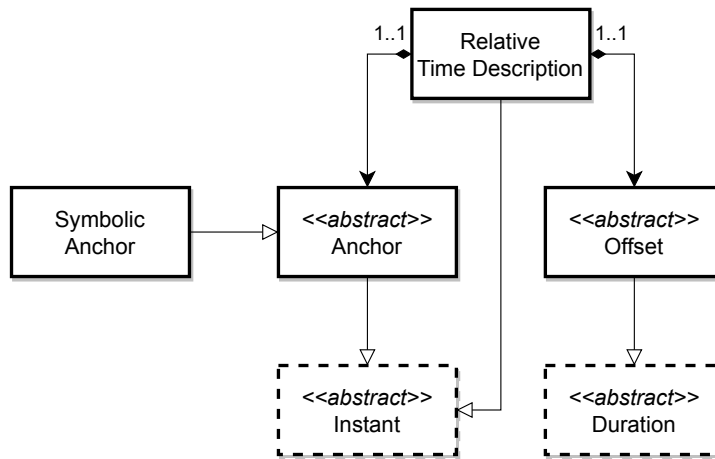


Figure 5.7: The *meta-model* of the Relative Time language. Dashed concepts are part of the dependent Time language

allows to create TimeDescriptions that are relative to a given point in time (e.g. the query execution time).

Semantically, the Time and Relative Time languages provide corresponding expected behavior. Every TimeDescription is reducible to either a single point in time (i.e. an Instant) or a set of multiple points in time (i.e. an Interval). For the usage in the *EISE* domain, I chose a point in time such as a ProperInstant reduces to the equivalent *POSIX Time* representation⁷:

$$\llbracket \text{PI} \rrbracket_{\text{RT}} = \llbracket \text{PI} \rrbracket_{\text{T}} = \llbracket \text{PI} \rrbracket_{\text{POSIX}} \quad (5.19)$$

Further, a RelativeTimeDescription RTD adds an Instant specialization which is denoted by the tuple of an Anchor *A* and an Offset *OFF*, which are semantically reduced to a joint ProperInstant based representation using common time arithmetics [All84]:

$$\begin{aligned} \llbracket \text{RTD} \rrbracket_{\text{RT}} &= \llbracket (A, \text{OFF}) \rrbracket_{\text{RT}} \\ &= (\llbracket A \rrbracket_{\text{RT}} + \llbracket \text{OFF} \rrbracket_{\text{T}}) \\ &= (\llbracket \text{PI} \rrbracket_{\text{T}} + \llbracket \text{OFF} \rrbracket_{\text{T}}) \end{aligned} \quad (5.20)$$

Figure 5.8 and Figure 5.9 on the following page depict exemplary temporal expressions which are expressible using the above presented languages. The Time language provides four absolute temporal constructs users can use to express temporal expansions. Figure 5.8 on the next page shows these four types and the concepts which are used to construct each temporal expansion: A ProperInstant (PI) allows to represent a single point in time, while the extended version, a ToleranceInstant (TI) describes a point in time with a given tolerance around it (technically representing an Interval). Intervals can either be based on given start and end Instants (II) or based

⁷ The number of seconds since January 1, 1970 midnight +00:00, minus leap seconds



Figure 5.8: Timeline of all four absolute temporal expressions expressible using only the Time language. The indicated involved concepts are named after the initials of concepts shown in Figure 5.6 on page 92. Normal lines indicate properties defined by the user, dashed lines are the resulting temporal boundaries.

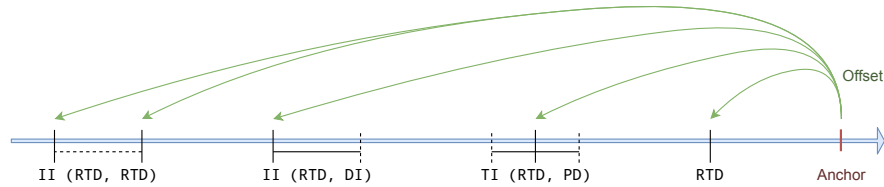


Figure 5.9: Timeline of exemplary complex temporal expressions expressible using both the Time and Relational Time languages. The indicated involved concepts are named after the initials of concepts shown in Figure 5.6 on page 92 and Figure 5.7 on the previous page. Normal lines indicate properties defined by the user, dashed lines are the resulting temporal boundaries.

on start Instant and Duration (TI). Combination of the Time and Relative Time language allows to define temporal expansions which are anchored to a set point in time. Additionally, the user provides an Offset which is added relative to the Anchor as show in Equation (5.20) on the preceding page. For the *EISE* domain, I chose a SymbolicAnchor SA representation, which expresses the query execution time. Figure 5.9 shows four examples of possible relative temporal expressions. The most left example is produced by creating an InstantInterval which holds two RelativeTimeDescription. These relative concepts act in this constellation as Instants and thus represent the start and end of the Interval. Many more combinations are possible using the Relative Time language by combining them with the concepts of the Time language. While the languages conceptually allow the representation of all required temporal expansions, it is important to note that the *concrete syntax* and language *pragmatics* need to ensure that the interface for the users is simple and hides unnecessary complexity of time expansion composition.

The adapter Temporal Graph Query language functions as the combination language of temporal features provided by the Time language, the Relative Time language and query capabilities provided by the Domain Graph Query language (FR5). The result is the additional feature to temporally constrain entire queries or parts of them. Figure 5.10 on the facing page shows the *meta-model* of this language which adds the two central capabilities of

- a) Annotating DomainElements as timed elements thus declaring the time annotation type (left), and

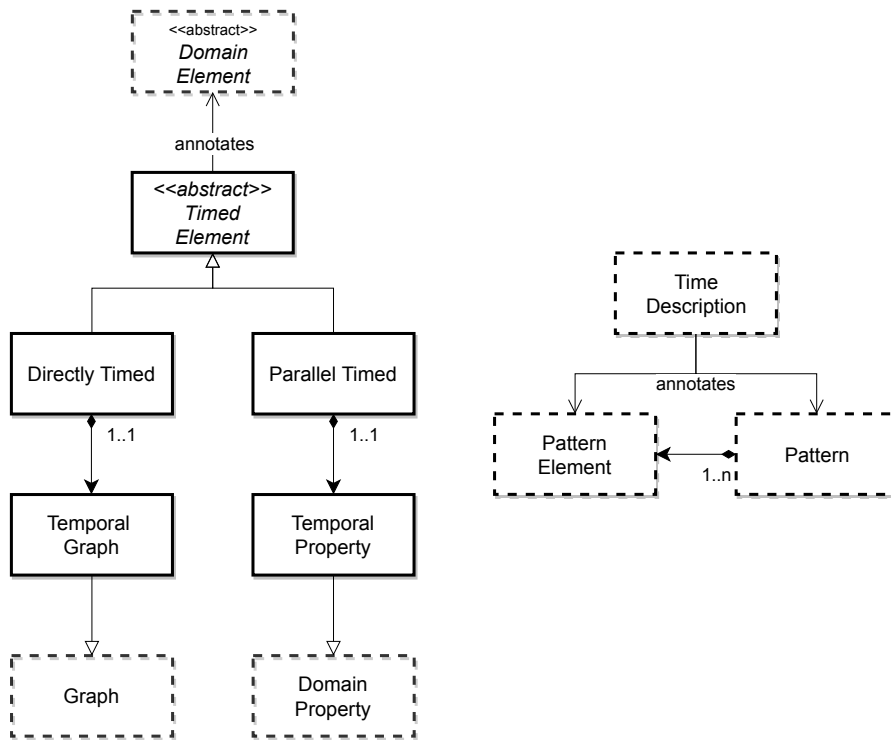


Figure 5.10: The Temporal Graph Query language *meta-model* which primarily acts as an adapter language to join the otherwise independent source languages. Dashed concepts are part of (transitive) dependent languages.

- b) Annotating Patterns or PatternElements with a TimeDescription (right).

The Temporal Graph Query language makes great use of orthogonal language composition (**NFR₃**), also referred to as language *annotation*. This technique allows to annotate any existing node of a given *abstract syntax tree (AST)* with concepts from other languages without the original *AST* requiring any knowledge about this addition. I chose this composition approach as time and temporal constraints are actually orthogonal to the queries themselves. Time is a structured and stable domain which has been analyzed extensively and as a result these concepts can reside in corresponding *meta-model(s)* within the *M₂* layer. This contrasts to the dynamic and thus unstable domain description where I chose a more dynamic approach moving domain descriptions to user *models* in the modeling layer *M₁*. Moreover, to be able to express temporal constraints is a crosscutting concern which has applicability at different levels and in a wide range of languages, hence further supporting an orthogonal language composition approach. For the application in queries, this composition type allows a seamless annotation of queries – or parts of it – with temporally constraining information, without changing the query itself. While the considerations shown here are in combination with the Domain Graph

Query language, a similar annotation on the generic Graph Query language is theoretically possible. With no domain-specific knowledge available, however, other mechanisms would be required in this case to ground temporal constraints.

The semantics of the Time language consist of two distinct parts: a) semantics of time representation in a given domain description, and b) semantics of *GQL* annotations for temporal queries. To explain the semantics of the Time language within a *GQL*, I base on the semantics for a single pattern-based query *Q* as shown in Equations (5.1) to (5.3) on page 85 and the semantics of the Domain Description language as shown in Equations (5.12) to (5.18) on pages 89–90.

TIME REPRESENTATION AND MODELING First, the definition of a time abstraction (or temporal domain description) needs to be created before the behavior of temporal query annotations can be defined. Different approaches are possible to express temporal properties within a graph-based structure; Figure 5.11 on the facing page shows two alternative variants of time representations on a graph. The first depicted approach (Figure 5.11 on the next page, left) uses the graph structure to represent a timestamp. This approach practically implements a temporal index which is not uncommon in *GDB* applications [TB09; Sfa+13; SP16]. This feature is usually not implemented within the database itself as proposed here, but rather provided by a dedicated index feature. This additional parallel graph structure allows to inquire time aspects starting directly from time graph elements. Queries targeting nodes related to the individual time constructs can easily be expressed. However, this option allows to only relate nodes to the individual time elements (seconds, minutes, etc.) but does not support to express temporal relationships. The alternative approach (Figure 5.11 on the facing page, right) does not include a separated graph. This representation embeds temporal information into the graph elements via distinct time related properties (i.e. timestamps). In this case nodes and relationships can be annotated with temporal information, allowing for a more flexible domain description. Queries on this abstraction result in a more complex query design: Queries need to be constructed such that they match the desired sub-graph and subsequently the results need to be filtered based on the time constraints. Both approaches allow to represent durations by attaching two temporal elements (e.g. a start and end time or start time and duration) which is helpful for example for data retention strategies.

During the iterative development process I chose the second approach (Figure 5.11 right) for the abstraction of the temporal representation. I use timestamps as distinct properties on nodes and relationships to represent temporal information. Though this approach provides a higher expressiveness for temporal concepts, the challenges

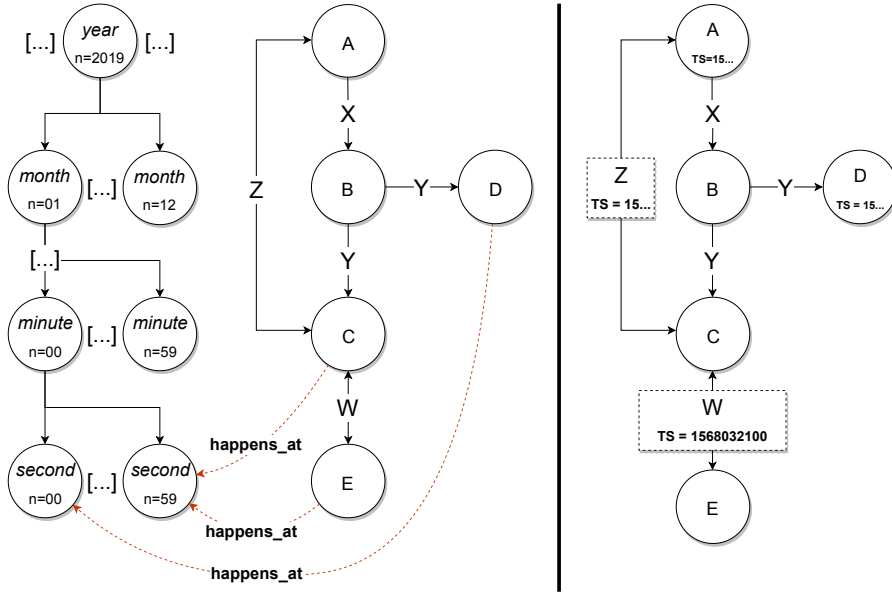


Figure 5.11: Two different approaches to represent time within a graph. First creating a parallel sub-graph expressing time and use distinct relationships to reference the temporal information (left); Second directly using the features of the *labeled property multidigraph* via node and relationship properties to annotate time information using a timestamp (right).

are potentially increased costs in terms of queries complexity within filtering clauses. However, the application of the *MDSE* approach mitigates this central issue: With the *model* knowledge available, additional complex statements or query elements can be generated. This reduces the cognitive task for the developer while being able to use a representation of higher expressiveness.

I chose to use the previously introduced *DomainDescriptionGraph* (Section 5.4.3 on page 88) to indicate the type of abstraction of temporal information. Domain description graph elements (*DomainNode* and *DomainRelationship*) can be marked via orthogonal language composition (i.e. concept annotations) as graph components containing temporal information (cf. Figure 5.10 on page 95, right). The resulting Time Graph Query language semantics of a domain description graph, which is annotated with a time description $\llbracket \text{DDG}^{\text{TD}} \rrbracket_{\text{TGQ}}$, are expressed by Equations (5.21) to (5.22).

$$\llbracket \text{DDG}^{\text{TD}} \rrbracket_{\text{TGQ}} = (V, E, \rho, \lambda, \llbracket \sigma \rrbracket_{\text{TGQ}}) \quad | \quad (V, E, \rho, \lambda, \sigma) = \llbracket \text{DDG} \rrbracket_{\text{DD}} \quad (5.21)$$

$$\llbracket \sigma \rrbracket_{\text{TGQ}} = (V \cup E) \times (\text{Prop} \cup \{\llbracket \text{TP} \rrbracket_{\text{C}}\}) \quad | \quad \text{Const} = \text{Types} \quad (5.22)$$

$$\rightarrow \text{Const}$$

Such a domain description graph is represented by its usual graph elements. The semantics for the *ParallelTimed* temporal annotation simply adds a dedicated *TemporalProperty* TP containing a user de-

defined property to the σ of the existing graph. This property expresses a temporal abstraction and expects individual timestamps onto `DomainElements` as depicted in [Figure 5.11](#) (right). Depending on the user data and domain properties the temporal annotation strategy and appropriate semantics can be chosen. The semantics of alternative approaches such as the `DirectlyTimed` are omitted in this thesis; the alternative approach is provided to highlight the customization aspects of the language design.

TIME ANNOTATION IN GRAPH QUERIES The second part of the Time Graph Query language describes the features and the behavior when combined with a GQL ([Figure 5.10](#) on page 95, right). Though these semantics must conform to the time abstraction semantics presented in the previous paragraph, their usage and resulting behavior is transparent to the query designers: Within my proposed languages a pattern-matching query with temporal constraints is expressed by a simple annotation of a `TimeDescription` on a `Pattern` or `PatternElement` (i.e. a matching sub-graph) using a `(Relative)TimeDescription` from the `(Relative) Time` language. This annotation holds a time description (i.e. a point in time or a range of time) which describes the temporal constraint the annotated element needs to satisfy. As a result, users of the query language do not necessarily need to be familiar with the underlying temporal abstraction as they simply mark the `Patterns`, `Nodes`, or `Relationships` with an intended temporal constraint. The `Relative Time` language additionally provides the means to formulate queries relative (i.e. with an `Offset`) to a set temporal `Anchor`. In the applied Time Graph Query language for the *EISE* domain I chose a `SymbolicAnchor` which references the time of query execution:

$$(A, OFF) = (SA, OFF) \quad (5.23)$$

$$= \llbracket SA \rrbracket_{TGQ} + \llbracket OFF \rrbracket_{TGQ} \quad (5.24)$$

This `Anchor` allows the users to always formulate their temporal constraints on a query with relation to the time of its future execution hence reducing the query composition complexity further. For the Cypher based pattern matching query semantics, the execution time `SymbolicAnchor` evaluates to the Cypher internal timestamp function.

$$\llbracket SA \rrbracket_{TGQ} = \text{(timestamp()/1000.0)} \quad (5.25)$$

[Equation \(5.20\)](#) on page 93 thus transforms to a representation including the execution time.

$$\begin{aligned} \llbracket RTD \rrbracket_{RT} &= \llbracket SA \rrbracket_{RT} + \llbracket OFF \rrbracket_T \\ &= \text{(timestamp()/1000.0)} + \llbracket OFF \rrbracket_T \end{aligned} \quad (5.26)$$

The *denotational semantics* for queries $\llbracket Q^{\text{DD},\text{TD}} \rrbracket_{\text{TGQ}}$ grounded into a DomainDescription and temporally constraint by TimeDescription annotations are formulated in Equations (5.27) to (5.36). Initially, the individual query clauses M' , C' , and R' are obtained following the evaluation of the query $\llbracket Q' \rrbracket_{\text{DGQ}}$ using Cypher semantics. Query Q' represents an identical query to $Q^{\text{DD},\text{TD}}$ which is stripped from all time annotations (Equation (5.27)). While the obtained clauses M' and R' are reused directly for the evaluation of $Q^{\text{DD},\text{TD}}$ in the Time Graph Query language, additional conditions are concatenated to the filter clause C' to satisfy time annotation constraints (Equation (5.28)). Time annotations are either defined globally as TD^g on a complete matching Pattern P^{DD} (Equation (5.30)) or locally as TD^l on each individual PatternElement PE^{DD} (Equation (5.31)). For each locally annotated PatternElement $\text{PE}^{\text{DD},\text{TD}^l}$ (i.e. either a node or a relationship) an additional condition is appended to the filtering clause as listed in Equations (5.32) to (5.36). These conditions ensure that the annotated element satisfies the provided time description in compliance to the chosen time model as described in Section 5.4.4 on page 96. Similarly, a global TimeDescription annotation TD_j^g on a Pattern P^{DD} propagates its annotation downward to each PatternElement PE_k^{DD} which holds a TimedElement annotation in the domain description (Equation (5.29)).

$$\llbracket Q' \rrbracket_C = (M', C', R') \quad (5.27)$$

$$\begin{aligned} \llbracket Q^{\text{DD},\text{TD}} \rrbracket_{\text{TGQ}} &= \mathbf{MATCH} \ M' \\ &\quad \mathbf{WHERE} \ (C' \oplus_{\text{AND}} (\llbracket M^{\text{DD},\text{TD}} \rrbracket_{\text{TGQ}})) \\ &\quad \mathbf{RETURN} \ R' \end{aligned} \quad (5.28)$$

$$\begin{aligned} \llbracket M^{\text{DD},\text{TD}} \rrbracket_{\text{TGQ}} &= \llbracket (P_1^{\text{DD},\text{TD}^l}, \text{TD}_1^g) \rrbracket_{\text{TGQ}} \oplus_{\text{AND}} \dots \\ &\quad \oplus_{\text{AND}} \llbracket (P_i^{\text{DD},\text{TD}^l}, \text{TD}_i^g) \rrbracket_{\text{TGQ}} \end{aligned} \quad (5.29)$$

$$\begin{aligned} \llbracket (P_j^{\text{DD},\text{TD}^l}, \text{TD}_j^g) \rrbracket_{\text{TGQ}} &= \llbracket (\text{PE}_1^{\text{DD},\text{TD}^l}, \text{TD}_1^g) \rrbracket_{\text{TGQ}} \oplus_{\text{AND}} \dots \quad | \ \forall P_j \in M \\ &\quad \oplus_{\text{AND}} \llbracket (\text{PE}_n^{\text{DD},\text{TD}^l}, \text{TD}_n^g) \rrbracket_{\text{TGQ}} \end{aligned} \quad (5.30)$$

$$\begin{aligned} \llbracket (\text{PE}_k^{\text{DD},\text{TD}^l}, \text{TD}_k^g) \rrbracket_{\text{TGQ}} &= \llbracket (\text{PE}_k^{\text{DD}}, \text{TD}_k^l) \rrbracket_{\text{TGQ}} \oplus_{\text{AND}} \\ &\quad \llbracket (\text{PE}_k^{\text{DD}}, \text{TD}_k^g) \rrbracket_{\text{TGQ}} \quad | \ \forall \text{PE}_k \in P_j \end{aligned} \quad (5.31)$$

Any PatternElement PE_k is either a node or a relationship and hence represented by the tuple of its children $(y, s, _)$, where y represents the local element identifier (i.e. a local variable) of graph element s with its properties $_$ ⁸. The individual conditions which are

⁸ The properties are irrelevant for the semantics on how temporal constraints are constructed and are hence omitted here.

added to the filtering clause follow the denotations in [Equations \(5.32\)](#) to [\(5.36\)](#).

$$\llbracket ((y, s, _), I) \rrbracket_{\text{TGQ}} = y \cdot \llbracket \text{TP} \rrbracket_{\text{C}} = (\llbracket I \rrbracket_{\text{RT}}) \quad (5.32)$$

$$\llbracket ((y, s, _), (I, D)) \rrbracket_{\text{TGQ}} = y \cdot \llbracket \text{TP} \rrbracket_{\text{C}} \geq (\llbracket I \rrbracket_{\text{RT}} - \llbracket D \rrbracket_{\text{RT}}) \text{ AND } y \cdot \llbracket \text{TP} \rrbracket_{\text{C}} \leq (\llbracket I \rrbracket_{\text{RT}} + \llbracket D \rrbracket_{\text{RT}}) \quad (5.33)$$

$$\llbracket ((y, s, _), (I, D)) \rrbracket_{\text{TGQ}} = y \cdot \llbracket \text{TP} \rrbracket_{\text{C}} \geq (\llbracket I \rrbracket_{\text{RT}}) \text{ AND } y \cdot \llbracket \text{TP} \rrbracket_{\text{C}} \leq (\llbracket I \rrbracket_{\text{RT}} + \llbracket D \rrbracket_{\text{RT}}) \quad (5.34)$$

$$\llbracket ((y, s, _), (I_1, I_2)) \rrbracket_{\text{TGQ}} = y \cdot \llbracket \text{TP} \rrbracket_{\text{C}} \geq (\llbracket I_1 \rrbracket_{\text{RT}}) \text{ AND } y \cdot \llbracket \text{TP} \rrbracket_{\text{C}} \leq (\llbracket I_2 \rrbracket_{\text{RT}}) \quad (5.35)$$

$$\llbracket ((y, s, _), \epsilon) \rrbracket_{\text{TGQ}} = \epsilon \quad (5.36)$$

Their implications are covered by the five cases where the user provided one of the following concepts⁹:

1. Instant I ([Equation \(5.32\)](#)),
2. ToleranceInstant (I, D) ([Equation \(5.33\)](#)),
3. DurationInterval (I, D) ([Equation \(5.34\)](#)),
4. InstantInterval (I, I) ([Equation \(5.35\)](#)), or
5. None ϵ ([Equation \(5.36\)](#)).

The semantics for these conditions implement Allen's logic and ensure the TemporalProperty TP of each PatternElement lies within the user provided TimeDescription constraint. The source tuples in [Equations \(5.32\)](#) to [\(5.35\)](#) describe nodes and relationships alike, such that $s \in V \cup E$.

5.4.5 Plug-ins and implementation modules

The three remaining *modules* shown in [Figure 5.2](#) on page 82 (dashed boxes) Query Execution, Query Analysis, and Query Visualization differ from the previously explained languages in their realization as they are implemented using language *pragmatics*¹⁰ [Rod15a]. These plug-ins and extensions are implementation-specific components and the proposed language architecture enables them to be unobtrusive and not centralized dependent *modules*. Unlike the language conceptualizations, the implementation of practical features is heavily influenced by the used *language workbench*, *GDB*, and *GQL*.

The Query Execution *module* provides a bridge between the user query and the storage back-end to allow query execution and presentation of query results. Similarly, the Query Analysis also uses

⁹ These cases cover both the absolute TimeDescriptions as well as the Relative-TimeDescriptions as the latter are a specialization of the abstract Instant (cf. [Figure 5.7](#) on page 93)

¹⁰ Language *pragmatics* are practical features implemented to enable a certain functionality. Unlike languages, *pragmatics* are not modeled explicitly as their pragmatic element is closely tied to implementation specific factors.

external tools, which are used to analyze the current query. The obtained results then also need to be evaluated and presented to the users. Lastly, the Query Visualization *module* provides a different *concrete syntax* of a user query.

The type of implementation (i.e. how it is integrated) depends strongly on the used *language workbench*. For example, when using *JetBrains Meta Programming System (MPS)* as the target platform, the visualization needs to be integrated using a *MPS Java plug-in module* and can make use of the projectional editing feature to provide an individual projection of a query that shows the matching sub-graph pattern(s).

5.5 TECHNOLOGY MAPPING

The language descriptions are implementation-independent and formulated from a conceptual perspective. The decisions in the technology mapping consequently summarize the conceptual decisions leading up to this point. These conceptualizations are linked by the mapping to the underlying implementation (RQ4). Figure 5.12 on the following page presents this mapping and covers the language layer, conceptual layer and implementation layer. The mapping starts at the tool level (top) and increases with respect to specificity via each of the three layers a) language layer, b) conceptual layer, and c) implementation layer.

The language layer picks up on the language composition in Section 5.4 and proposes my decision to use *MPS* as the *language workbench* for *DSL* implementation. This choice allows to realize individual requirements via provided features of the *language workbench*. Factors regarding language composition (NFR3) and system integration (FR7, NFR4, NFR5) are implemented using *MPS*'s extensive language composition and *IDE* generation features. Further, orthogonal features such as lifted feedback and visual representations are well supported via the projectional editing features in this particular *language workbench* (FR5, FR6 and FR8). Lastly, the provided *BaseLanguage* provides a *DSL* and generators for the *GPL* Java and thus supports the decision considering developer bias and implementation system architecture (FR6, NFR1, NFR5).

The implementation layer presents the specific software, libraries, and technologies chose for the implementation of the *vertical prototype*. Middleware and domain types use the *Robotics Service Bus (RSB)* ecosystem (rsb-java, rsb-proto, and rsb-proto-csra) which is also used in the *CSRA* project. This fosters the system integration and allows to use domain-specific *Robotics Systems Types (RST)* types for grounding in the existing environment (FR7, NFR1, NFR5). I use the combination of Neo4j and Cypher as the technological basis for *GDB*, *GQL*, and *GQL* engine. Their combination presents an optimal choice

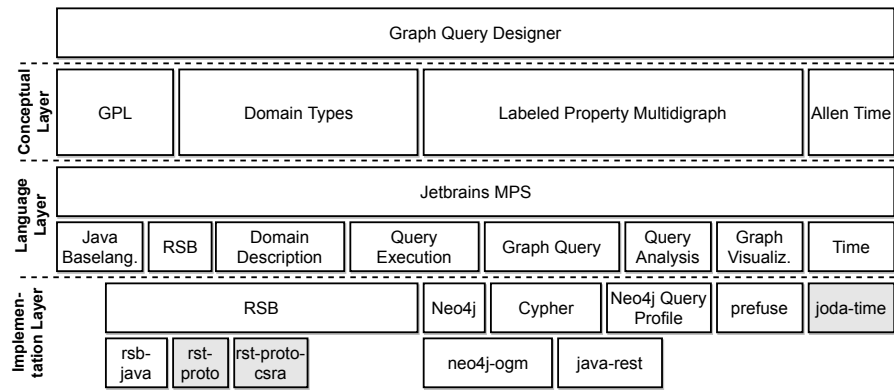


Figure 5.12: The technology mapping binding the system architecture shown in Figure 5.1 on page 80 and composing languages in Figure 5.2 on page 82 to specific conceptual abstractions and specific implementation technologies applicable in the *EISE* domain. The specialization starts at the top from the intended tooling and increases in specificity with each layer to the bottom. Grayed elements are not used/implemented in the *vertical prototype* of this work.

(cf. Section 2.2 on page 14) for the application in the *EISE* domain. The usage of Cypher fulfills **NFR₁** with its closeness to known well known query languages such as *SQL*. Access further is simplified using the `java-cypher-ogm` wrapper allowing to execute queries towards Neo4j while keeping node and relationship type safety according to the domain description. Other access to the database is realized via the *REST API*, e.g. for the extraction of query profiling from the query engine. Lastly, graph visualizations within the created tooling makes use of the `perfuse` graph visualization library. The creation of relatively timed queries makes use of the `joda-time` library to easily compute time element relations.

5.6 SUMMARY

This chapter presents an implementation-independent conceptualization of the application of a *MDSE* approach for the *EISE* domain. As the basis for the abstractions proposed in this chapter, the objectives, eight functional, and five non-functional requirements are extracted. These requirements are derived from the domain analysis and objectives of the approach previously presented in this thesis. The following implementation-independent system architecture specifies the concepts involved in the targeted system and relates the individual structural elements of the target *IDE* to the *OMG* layers. A further contribution of this chapter is the detailed language composition and modularization description. The presented languages and their relations are extensively described and discussed individually. For each language intersection *denotational semantics* are presented that show

the intentions and usage of resulting languages and concepts. This involves languages representing of a *labeled property multidigraph*, domain descriptions, pattern based graph queries, and (relative) time constrained queries. Lastly, this chapter presents a mapping of the extracted theoretical ideas and conceptualizations into the technological implementation real-world. This chapter is the theoretical foundation for the next chapter in which the applied *MDSE* is presented which yields a *vertical prototype* of the core features. The created prototype serves as a proof for the applicability of these theoretical abstractions and simultaneously serves as the tool used for the evaluation of the approach in the subsequent chapter.

Part IV

MODEL-BASED SUPPORT FOR BEHAVIOR DEVELOPERS

The fourth part presents a *vertical prototype* which implements the requirements and provides support for application developers.

IMPLEMENTATION AND LANGUAGE PRAGMATICS FOR THE EISE DOMAIN

“Above all, don’t fear difficult moments. The best comes from them.”

—Rita Levi-Montalcini
Nobel Prize-winning neurobiologist
who co-discovered nerve growth factor

This chapter presents details of the implementation (**RQ4**) of the individual *domain-specific languages (DSLs)*, the language *pragmatics*, automation aspects of the MDSE application in a research setting, and finally the combined *integrated development environment (IDE)*. The contributions are based on the previously defined a) system architecture, b) implementation-independent language *meta-models*, c) *denotational semantics* for composed languages, and d) the technology mapping. The content of this chapter is thus anchored in the phases Phase **P3. Language Implementation** and Phase **P4. Automation** of the development cycle proposed in [Section 3.3](#) on page 44. Similarly to previous chapters, the presented implementation results from multiple development iterations within the *Cognitive Service Robotics Apartment as Ambient Host (CSRA)* project. However, in contrast to the theoretical considerations in the previous chapter [Chapter 5](#), I present the implemented languages and highlight implementation specific changes if applicable. Changes to the *meta-models* are required if for instance a certain feature is not supported by the used *language workbench*. One example for this is single cardinality only references in MPS. To implement references with multiple cardinality a helper concept needs to be introduced which holds a list of the references. As a final contribution of this chapter, I present the *EISE Query Designer (EISEQD)*, the current version of the *vertical prototype* of the query *IDE*. I provide a view from the user perspective on how to create a domain description, declare temporal entities, and combine these features in the query design process. A version of this *IDE* was used for the user study and evaluation in the following [Chapter 7](#).

6.1 LANGUAGE IMPLEMENTATION

The implementation of the presented languages is carried out using the *language workbench JetBrains Meta Programming System (MPS)*. The clear separation of a language into individual language *aspects* in *MPS* provides the development process with dedicated support for

language development. As such, the language creation is clearly separated into:

- *Abstract syntax* of a language is defined via the *structure aspect*,
- *Concrete syntax* is implemented using the *editor aspects*,
- To restrain the *model* and to conform to the desired language semantics one can use the *constraints aspects*, *typesystem aspects* or *dataflow aspects*,
- *Model-to-model (M2M)* and *model-to-text (M2T)* definitions are done in the *generator aspects* and *textgen aspect*,
- All other elements, such as language *pragmatics*, can be defined in *actions aspects* or other *user created aspects*.

Further, using a projectional editor such as *MPS* allows to lift certain language *pragmatics* from the language layer into separated *modules*. For example, visualizations are not languages themselves but rather

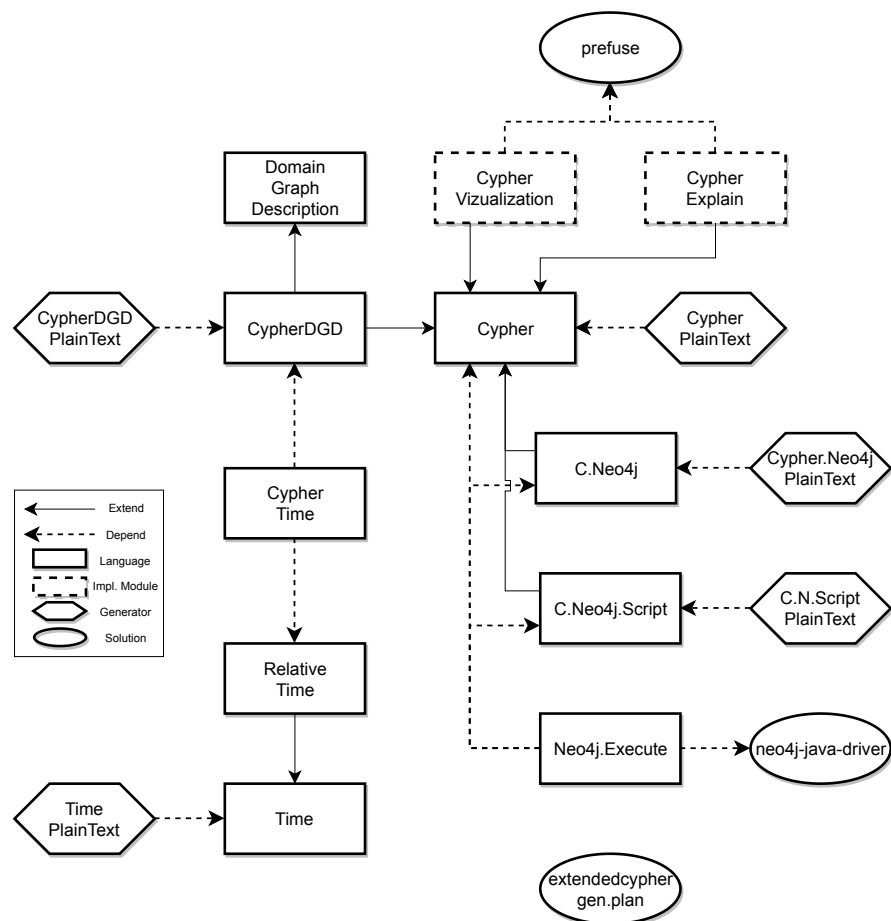


Figure 6.1: Language composition implemented. Dependencies into JetBrains *MPS* internal *modules* are omitted for the sake of readability.

provide an additional *concrete syntax* to an existing language via an implementation *module* containing no new abstract concepts. This allows to extend languages easily and transparently to the users (e.g. by providing dedicated visual projections of the *model*).

6.1.1 Language composition

I compose the set of languages using multiple *MPS modules*, which provide abstractions for individual sub-domains and/or solve other technological hurdles. The implementation was created based on the language independent considerations in the previous [Chapter 5](#). The current iteration of the language dependency graph is shown in [Figure 6.1](#). The composition depicts the languages, *solutions*, and implementation *modules* as well as the corresponding interconnections using MPS notations of *module extension*, *dependence*, and *use*.

The use of these is supported via *Devkits* which bundle up *MPS modules* and can be imported to jointly provide all dependencies for a certain functionality. [Figure 6.2](#) additionally shows the provided *Devkit* structure. Each *Devkit* abstracts a sub-domain or larger parts of each, for example the Time *Devkit* provides the dependencies to

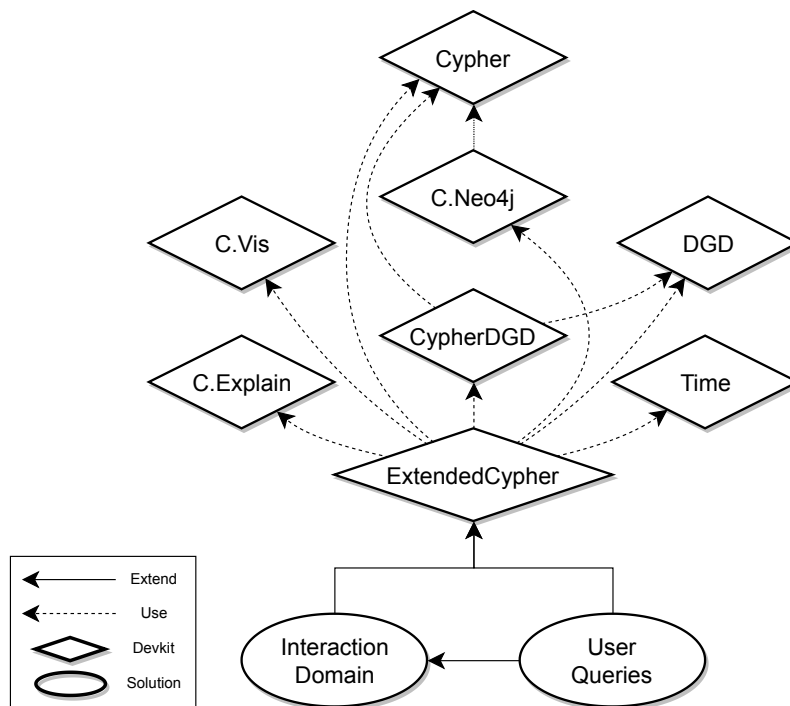


Figure 6.2: *Devkit module* composition which provides the final abstraction layer for users. Each *Devkit* aggregates multiple *modules* from the implemented languages shown in [Figure 6.1](#) and from *MPS internal modules*. The aggregated *modules* are omitted for readability. Only the final *ExtendedCypher Devkit* is required for a user *solution* which provides all *EISEQD* capabilities.

MODULE	STRUCTURE								ASPECTS					
	CONCEPTS	INTERFACES	ENUMS	PROPERTIES	REFERENCES	AGGREGATES	IMPLEMENTS	EXTENDS	EDITOR	CONSTRAINTS	BEHAVIOR	TYPESYSTEM	INTENTIONS	ACTIONS
Cypher	110	35	3	7	4	79	79	116	70	11	7	41	16	3
C.Neo4j	1	0	0	0	0	2	1	1	1	0	0	0	0	0
C.Neo4j.Script	15	1	0	0	3	6	15	14	14	4	1	7	0	1
C.Neo4j.Exec	1	0	0	0	0	1	1	1	0	0	0	0	0	0
C.Explain	0	0	0	0	0	0	0	0	2	0	8	0	0	0
C.Vis	0	0	0	0	0	0	0	0	3	0	8	0	0	0
DGD	25	4	0	0	10	22	22	28	44	9	3	19	0	0
CypherDGD	4	0	0	0	0	0	11	4	2	1	2	1	0	0
Time	23	2	15	26	0	26	11	24	18	1	14	1	3	0
RelativeTime	6	1	2	3	0	8	4	6	4	1	1	0	0	0
CypherTime	4	1	0	2	0	1	1	4	3	0	0	1	4	0
Dot	25	4	3	7	4	19	11	22	22	1	3	0	0	0
DepDiagram	3	1	0	13	0	3	2	3	2	0	2	0	0	1
Total	217	49	23	58	21	167	158	223	185	28	49	70	23	5

Table 6.1: Statistics on the implemented languages covering details for all *MPS* aspects. Language coupling is expressed by showing references, aggregations, implementations, and extensions used.

the Time and RelativeTime languages as well as the TimePlainText generator. The ExtendedCypher *Devkit* jointly provides all required dependencies for the full capabilities of the *Embodied Interaction in Smart Environments (EISE)* Domain Query Designer.

The full *meta-models* of the implemented languages are omitted at this point as the languages contain many implementation and tool specific decisions. This is the result of the language mapping and other resulting implementation-specific constraints. Nevertheless, the *meta-models* of the shown *modules* conform to the theoretical descriptions in [Chapter 5](#).

[Table 6.1](#) provides information on the complexity of the implemented languages. The table shows statistics of all languages implemented in this work by listing the total amount of elements for each *language aspect* (structure, editor, constraints, behavior, typesystem, intentions, and actions). In addition, the structure aspect is further unraveled and separated into its composing elements and relation types: concepts, interfaces, enumerations properties, references, aggregations, implementations, and extensions. The numbers show, that central languages of the composition (compare [Figure 6.13](#)) contribute the most concepts in their implementation: The Cypher lan-

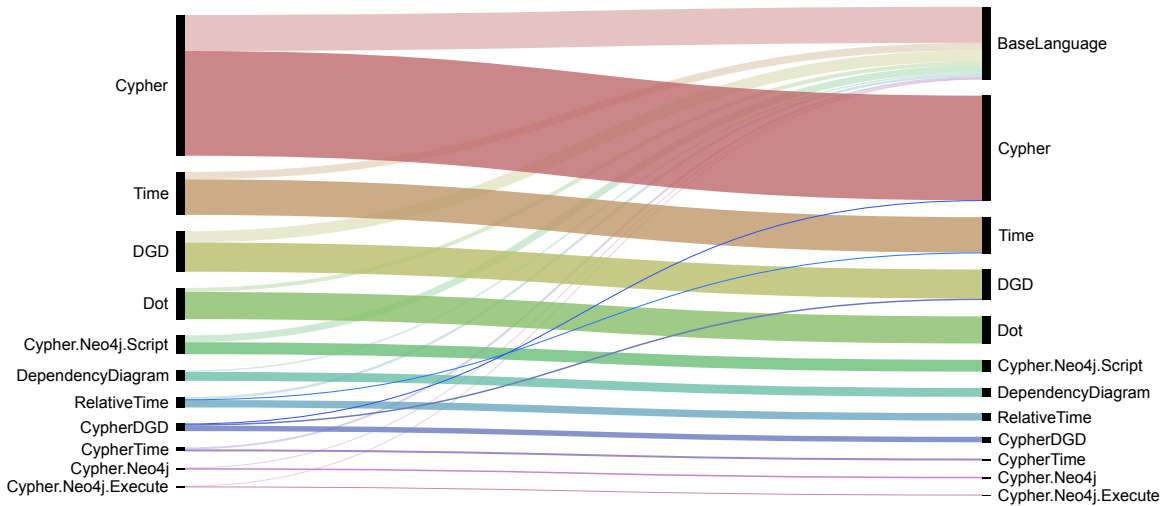


Figure 6.3: Alluvial diagram of the implemented language dependencies. Connections from concepts of languages on the left to concepts of languages on the right indicate a dependence between languages. The thickness of the connecting lines represent the amount of connections causing these dependencies (i.e. the sum of properties, references, aggregations, implementations, and extensions as shown in Table 6.1). The opacity of connections to the BaseLanguage is reduced to increase overall visibility.

language provides 110 concepts, the DomainGraphDescription (DGD) language provides 25 concepts, and lastly the Time language provides 23 concepts. At the same time, these languages also make increasing use of coupling relevant connections, such as extension or aggregation. The other languages provide comparably low numbers of new concepts but rather provide individual features beyond the extended languages on which they depend. To further analyze the language coupling, the alluvial diagram shown in Figure 6.3 gives detailed information on all existing connections between languages concepts. In this depiction, all languages on the left side have connections from their concepts (colored lines in the center) to any concept of other languages on the right side. The connections are composed of all dependency introducing structure aspects: The sum of properties, references, aggregations, implementations, and extensions as listed in Table 6.1. The height of the black bars of each language represents the sum of concepts, interfaces and enumerations, while the thickness of the connecting lines corresponds to the amount of connections. For example, concepts of the Cypher language connect to a total of 212 concepts within itself and further 73 connections exist to the BaseLanguage. This diagram shows three key properties of the implemented languages.

First, a sizable portion of the connections of most languages are to the BaseLanguage. This is to be expected as language development

using *MPS* generally results in a heavy usage of this fundamental language. If one chooses to not use the *BaseLanguage*, many low level mechanisms need to be implemented by hand. Not reusing the available and highly tailored *BaseLanguage* would result in unmanageable implementation effort.

Second, the implemented languages exhibit low coupling. The concepts of languages are mostly connected to other concepts within the same language. Examples for this type of connection are shown in the *meta-models* of the languages in [Chapter 5](#): Each arrow of a *meta-model* counts as a connection. The only languages directly involving other languages besides the *BaseLanguage* are the *RelativeTime* language (connecting to the *Time* language) and the *CypherDGD* language (connecting to the languages *Cypher* and *DGD*). This low coupling factor of the implementation results from the detailed language composition planning in [Section 5.4](#).

Third, the orthogonal language composition, which makes use of language annotations, can be identified as no dependency between the involved languages are present. As such, the *CypherTime* language is only connected to the *BaseLanguage* which provides the annotation feature as a part of the *language workbench* but no connection exists to the *Cypher* language (cf. [Section 3.1.2.4](#)). Consequently, no direct dependencies were introduced between the languages *Cypher*, *Time*, and *CypherTime*.

In the following I present each of the language *modules* in depth and explain the necessity and overall integration.

6.1.2 *Graphs and graph query languages*

The implementation of the *graph query language (GQL)* is based on the *Cypher* language. As such, the language implementation diverts from the usual approach in which the implementation closely follows the implementation-independent *meta-models*. The *Cypher* language was hence developed alongside the *Extended Backus–Naur Form (EBNF)* grammar definition of *openCypher*¹ (see [Listing 5.1](#) on page 86 for an example excerpt from this grammar). As the formalization provided by the *openCypher* initiative does not contain all features used in the existing *Cypher* implementation contained in the Neo4j *Graph Database Management System (GDB)*, I implemented additional languages covering these features. As a result, the *Cypher.neo4j* language and the *Cypher.neo4j.script* language further depend on the base *Cypher* language (cf. [Figure 6.13](#) on page 123). Following this implementation approach, no additional graph language implementation is used in the *vertical prototype*.

¹ The *EBNF* grammar version M15, as published on <http://www.opencypher.org/resources> [Neo15]

Structurally, the implemented concepts in the Cypher language rely heavily on the basic concepts provided by the *MPS* Baselanguage. Any extension provided by other languages make use of these shared common Baselanguage concepts and interfaces (e.g. BaseConcept, Expression, or INamedConcept). The resulting structures representable by the Cypher language conform to the implementation-independent descriptions in [Section 5.4.1](#) on page 84 and [Section 5.4.2](#) on page 84 and further also provide all query capabilities the Cypher query language allows (e.g. graph traversal queries and graph algorithms).

In sum, the Cypher related languages consist of 165 concepts with an expected degree of coupling ([Table 6.1](#)). The central top-level concepts (root concepts) provided by the Cypher related languages are

- Cypher Query,
- Cypher Query Collection,
- Neo4j Query Execution, and
- Neo4j Query Script.

```

RegularQuery
ReadSinglePartQuery
MATCH (flobi : Robot { string name : "flobi" }) — (aPerson : Person)
(aPerson) —[at : IS_AT]→ (kitchen : Room)
Error: type Relationship is not a subtype of int
WHERE ( at IN RANGE(1574185192 , 1574185200 , 1) ) = true AND EXISTS(aPerson)
RETURN aPerson.name AS presentPerson
Warning: This property is generic and has no grounding in the query. Typesystem checks will not check this property access.
ORDER BY ASCENDING
<no skip> aPerson ^variable (d.c.c.sandbox.generation.collectio
<no limit> at ^variable (d.c.c.sandbox.generation.collectio
flobi ^variable (d.c.c.sandbox.generation.collectio
kitchen ^variable (d.c.c.sandbox.generation.collectio
< no query union >

```

Figure 6.4: Concrete syntax example of the Cypher query language

The *concrete syntax* of the Cypher language is an unchanged implementation based on the grammar definitions. [Figure 6.4](#) shows a short *concrete syntax* example as presented by this language. Corresponding *editor aspects* are thus implemented to provide a seamless language interface.

The Cypher language already provides users with query design supporting features beyond the default state-of-the-art tools such as the Neo4j web interface. Besides syntax highlighting, completion for node and relationship concepts, local variable names, properties (via common dot notation), labels, and basic types are provided. For example, the `string` type in the first matching pattern shown in [Figure 6.4](#) is inferred from the user input in the property assignment. The type system also checks function calls and the individual clauses, such that for example the `WHERE` clause always evaluates to a boolean value. If statements are created, which have not attached type information (e.g. by accessing node properties), a warning message is displayed

informing the user that the type system cannot ensure type safety for this concept. Additionally, various intentions are present which allow to do common mundane tasks, such as switch direction of a relationship or surround statements with a function call.

The Cypher language also includes a separated *M2T* generation *module* using the existing text generator plug-in². As a result, plain text Cypher queries can be generated from the representations in *MPS*. The separated `Neo4j.Execute` language makes use of this generation target and uses the generated text *artifacts* to execute them on an existing Neo4j database. Following languages make use of *M2M* transformations to convert their language specific additions into the Cypher language *meta-model*.

6.1.3 Domain description language

In contrast to the Cypher language, the `DomainGraphDescription` (DGD) language is implemented closely to the theoretical *meta-model* shown in [Figure 5.5](#) on page 88. The language fully conforms to the theoretical considerations in [Section 5.4.3](#) on page 88. The `Node` and `Relationship` concepts are grounded in the `BaseLanguage` as `Expressions` and can thus further be embedded in the existing `Expression` language. Additionally, `Relationships` further specialize `BinaryOperations` for seamless integration as operators within `Expressions`.

The DGD language consist of a total of 29 concepts with a total of 73 relations to other concepts. Two central top-level concepts allow to describe a) `DomainDeclarations` to model concepts and relations of a domain, and b) `DomainInstantiations` to create certain instances of concepts and relations in a `DomainDeclaration`.

[Figure 6.5](#) on the facing page shows the *concrete syntax* provided to create `Domain Declarations` consisting of `Entities` and `Relationships`. Each concept is visually contained via vertical surrounding brackets. This declaration is executed in user *solutions* and allows domain experts to create their *model* of the domain instead of a *meta-model*. As shown in the depiction, suitable scoping rules in the DGD language ensure that code completion suggestions provide contextually correct concepts. The types which are used to denote properties are reused from the `BaseLanguage` and cover common basic data types. Instances of the declaration concepts defined in the domain declaration can be used to compose actual concrete instances of the domain, as shown in the *concrete syntax* example in [Figure 6.6](#) on the next page. The instances allow to define their stereotype (e.g. «Person») and subsequently provide access to the schema defined in the corresponding `Domain Description`. Again, scoping is provided, but in this case a further reduction of offered concepts is computed. For the depicted example, the only stereotype allowed for the right instance is

² <https://jetbrains.github.io/MPS-extensions/extensions/plaintext-gen/>

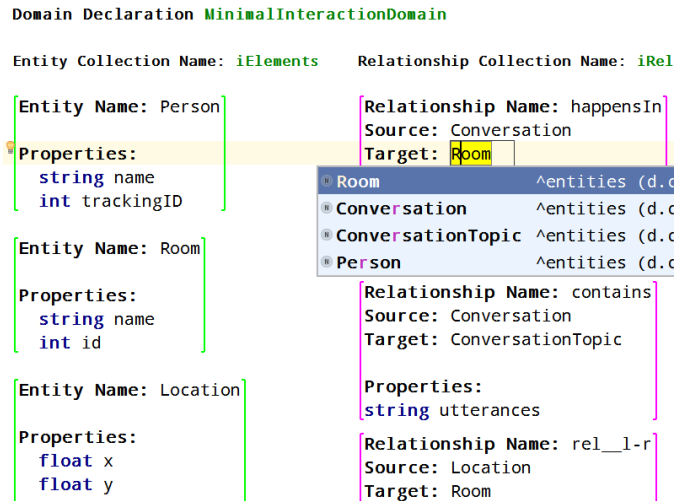


Figure 6.5: The *concrete syntax* provided by the domain description language allows to describe Entities (surrounded by green brackets) and Relationships (surrounded by magenta brackets).

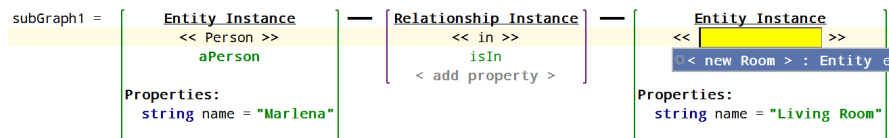


Figure 6.6: Additional *concrete syntax* used by the Domain Description language to express instances of the Entities and Relationships shown in Figure 6.5.

a Room as the description only defines this Relationship. Similar completions are provided when accessing Properties, however Generic Properties not defined in the description are also allowed (resulting in a warning due to missing type support).

6.1.4 Time languages

The Time language implementation is inspired by the *Time Ontology in OWL* as presented by the *World Wide Web Consortium (W3C)*. This ontology gathers all central temporal classes, their topology, and principles [W3C17; HP04]. The final implementation is comprised of 40 concepts in total of which a 15 concepts are specialized enumerations. The time domain has multiple predefined ranges of constants (e.g. timezones, weekdays, or number of seconds with a minute) and thus this language requires this increased amount of enumeration concepts. The implementation conforms to the *meta-model* as presented in Figure 5.6 on page 92 and is fully independent of all other languages presented in this thesis. However, the Time language also depends upon the internal Baseline language for eased language composition. As a general language to express time related elements, this lan-

language provides top-level concepts for Instants, Intervals and Durations. Figure 6.7a depicts examples for the *concrete syntax* included in the language. Additionally, included language *pragmatics* are shown, such as intentions for time selection or time zone completions based on the time zone enumerations.

The expression of relative time concepts is provided by the Relative Time language. In contrast to the Time language, the Relative Time language contains only eight concepts in total as it mainly relies on the Time language and extends it with feature to represent relational temporal expansions. The implementation of this language is also conforming to the theoretical considerations in Figure 5.7 on page 93. Though the Relative Time language is an intermediate language, it also provides top-level concepts and suitable language *pragmatics* to describe relational time descriptions as shown in Figure 6.7b. Figure 6.7 shows multiple examples of the editor representations of time related concepts.

6.1.5 Model transformations and generation of queries

The generation of *artifacts* of the different *models* and representations fall back onto the underlying GQL which is Cypher. The implemented generation plan thus initially transforms all concepts expressed in higher order languages (e.g. domain description instances or time constraints) to the GQL representation via the provided M2M transformations. Therefore, the *denotational semantics* as explained in Section 5.4 on page 81 are implemented by the M2M transformation generators. A dedicated GQL M2T generator then transforms this common *model* to the textual representation pendant using the corresponding reduction rules for each concept.

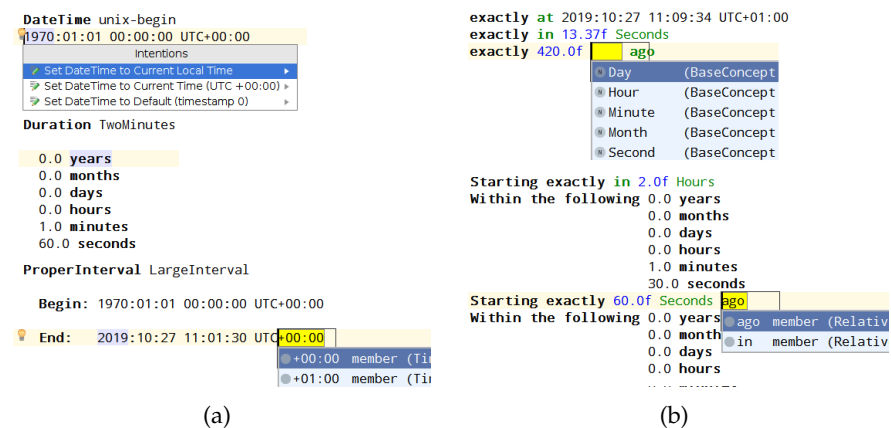


Figure 6.7: *Concrete syntax* examples provided by the Time (left) and Relative Time language (right) languages used to express time related concepts.

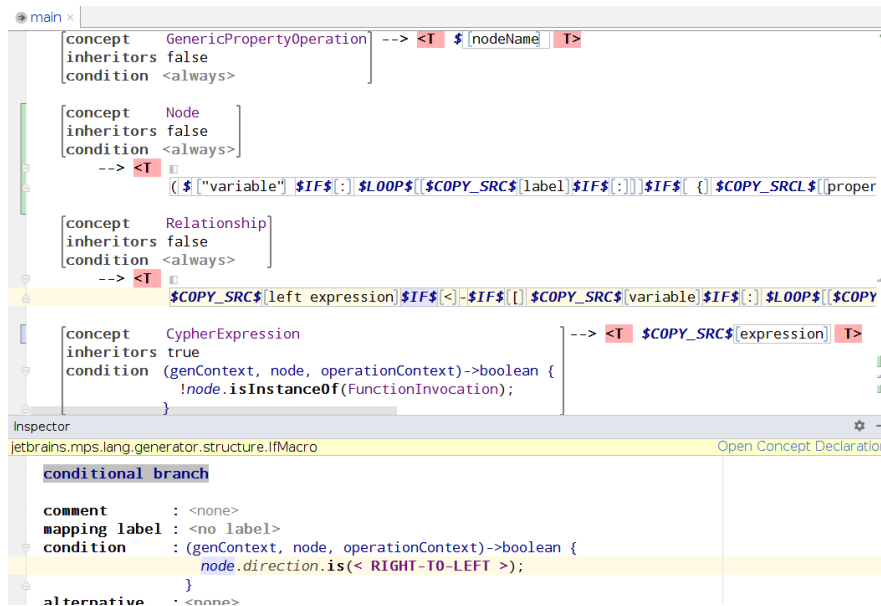


Figure 6.8: Excerpt of the Cypher generator used to generate plain text Cypher queries. The complete generator is separated into its own *module* and the shown part contains the reduction rules for Node and Relationship concepts.

Technologically, the generation implemented is designed for extensive language composition and diverse options for (future) integration, by practicing *separation of concerns*. As such, each language, generator, implementation *module*, and build *module* is implemented in a dedicated project. The generation pipeline reuses the *Plaintex* plug-in³. This allows to create dedicated *M2T* generators using the full MPS generator feature set, instead of the integrated easy to use but feature wise simplified internal *textgen* language aspect. With the generators extracted into their own languages, the *DSLs* are independent from the generation step. Multiple generators for different targets can be defined and co-exist. The users can chose the generators (or generator plans) involved in the generation step for their application. One example for this feature is the Cypher to text generator which creates plain text from the query, while another generator can generate source code for the Neo4j *Object Graph Mapping* library.

Figure 6.8 shows an example screenshot of the Cypher text generator implemented in the *vertical prototype*. The generator resides within its own language and solely provides a transformations for the parent Cypher language: Language concepts are generated into their corresponding textual representations. The example depicts four reduction rules for the concepts *GenericPropertyOperation*, *Node*, *Relationship*, and *CypherExpression*. With the help of *MPS's* template language, each concept is reduced to its *concrete syntax* elements,

³ This plug-in is part of the officially by JetBrains supported community MPS extensions at <https://github.com/JetBrains/MPS-extensions>

```

Inspector
jetbrains.mps.lang.generator.structure.InsertMacro

insert a node

comment      : <none>
mapping label : <no label>
output node  : (genContext, node)->node<> {

    foreach match in node.descendants<concept = Match> {

        if (match.@TimeAnnotation.isNotNull) {
            // skip
        } else {
            foreach dgdRel in match.descendants<concept = DGDRelationship> {
                if (dgdRel.@TimeAnnotation.isNotNull) {

                    if (match.where.isNotNull) {
                        match.where.set new(<default>);
                        match.where.expression.set new(<default>);
                        match.where.expression.expression.set new(<default>);
                        match.where.expression.expression.set($BooleanConstant(value: true));
                    }

                    node<CypherAndExpression> andExp = new node<CypherAndExpression>();
                    andExp.rightExpression = match.where.expression.expression;

                    andExp.leftExpression = $ParenthesizedExpression(expression:
                        $GreaterThanEqualsExpression(
                            leftExpression: DotExpression(
                                operand: DGDRelationshipReference(binaryRelationshipInstance: # dgdRel),
                                operation: GenericPropertyOperation(name: "when"),
                            ),
                            rightExpression: MinusExpression(
                                leftExpression: TimestampFunction(expression: EmptyExpression()),
                                rightExpression: FloatingPointConstant(
                                    value: "" + dgdRel.@TimeAnnotation.when as
                                    SimplePreviousRelativeTimeInterval.duration.
                                    getTotalSeconds())));
                }
            }
        }
    }
}

```

Figure 6.9: Temporal query constraint generator example showing an excerpt of the \$INSERT\$ macro generating temporal constraints.

for example by looping over child concepts (\$LOOP\$), calling subsequent reduction rules (\$COPY_SRC\$), or by condition reduction (\$IF\$). The latter example is selected and in the inspector at the bottom of the screen the condition is resolved via the actual node direction defined in the user *model*.

Figure 6.9 shows another generator example to illustrate the temporal constraint generation. This excerpt realizes a small part of the language behavior described in the *denotational semantics* presented in Equations (5.27) to (5.36) on pages 99–100. It traverses the available Match statements and appends a new GreaterThanOrEqualsExpression, which conforms to the TimeAnnotation attached to the current Relationship. The presented code uses the internal SModel language, which allows to query and modify MPS models⁴, to perform the M2M transformations.

6.1.6 Language pragmatics and implementation modules

The implementation of language *pragmatics* and *model* checking features follows the description of the implementation-independent language composition (cf. Figure 5.2 on page 82). As a result, these features are split off into independent *modules*, languages, or behavioral

⁴ See <https://www.jetbrains.com/help/mps/smodel-language.html> for detailed information

aspects of existing languages. *Pragmatics* implemented⁵ in the *vertical prototype* of this thesis include:

- 1) **Query execution:** Capability to directly execute a query from within the *IDE*. Queries are transformed, send to a Neo4j *GDB* via the REST *Application programming interface (API)*, results are retrieved and lastly displayed to the users.
- 2) **Query pattern visualization:** The (sub-)graph pattern of a query are additionally visualized next to the query in a separated projection of the *abstract syntax tree (AST)*.
- 3) **Temporal query constraints:** Attaching temporal constraints onto queries is enabled using *MPS*'s intentions aspect.
- 4) **Extraction of nodes, patterns, or other local variables:** Common helpers allowing to transform a query by extracting parts of it into own *modules*.
- 5) **Automatic local variable name specifications:** Convenience functionality, which generated local variable names based on the used concepts.
- 6) **Query analysis and visualization:** Capability to analyze a query from within the *IDE*. Similar to the execution feature, but uses the Neo4j internal EXPLAIN functionality to obtain a query analysis.
- 7) **Language composition visualization** Provides a visualization of the underlying language composition by traversing the dependency tree and generation a graph using the DOT format. This feature is mainly for development purposes to ensure that the theoretical language composition is realized and to identify necessary changes.
- 8) **IDE generation** The generation of an independent domain-specific *IDE* is implemented using the *MPS* internal Build language.

In the following paragraphs, I highlight *pragmatics* implementation of the exemplary items 2) the query pattern visualization, 6) the explain feature selected, and 7) language composition visualization.

```

ReadSinglePartQuery
MATCH [ Entity_Instance ] — [ Relationship_Instance ] — [ Entity_Instance ]
      << Conversation >>
      aConversation
      < no property >
      << contains >>
      aContains
      < add property >
      << ConversationTopic >>
      aTopic
      Properties:
      string type = "Greeting"
    
```


Graphical Patterns

```

<no where>
RETURN COUNT(aContains) < AS ... >
    
```

Use default hints
 Use custom hints:

com.dsifoundry.plaintextgen

Structural: View under-the-hood structure

de.citec.cypher.explain

queryExplain: Explain a query and show expected results

de.citec.cypher.visualisation

patternGraphical: Use a graphical representation for all graph patterns in MATCH clauses
 patternPartGraphical: Use a graphical representation for individual graph patterns in MATCH clauses

jetbrains.mps.baseLanguage

VisibleExpressionBoundaries: Visible Expression Boundaries

Figure 6.10: Custom editor hits are used to provide an additional *concrete syntax* projection of a query. The depicted example shows a visualization of a query as a graph to foster the understanding of pattern structure.

6.1.6.1 Visualization

Figure 6.10 show an example query with an activated visualization for existing patterns in a MATCH clause. This visualization is realized using MPS's *editor hints* feature allowing to provide projections (i.e. coexisting different *concrete syntaxes*) for any defined concepts. Figure 6.11 on the next page shows the corresponding editor hints implementation of this projection. The Cypher `.visualisation` language provides its projection via a separated editor definition for the existing concept `Pattern` of the Cypher language. The existing *concrete syntax* is reused and the graphical representation embeds below as a `$swing component`, which provides a Java swing graphical user interface object. This swing object is further defined in the inspector view at the bottom of Figure 6.11, where implemented the Java code creates and returns the `PatternPrefuseVisualisation` object. All custom Java classes necessary are bundled with in the language behavior aspects, while the library runtime (i.e. *prefuse*) is contained in an individual runtime solution.

⁵ Most implemented *pragmatics* are prototyped and further implementation refinement is required for a fully featured release

6.1.6.2 Query analysis

The query analysis feature provides an integrated view for developers which provides means to explain a given Cypher query. Similarly to the query pattern visualization feature, the explain feature is implemented as an implementation *module*. Additional information is provided as an additional projection which can be enabled by the users at any time. When activated as shown in [Figure 6.12](#) on the following page, the projection adds an explain section below the query with a button to trigger query explanation. After triggering, the projection provides the query developer with information on a) the actual query, which is generated from the MPS representation, b) metadata of the query plan, which will be used by the Neo4j Cypher engine, and c) a visualization of the query plan containing further plan execution information for each step. From the information presented, developers acquire an estimation of how many rows a given query will return for each step of the query execution plan. Most importantly, costly query steps such as Cartesian products (e.g. by expressing two or more unrelated matching patterns) can be identified by the developers and ideally be removed to optimize the query execution duration. Technologically, the implementation makes use of the Cypher internal EXPLAIN feature. The query engine can provide this information of any valid query by prepending the EXPLAIN keyword. Pressing the *Explain this Query!* button thus triggers the following steps in the plugin:

1. The MPS query text generator is activated to obtain the plain text representation of the query,

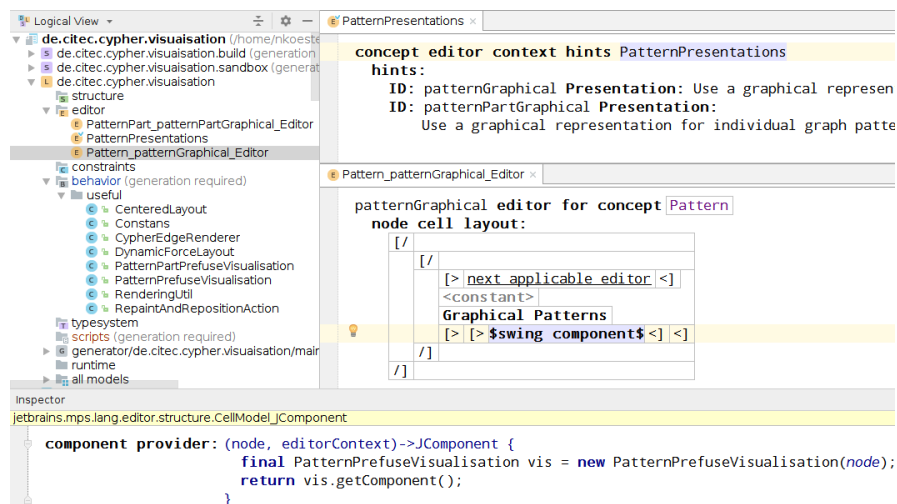


Figure 6.11: Editor aspects implementation used to provide the additional visualization depicted in [Figure 6.10](#) on the preceding page. The Java based visualization attaches a swing interface component to the existing *concrete syntax*.

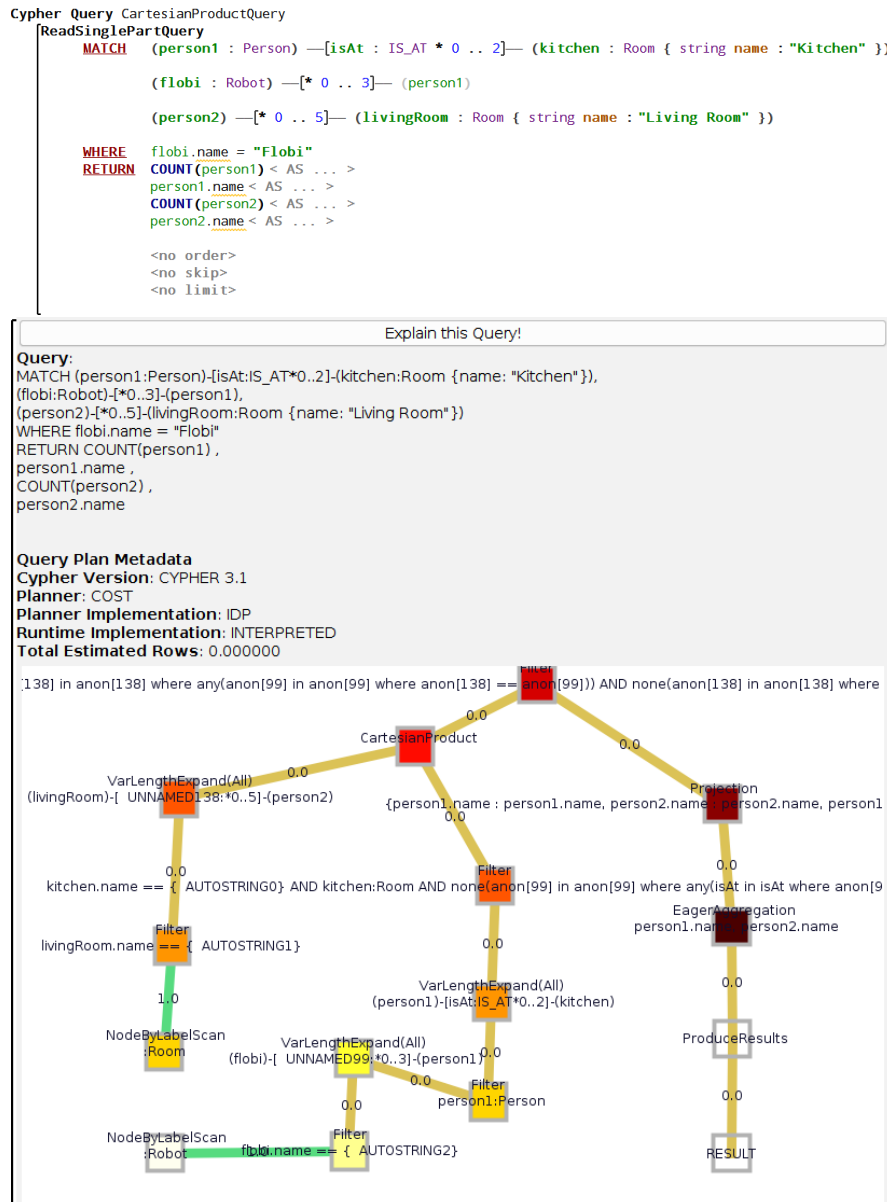


Figure 6.12: Example of the query explanation feature used to explain a Cypher query. The result is embedded below the query and shows information on the actually generated query, query plan metadata, and the full query plan as a graph.

2. The query is sent to the Neo4j database query engine via the provided REST web service interface
3. The analysis is done by the engine and sent back to MPS
4. The returned JSON document is parsed, transformed, and displayed to the user

As solely valid queries can be analyzed via the EXPLAIN mechanism, this feature is executable only on error free queries and also needs to be triggered manually by the users.

6.1.6.3 Language composition visualization

To mitigate *DSL* composition challenges (cf. [Section 3.1.2.1](#)), I developed the `de.citec.dependencydiagram` *MPS* plug-in, dedicated to the creation of dependency graphs. The plug-in shows the exact composition of all involved *modules* and *models*. [Figure 6.13](#) shows an exemplary dependency graph, which is generated on the basis of the `de.citec.dependencydiagram` plug-in itself. This overview allows to identify potentially erroneous and unintended dependencies amongst language compositions. Internally, the *module* structure is analyzed and a representation of the structure is realized using the *DOT* graph description language. Common tools which support the *DOT* notation generate visualizations from this representation and even large compositions can simply be generated and layout automatically.

To ease language composition I make heavy use of *Devkit modules*, which allow to group multiple *MPS modules* together and expose their composition as a single unit. Thus, users solely import the according *Devkit* (e.g. `de.citec.dependencydiagram.devkit` for the dependency graph plug-in) within their user *solution* and can begin creating their own *models*.

6.2 AUTOMATION ASPECTS IN APPLIED MDSE RESEARCH

Several supporting aspects have been implemented as part of Phase [P4. Automation](#) of the development process presented in [Section 3.3](#) on page 44. These provide accessibility to *artifacts* and languages for

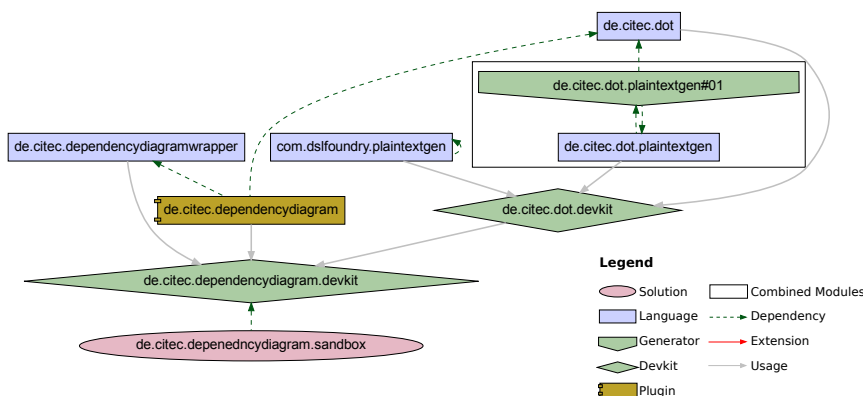


Figure 6.13: Dependency diagram of the dependencies required for a *solution* (red ellipse) realizing the dependency diagram *MPS* plug-in. This includes the *Dependant Devkits* (green diamonds), *plugin solutions* (yellow component rectangle), *languages* (blue rectangles), and *generators* (green upside down house shape). *Modules* which reside in the same language are grouped by surrounding boxes.

users and integrate the overall tool into the research application environment. This section describes two exemplary automation aspects, which were implemented: i) building and integrating *DSLs* and ii) deploying languages to the final users.

6.2.1 Continuous integration of *DSLs*

In small projects consisting of only few languages with few dependencies, *artifact* generation via a manual build process from within the *MPS* development environment is feasible. However, given the complexity inherent in the language design and composition of my approach, systematic building of the involved *modules* and an appropriate deployment strategy are critical elements which require intensive consideration. Additionally, the *models* designed by the domain experts need to remain valid over the time of use. In the case of version updates – either of my own languages or of the used *language workbench* – I need to provide reproducible *artifacts* and be able to execute strategic *model* migrations.

I consider it thus necessary to use common automation tools, such as the *Jenkins* [Jen] *Continuous Integration (CI)* server, to generate the intended build *artifacts*. *MPS* allows to build *artifacts* in a headless mode via the build process automation tool *Ant* [Apa00]. To guide this process, developers can use the included *BuildLanguage* *module* to compose their *modules* and *artifact* layouts. In either case, *module* dependencies must be present within the used *MPS* instance (e.g. as a loaded plug-in) or provided as build arguments pointing to the folders with the necessary *artifacts*. Thus, when deploying composed *MPS* languages via a *CI* server, the knowledge about dependencies is duplicated in the languages themselves and in the topology of the build jobs. Maintenance of these duplications is an error prone task which can lead to missing or faulty dependencies and thus incompatible or nonfunctional *artifacts* (and in edge cases even influence user *models*). Further, in scenarios where reproducibility of the overall system is required, the reproduction process suffers as in-depth knowledge about the *MPS* *module* topology is required for the setup of a build system.

To mitigate these risks I use the *CITk* as proposed by Lier et al. to create a reproducible build setup for my *modules* [Lie+14b]. To include *MPS* *modules* within the *CITk* system I created a template for *MPS* based projects (c.f. [Listing 6.1](#) on the next page) that summarizes common project properties. I include each individual *module* in the *CITk* as a project based of the *MPS* template, for example [Listing 6.2](#) on page 126 shows the resulting project file for the Cypher language. In the generation step the *CITk* build generator extracts dependency knowledge per project from the provided *Ant* files and creates or updates the corresponding *Jenkins* jobs. As a result, I eliminate

```

1 variables:
2   natures:
3     - mps-plugin-build-file
4
5   platform-requires:
6     ubuntu:
7       packages:
8         - ant
9         - '{@next-value|[]}'
10
11   default-build-file:
12     - build.xml
13   build-file-name: |
14     ${next-value|${analysis.plugin-build-files|${default-build-file}}}
15
16   home-variables: ${next-value|${analysis.home-variables}}
17
18   extra-requires:
19     - nature: program
20       target: mps.sh
21       version: ${mps-version}
22     - '{@next-value|[]}'
23
24   aspects:
25     - name: mps.shell
26       aspect: shell
27       variables:
28         home-variable-options: '-D${home-variables}=${dependency-dir}" '
29         ant-calls: |
30           ant -Dmps_home=${dependency-dir}/mps-${mps-version}"
31             @home-variable-options}
32             -file "${build-file-name}"
33         aspect.shell.command: |
34           # Invoke ant for each plug-in build file
35           @ant-calls}

```

Listing 6.1: The template for MPS based projects within the *Cognitive Interaction Toolkit (CITk)*.

dependency duplications as the dependency knowledge only resides within the *modules* themselves and is solely extracted as dependencies into *Jenkins* build jobs.

For example, [Figure 6.14](#) on the next page shows the resulting *DSL* and *module* build dependency graph as extracted from a *Jenkins* server. The nodes of the graph represent individual *Jenkins* jobs which are connected to all other dependent jobs. Once the dependencies of a *module* are build the downstream jobs are triggered, finally building the entire *module* stack. The shown graph was generated by the *CITk* build generator tool processing the build project file which provided an early version of the *vertical prototype*.

```

1  templates:
2  - code-corlab
3  - mps
4  - base
5
6  variables:
7  description: MPS Cypher Language
8  keywords:
9  - dsl
10 - cypher
11 - neo4j
12 access: private
13
14 repository: ${redmine.instance}/git/cypher-dsl.cypher-language.git
15 scm.credentials: code.corlab
16
17 build-file-name:
18 - "build-plugin-cypher-mps.xml"
19
20 versions:
21 - name: 2018.2.1
22   variables:
23     mps-version: 2018.2.1
24     branch: "2018.2"

```

Listing 6.2: Exemplary project file for the Cypher *DSL*.

6.2.2 Language deployment: A DSL plug-in server

The next step for manageable language deployment is the distribution of *module artifacts*. For MPS based *DSLs* development these are commonly *IDE* plug-ins. At first, the end-users obtain a generated *IDE*, which contains the current version of all necessary plug-in and settings to begin the modeling of the domain. These settings also contain an update *URL* pointing to a server that provides updates of the included *modules*. Further *artifact* updates are then deployed to the

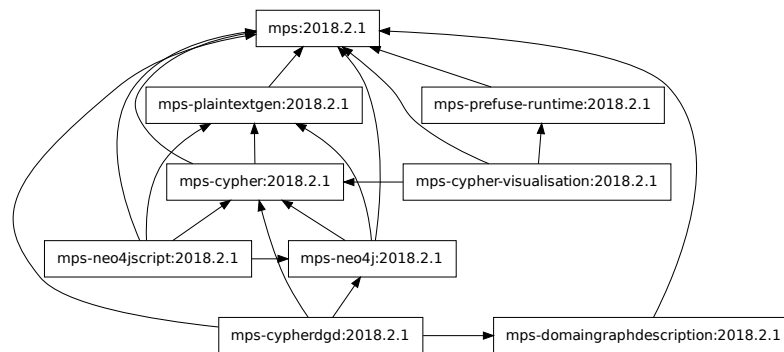


Figure 6.14: MPS *module* build dependency graph generated from the Jenkins job layout. Contains the composed languages used to build a *vertical prototype*.

```

1 <plugin-repository>
2   <ff>"MPS"</ff>
3   <category name="MPS">
4     <plugin date="1543249645.5626543"
5       ↪ url="127.0.0.1/plugins/de.citec.time-1.0.0/
6       ↪ de.citec.time-1.0.0--MPS-181.4445.78-2018.1.zip">
7         <id>de.citec.time</id>
8         <name>de.citec.time</name>
9         <version>1.0.0</version>
10        <idea-version since-build="181.0"
11        ↪ until-build="181.4445"/>
12        <depends>jetbrains.mps.core</depends>
13        <depends>com.dslfoundry.plaintextgen</depends>
14        <description>
15          <!-- detailed plugin description ... -->
16        </description>
17        <vendor url="https://cit-ec.de/cse">
18          <!-- detailed vendor description ... -->
19        </vendor>
20      </plugin>
21    <!-- further plugins ... -->
22  </category>
23 </plugin-repository>

```

Listing 6.3: Example entry within the *updatePlugins.xml* database of the MPS plug-in server.

users via the MPS internal plug-in update mechanism. The end-users are notified if updates to the used languages exist and a seamless update process is offered. This ensures consistency and stability of end-user *models* as each update will apply all provided version update migrations.

To allow this seamless integration I developed a plug-in server to maintain and deploy an *Extensible Markup Language (XML)* based plug-in database, as required for the MPS plug-in system. The database consists of the commonly named *updatePlugins.xml* file (cf. Listing 6.3 for an exemplary file content), containing minimal server information (lines 1-3 and 19-20) and entries for each plug-in (lines 4-17). As I distribute each individual plug-in as a compressed ZIP file via the CI server (cf. 6.2.1), the plug-in server simply directly extracts the required information from the packaged *plugin.xml* file. The plug-in server architecture primarily follows the *Observer* pattern [Gam07, p. 293]. It observes a staging folder to which newly build plug-in are copied after successful building by the CI server. Each new build is analyzed by the server and compared to the current state of the plug-in repository. Additions and changes are added to the repository and the new plug-in is copied to a target folder accessible by the connecting clients. The server creates a new unique sub-folder in the target folder for each plug-in to support the parallel existence of plug-ins of various versions within the repository

6.3 USER PERSPECTIVE: THE EISE QUERY DESIGNER

The languages presented in the previous sections are combed into a standalone workbench for the EISE domain, the *EISEQD*. This integrated *solution* is build and distributed using the *CI* infrastructure as described in Section 6.2.1. The individual languages are internally deployed as plug-ins into the *IDE* and updates can be generated and pushed to the clients using the plug-in server and the MPS internal plug-in architecture as described in Section 6.2.2. All languages listed in Table 6.1 on page 110 are bundled into the *EISEQD* so that the features described in Section 6.1 are jointly available. *Behavior developers* download the complete package *artifact* which contains the MPS runtime as well as all languages directly from the Jenkins server. They can create new *DomainDescriptions* or use already available descriptions from a project repository. Figure 6.15

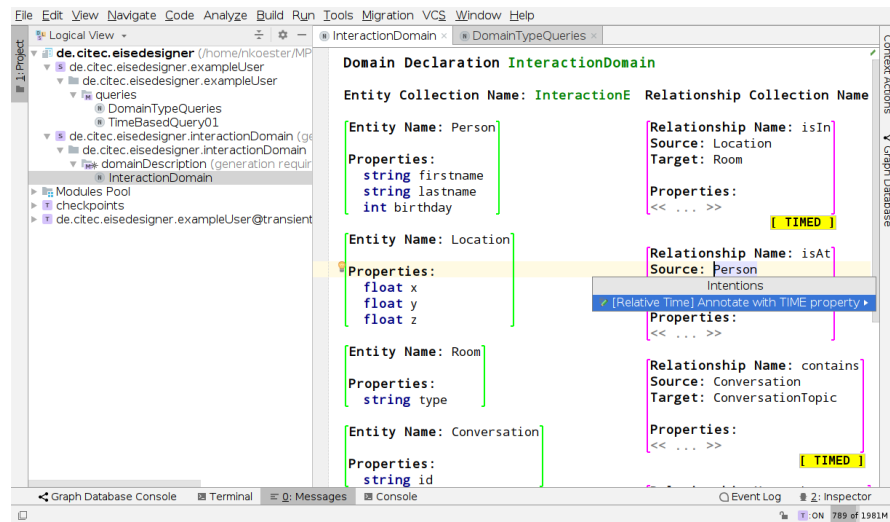


Figure 6.15: Screenshot of the *EISEQD* interface editing a *DomainDescription* concept which outlines description of the interaction domain. Individual Relationships or Entities can be annotated with the time property indicating that temporal information is recorded in the nodes properties.

shows a screenshot of the tool. The project view shows two dedicated *solutions*. The first *solution* contains a *DomainDescription* modeling the *EISE* domain and the second *solution* holds queries targeting this domain. The domain declaration for is being edited in the main view. While the left column of a *DomainDeclaration* concept allows to define the *DomainNodes* of the domain (*DomainNode* are surrounded with green colored brackets), the right part allows to define *DomainRelationships* from one *DomainNode* to another (each *DomainRelationships* is surrounded with magenta colored brackets)⁶.

⁶ The *concrete syntax* removes the technological details and simply displays them as *Entities* and *Relationships* to the users to reduce the complexity.

```

1 MATCH (aConv:Conversation)-[aContains:contains]->
2         (aTopic:ConversationTopic {type: "Greeting"}),
3         (aConv2:Conversation)-[aContains2:contains]->
4         (aTopic2:ConversationTopic {type: "Goodbye"})
5 WHERE (aContains.ts >= (timestamp())[1000][5.0] AND aContains.ts <=
6         ↪ (timestamp())[1000][5.0] AND
7         (aContains2.ts >= (timestamp())[1000][10.0] AND aContains2.ts <=
8         ↪ (timestamp())[1000][5.0])
9 RETURN COUNT(aConv) + COUNT(aConv2)

```

Listing 6.4: Generated Cypher query code from the finished query shown in Figure 6.16. The temporal constraint are expanded into the corresponding WHERE clause filters filtering the query results according to the defined temporal expansion.

Each entry in the DomainDeclaration can contain DomainProperties whose Types are taken from the basic type language of the internal Baselanguage. The temporal constraining feature allows to attach a TimedElement annotation to any Entity or Relationship to the DomainDeclaration concepts, indicating that these concepts follow the temporal *model* as presented in Section 5.4.4 on page 96. The annotations are added using the *Intentions* mechanism of MPS. Figure 6.16 on the following page shows a screenshot of the content of one of the query sheets. The shown query makes use of the DomainNodes and DomainRelationships concepts defined in Figure 6.15 and creates suitable DomainNodeInstances and DomainRelationshipInstances respectively. For missing entries the suitable completion is provided, in this example only Conversation are connected to ConversationTopic concepts via a contains relation, thus only this concept is shown in the completion. These DomainElementInstances act as local variables and can be reused in the query as expected. The matching Conversations are for example counted in the RETURN clause via the internal COUNT function. Further, correct intentions are provided to the users to add these kind of functions around existing concepts. In the shown example the visualization additionally shows the subgraph which is being matched. As the query patterns do not relate to each other, the graph is separated and two subgraphs are shown. At the top of the query a temporal constraint is added to the query. This constraint is applied on the entire MATCH clause and all DomainElementInstances which are annotated with a TimedElement will be restricted as described in the semantics in Section 5.4.4 on page 91. In the depicted example, an Interval is considered which starts ten seconds before the execution and ends five seconds later. A resulting Cypher query string obtained from the generator is shown in Listing 6.4⁷. All concepts are generated to the corresponding Cypher *concrete syntax*, including local variable names, labels, types, and properties within the MATCH and RETURN clauses in lines 1 and 7. Even though no WHERE clause is defined in the original query, a filter clause

⁷ Generation was obtained once the errors were removed from the query.

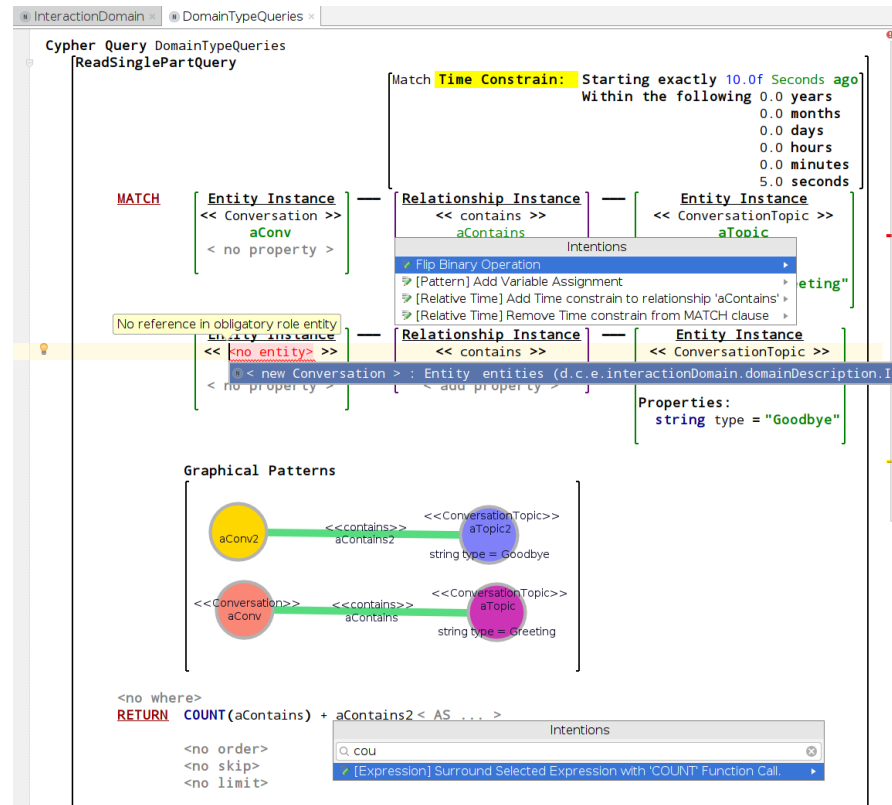


Figure 6.16: Screenshot of the query editor in the EISEQD containing an exemplary query based on the interaction domain description in Figure 6.15. Multiple query design supporting mechanisms are shown, for example intentions for time constraint annotation, concept completion dialogs, or graphical pattern visualization.

was added by the generation step in line 5 and 6. This filter satisfies the temporal restrictions defined in the EISEQD and ensures that the `ps` property of the annotated concepts in Figure 6.15 on page 128 lies in the given interval.

6.4 SUMMARY

This section presents detailed excerpts about the implementation executed as a part of the development process Phase P3. **Language Implementation** of this thesis. These implementations are integrated into a *vertical prototype* which covers a functional vertical slice of the envisioned tool. Thus, an overview is presented on the implemented *DSLs*, the language composition, application modules, language *pragmatics*, and automation aspects. While the implemented languages conform to the implementation-independent *meta-models* presented in Chapter 5 on page 75, the implementation contains implementation-specific adjustments. This is done as the development of languages in the followed iterative process described in Section 3.3 on page 44 did not fully align with the theoretical considerations presented. These adjust-

ments were done to overcome technological challenges and tool limitations or constraints. This chapter highlights the central differences of the languages to their theoretical considerations in [Chapter 5](#). Further, an overview of the language's complexity (i.e. number of concepts, relationships, aggregations, or extensions) and implemented language *pragmatics* are provided. For each central language, the implementation specific changes to the *meta-models* are highlighted and the created *concrete syntax* is exemplary presented. As a part of the Phase [P4. Automation](#), the implementation of approaches for *CI* and *DSL* deployment are additionally presented. These allow the execution of a feasible deployment and maintenance strategy of languages and *modules* with low effort for the final users. Updates to *meta-models* and languages can be delivered automatically to the users resulting in a smooth user experience. These automation elements are used to integrate the *Model-driven Software Engineering (MDSE)* approach of this thesis into the example environment of the *CSRA* project. The final contribution of this chapter highlight the user perspective by presenting the generated *IDE* for the *EISE* domain, the *EISEQD*. This view exhibits usage examples and shows the applicability of the *MDSE* approach of this thesis by example. An earlier iterative version of the *IDE* presented here was used to execute a user evaluation. The following chapter presents the details of this evaluation and the results obtained from the analysis.

Part V

EVALUATION OF MDSE APPROACHES

The fifth part presents the quantitative and qualitative evaluation carried out using the *vertical prototype* of the *EISE Query Designer (EISEQD)*.

“As always in life, people want a simple answer ... and it’s always wrong.”

—Susan Greenfield
Neurochemist currently researching
Parkinson’s and Alzheimer’s diseases.

Empirical evaluation of software that originates from traditional software development processes is common practice [BBL76; Bro96; LHS08; FB14]. However, the evaluation and subsequent validation of a *Model-driven Software Engineering (MDSE)* process and its resulting *domain-specific languages (DSLs)* presents a more complex task, which is often overlooked [KBM16; GGA10]. Since the relevant concepts, relations, and other domain knowledge is often scattered, a precise definition of a baseline allowing for approach comparison is difficult to specify. This distribution of information can be mitigated by a preceding detailed domain analysis as part of the *MDSE* process to create appropriate formalizations and domain *models*. To show the improvements and benefits of an approach and its application one then needs to attend different fields of evaluation (qualitative and quantitative) at all stages of the language development (proof of concept, actual development, evolution, and maintenance). Additionally, evaluations are required to also investigate the viability of the improvements promised by *MDSE* approaches. As a result, *DSLs*, tools and other *artifacts* of the *MDSE* process are overall rarely evaluated systematically [KBM16; GGA10]. Thus, in this chapter I present the quantitative and qualitative evaluations I conducted to validate the previously presented approach and its primary result, the *EISE Query Designer (EISEQD)*. Further, as a part of the study description and execution the practical use of the *integrated development environment (IDE)* is shown.

This chapter presents results from Phase **P5. Evaluation** and Phase **P6. Application** of the development process and subsequently investigates research question **RQ5**. The user evaluation via a user study was executed using an earlier iteration of the *vertical prototype* presented in the previous chapter. Parts of the here described evaluation approach, the study technologies, the obtained results, and their discussion have previously been published by me and peer-reviewed by the community. This primarily includes the publications “Evaluating a Graph Query Language for Human-Robot Interaction Data in Smart Environments” presented during the STAF 2017 Collocated

Workshops and “Evaluation of a Model-driven Knowledge Storage and Retrieval IDE for Interactive HRI Systems” published in the International Journal of Semantic Computing [KWC18c; KWC19].

7.1 INTRODUCTION TO MDSD EVALUATION

In practice Neto et al. identified five levels of evidence an evaluation can provide regarding the usefulness of an *MDSE* approach: 1) speculation, 2) example, 3) proof of concept, 4) experience or industrial reports, and 5) experimentation (ordered from low to high evidence) [Net+08]. As an example for an application of a medium layered evidence type (i.e. proof of concept/industrial report) Kärnä et al. used and evaluated their developed *solution* in the context of product development [KTK09]. In their setup, only six users (familiar with the target domain) had to develop an application with their tool-chain. They qualitatively compared the outcomes along the three dimensions of developer productivity, product quality and the general *usability* of the tooling. To alternatively reach higher evidence, one can carry out an extensive case study analysis involving a large user base (experience/industrial reports). This evaluation approach is especially effective if a large user base already exists for the provided tools who make extensive use of its features and functionalities and can share their experiences. However, it is important to note that this is not generally applicable when developing *DSLs* for smaller domains such as the *Embodied Interaction in Smart Environments (EISE)* domain due to the low user base size.

Völter et al. presented an excellent example of a case study providing insights on benefits gained from the development of the *mbeddr* platform [Völ+19]. The *mbeddr* project successfully uses *JetBrains Meta Programming System (MPS)* to provide an *IDE* [ite17] with a set of integrated and extendable languages for embedded software engineering [Völ+12]. Their evaluation primarily targets the language engineering process using JetBrains *MPS* as a *language workbench*. They conclude that designing languages that handle complex domains and that are modular and scalable is possible using *MPS*.

However, multiple case studies which investigate the evaluation of *MDSE* processes showed that the evaluation itself is often simply ignored by language designers and never carried out properly [KBM16; GGA10]. Far worse, there is no systematical report culture on the design and execution of experimental validations of the languages or environments which emerge from the processes. The evaluations that are executed are often informal or anecdotal with little to no comparability and thus of low level of evaluation confidence. Further complications arise from the fact that an approach’s effectiveness is not measured at all, due to difficulty to formulate this metric. Kosar et al. thus correctly conclude that generally, the core *DSL* development phases

which are lacking investigation are domain analysis, validation, and maintenance [KBM16]. This disconnection from the systematical reporting culture stands in strong contrast to the otherwise systematical approach language engineers follow. *DSLs* and environments need to be evaluated and their effectiveness for the target audience needs to be assured.

Barišić et al. thus proposed a development process tightly involving the evaluation process for the usability of *DSLs* which is primarily applied during the development life cycle [BAG18; Bar+12]. The authors identify the mostly anecdotal evidence presentation in literature and thus created a development and evaluation process to be applied during language development. Besides an initial definition of *usability*, they conclude that *Quality in Use* [ISO9126] (withdrawn and succeeded by [ISO25010]) is the optimal evaluation target, as it covers effectiveness, efficiency, satisfaction, and accessibility in specific user-task scenarios. They encourage using multiple metrics, including questionnaires targeting the subjective measures such as cognitive load or perceived *usability*. For my following evaluation I extend this idea with features from integrated iterative testing approaches which focus on the analysis of pre-defined metrics [Weg+13; Bar+12; Bar13]. According to Barišić et al. the evaluation needs to span the entire *DSL* life cycle by assessing motivation, carrying out qualitative interviews, validating the *DSL* design, and quantifying benefits. Mixing quantitative and qualitative criteria is required in the evaluation process as the use of simple metrics such as *(Source) Lines Of Code (LOC)* are unable to cover all advantages and risks of the application of a domain-specific modeling solution [Weg+13]. Nevertheless, they conclude that each measurement itself is important and individual results need to influence the *DSL* development process.

✍ *usability*

Further difficulties in *DSL* evaluation arise from the fact that various *language workbenches* exist which do not share the same properties and features or make use of very different approaches (e.g. textual and projectional editing; cf. Section 3.1.2.5 on page 38). A cross-workbench comparison is thus increasingly difficult. The actual reporting culture on *DSLs* is thus often reduced to the presentation of the domain, its analysis, *concrete syntax* examples and a *meta-model* showing the implemented language. While all these elements are of great importance, effectiveness is measured often anecdotally via application and implementation in example domains. One approach to mitigate this issue for language developers within the *MPS language workbench* ecosystem is presented by Häser et al. [HFB16]. The authors identify the difficulties for language engineers to safely determine the effects of language design decisions on the *usability* of the languages for the end-users. They present a practical approach closing the development and evaluation loop to investigate *DSLs* and their effectiveness. To realize this, they created an integrated set of languages and plug-ins

within the *language workbench MPS* which allow to describe controlled experiments. These languages include the modeling of planning, operation, analysis and interpretation, presentation and packaging of results as parts of the environment and thus provides a data-driven language development support. Unfortunately, the created environment is not freely provided to use and I thus manually realized the process to create a similar evaluation study.

Cervera et al. for example presented a recent evaluation of a *MDSE* approach using multiple evaluation metrics [Cer+15]. The authors present a detailed description of the study setup and its execution alongside their goal to measure the usefulness and ease of use of the created tools and *DSLs*. The used measurements are the *Technology Acceptance Model* and the *Think Aloud Method* to gain insights into the end user thoughts about the provided tools. However, while the metrics are comparably concise and applicable, the *Think Aloud Method* solely covers the subjective perception of a participant. This is a deliberate decision by the authors as they perceive questionnaires as unreliable and biased. On the contrary, I argue that this decision is for the wrong reasons. The *Think Aloud Method* allows to gain additional insights into the user perspective and can be used to further investigate the *usability*. There are difficulties to compare the results obtained from this method and draw clear conclusions for the comparison to a baseline – in contrast to questionnaire and statistical analysis.

7.2 EVALUATION METRICS

Because there exists a wide range of metrics applicable within the *MDSE* process, I describe a concise selection of the most used and relevant evaluations. In this section I provide an overview on different quantitative and qualitative metrics I consider applicable and helpful for *MDSE* approach evaluation.

MDSE evaluation metrics

One of the oldest, most prominent, and tangible metric in software development and software engineering evaluation is the count of *(Source) Lines Of Code (LOC)*. It is consequently used in literature as the main metric for *DSL* assessment [Völ+19]. This metric gives insight on the effectiveness at design time (i.e. how much less code does a *DSL* user need to write) and also on improvements at compile time (i.e. the volume of generated *artifacts*). A common problem arises for *language workbenches* which make use of the projectional editing feature (such as *MPS*). *DSL* editors consist of individual cells which in turn can contain much more information as a single word and thus lines of cells are not easily mappable to traditional *LOC*. In this case it is common to approximate the *LOC* by estimating 4 editor cells per line [Völ+12; Völ+19].

(Source) Lines Of Code (LOC)

The recording and analyzing of keystrokes of study participants during task execution using the provided software is a similar quan-

titative measurement. With reduced overhead and boilerplate code necessary to write when using a special *DSLs*, one expects study participants to exhibit reduced, more specific and less error correcting inputs. This metric investigates the participants at design time rather than compile time and can additionally help to uncover repetitive tasks and common input errors.

Besides effort intensive direct observation of study participants and their physical reactions (e.g. eye movement, facial expressions, or heart rate), the most common approaches to quantitatively assess software product *usability* are questionnaire driven analysis during and after study task execution. The *System Usability Scale (SUS)* is an effective and reliable tool for measuring the *usability* of software components [Bro96]. It is appropriately short with only 10 items and it is intuitive to understand [BKMo8]. Comparability – an important measurement for *MDSE* products – is given, as the scale is uniformly interpretable. A potential study design needs to include a baseline condition for comparison and the application of the *SUS* on *MDSE* products can thus easily be done by researchers (refer to Figure A.1 on page 165 for the full questionnaire). Similarly to the *SUS*, the *User Experience Questionnaire (UEQ)* provides a further detailed analysis on product *usability* [LHSo8]. However, in difference to the *SUS*, the *UEQ* does not calculate a single comparison measurement but provides comprehensive impression of user experience. It investigates both *usability* aspects (efficiency, perspicuity, dependability) and user experience aspects (originality, stimulation). It categorizes these aspects by *attractiveness*, *perspicuity*, *efficiency*, *dependability*, *stimulation*, and *novelty*. The application itself is analogous to the *SUS* by incorporating the appropriate 26 items in a questionnaire subsequent to program usage (refer to Figure A.3 on page 166 for the full questionnaire).

Besides questionnaires targeting the *usability*, I consider the cognitive load of study participants during task execution as an important measurement. With one benefit of *DSLs* being the reduction of complexity by allowing domain experts to formulate problems in the concise language of the domain, the cognitive load of users is expected to not increase. Ideally, the complexity reduction also reduces the cognitive load as common concepts and abstractions of the domain can directly be used without the need to encode them in a different language. Further, the combination of different domains and their languages is also expected to not increase the load on users. A validated metric for the measurement of subjective cognitive load is the *NASA Task Load Index (TLX)* [HS88]. The measurement of the total workload is separated into six subscales that are presented to subjects on a single page between individual tasks, each on a scale from 0 to 100 points. These individual scales are *Mental Demand*, *Physical Demand*, *Temporal Demand*, *Performance*, *Effort*, and *Frustration* (refer to

✍ *System Usability Scale (SUS)*

✍ *User Experience Questionnaire (UEQ)*

✍ *NASA Task Load Index (TLX)*

Figure A.2 on page 165 for the full questionnaire). A systematic evaluation 20 years after its introduction further showed that there is no need to adjust the *TLX* scale to individual participants as a normalization measure and it is just as accurate without, thus simplifying its application further [Har06].

Lastly, the measurement of task execution duration and the calculation of an error rate for individual tasks can show whether or not a *DSL* reduces (user) execution speed and errors. The latter depends on the task specification and study design requires appropriate adjustment. For example, to calculate a syntactical error rate for individual tasks one can implement:

$$\varepsilon_T = \frac{1}{N} \left(\sum_{n=0}^N \frac{t_i^n}{(t_i^n + t_c^n)} \right), \quad (7.1)$$

where t_c^n and t_i^n are described by the amount of correct and incorrect task executions for participant n respectively.

Besides the aforementioned quantitative metrics, a multitude of additional qualitative measures need to be obtained by researchers to reach high levels of evidence [Net+08]. These qualitative metrics are important as they help to uncover more intangible issues with a tool which will not necessarily be covered by the metrics mentioned above. A straight forward method is to gather free comments and feedback as a part of the questionnaire from participants directly after their interaction with the tool. The users will most likely report on the biggest issues they encountered during the study execution which they potentially cannot voice in the questionnaires (e.g. unnecessary switches between mouse and keyboard can break concentration for some participants). Additionally, verbal feedback and discussions are a good tool to extract information beyond the usual items of the questionnaire. To identify further *usability* issues, recordings of the participants view (screen recording) as well as key and/or mouse inputs are a helpful. Qualitative (and also quantitative as described above) analysis of these can yield information on common problems with *DSLs*, tooling, and study tasks.

7.3 EVALUATION OF THE EISE QUERY DESIGNER

As shown in the previous sections, qualitative and quantitative analysis is required to reach valuable insights on tools created in the context of the *MDSE* process and validate their applicability. Besides the development of languages, their composition (Chapter 5) and the creation of the *EISEQD* (Chapter 6), I hence conducted an evaluation study to assess the approach applicability and improvements. I designed a user study with the goal to reach a high level of evidence showing the effectiveness of my approach [Net+08]. Participants were

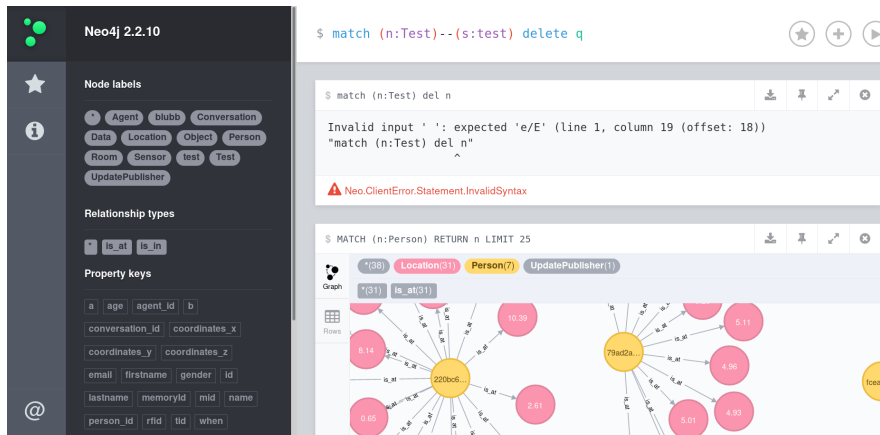


Figure 7.1: A screenshot of the Neo4j web interface. The left side of the interface gives limited domain-specific hints on node labels and relationship types as well as property keys. This information is only updated irregularly. The right side shows the query results with a successful query (bottom right) and a query containing a syntax error (center right). At the right top users can compose and send queries. Here an unsent query is shown containing a wrong reference to a variable (i.e. q) which is not identified or highlighted at design time.

requested to design queries in the Cypher query language and run these towards a given sample database.

I defined a baseline condition to successfully compare the results of my approach (the *EISEQD*) against. This baseline is designed to represent the usual workflow a real world user commonly follows when creating *Graph Database Management System (GDB)* queries towards a Neo4j database. Users designed and execute their queries with the Neo4j web interface which is depicted in Figure 7.1. They then subsequently copied the resulting query string into a text document for persistent storage. In contrast to this, the MPS condition made use of the *EISEQD* to design, execute, and store the queries. This results in the two conditions listed in Equation (7.2). Where Neo4j refers to the baseline condition and MPS refers to the test condition involving the *EISEQD*:

$$C = \{\text{Neo4j}, \text{MPS}\} \quad (7.2)$$

I identified three evaluation research questions (**ERQ₁-ERQ₃**) for my study along the three axis of effort, effectiveness, and intuitiveness of the used tooling among the two conditions C:

○ *evaluation research questions*

ERQ₁ What is the effort to design and execute a query towards the EISE domain when using either Neo4j or MPS?

Measurements: time, cognitive load, keystrokes

H₀ Null hypothesis: The effort is similar when using Neo4j and MPS to design a query towards the *EISE* domain.

- H₁ Alternative hypothesis:** The effort is lower when using Neo4j to design and execute a query towards the *EISE* domain.
- H₂ Alternative hypothesis:** The effort is lower when using MPS to design and execute a query towards the *EISE* domain.
- ERQ₂** How effective are users when designing and executing queries using either Neo4j or MPS?
- Measurements:** Amount of correct/incorrect queries, error rate, keystrokes
- H₀ Null hypothesis:** The effectiveness is similar when using Neo4j and MPS to design a query towards the *EISE* domain.
- H₁ Alternative hypothesis:** The effectiveness is higher when using Neo4j to design and execute a query towards the *EISE* domain.
- H₂ Alternative hypothesis:** The effectiveness is higher when using MPS to design and execute a query towards the *EISE* domain.
- ERQ₃** How intuitive is it to design and execute queries using either Neo4j or MPS?
- Measurements:** *UEQ* and *SUS* questionnaire
- H₀ Null hypothesis:** The intuitiveness is similar when using Neo4j and MPS to design a query towards the *EISE* domain.
- H₁ Alternative hypothesis:** The intuitiveness is higher when using Neo4j to design and execute a query towards the *EISE* domain.
- H₂ Alternative hypothesis:** The intuitiveness is higher when using MPS to design and execute a query towards the *EISE* domain.

7.3.1 *Methods and study design*

To answer research questions **ERQ₁-ERQ₃** and show the applicability of my approach, I designed a *between-group* user study (cf. [Figure 7.2](#) on the facing page) in which each participant solves tasks of varying difficulty [[MDFo5](#)]. The *between-group* design – where each participant only experiences one condition – was chosen over the opposing *within-subject* design to allow for statistical comparison between the two conditions C and show a relationship between the independent variables and the outcomes. This design choice minimizes the influence by external factors and provides an independent measure as every participant is only using one of the tools of one condition with similar previous knowledge and bias. Improvements due to increased practice and experience are ruled out and will not influence the results.

I avoided the typical *experimenter bias* by preparing all participants similarly with predefined textual material [[MDFo5](#)]. As a result, the

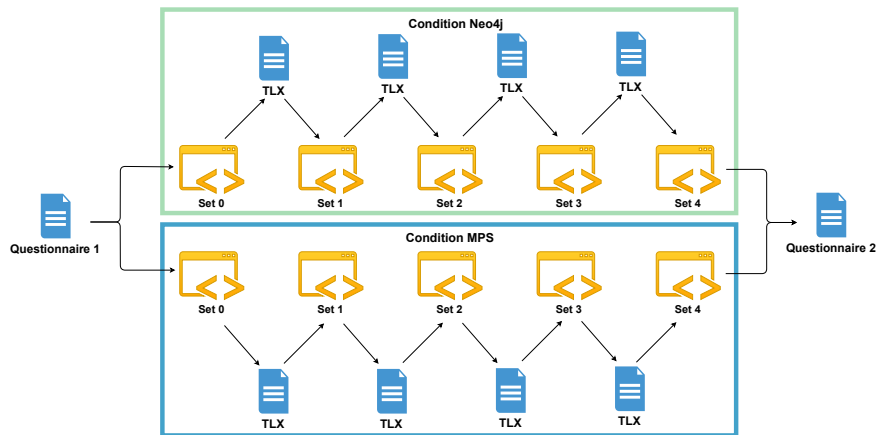


Figure 7.2: Depiction of the in-between study design implemented in the user study. Participant start with the initial Questionnaire (left) and execute each task set in alternation with a *TLX* questionnaire (center). The tasks in the individual sets are identical across both conditions. They finish the study after the second part of the questionnaire (right).

experimenter did not provide any information and hence did not unintentionally influence participant actions in any way. For this I created the following introductory preparation material:

- a) Cypher Information Sheet ([Appendix A.2.1](#) on page 167)

This document contains a summary of the most important principles on the Cypher query language. All elements of importance for the tasks are explained in detail alongside an example of their usage. Additionally, it provides a dense one page summary containing all relevant information for the use during the study.
- b) Embodied Interaction in Smart Environments Domain Information Sheet ([Appendix A.2.2](#) on page 171)

As the tasks for participants revolve around the *EISE* domain, a simplified and reduced domain representation as a graph is provided to the participants. It holds eight concepts, their properties and relations to each other as present in the database towards which the queries of each task are executed.
- c) Tool Information Sheet ([Appendix A.2.3](#) on page 172)

Depending on the condition, participants get an introduction sheet that summarizes the use of involved tools. For both condition the information sheet shows how to use the tools for query design and execution. Additionally, the task workflow is explained and the most important shortcuts are listed.
- d) *Task Sheet* ([Appendix A.2.4](#) on page 179)

The actual tasks each participant had to solve during the study

are listed in the task material. The study consists of five sets of queries **S0**, **S1**, **S2**, **S3**, and **S4**. Each set contains two to three individual queries

$$Q = \{S0Q1, S0Q2, \\ S1Q1, S1Q2, S1Q3, \\ S2Q1, S2Q2, \\ S3Q1, S3Q2, \\ S4Q1, S4Q2, S4Q3\} \quad (7.3)$$

participants had to solve. The queries are presented in natural language text alongside a hint to remove any ambiguity and ensure that all participants understand the goal of the query. Further, the expected result is listed so that participants can be sure that their created query is correct. As a time measurement helper, each query is also annotated with a maximum amount of time it should take to formulate the Cypher query. **S0** is an introductory set to familiarize the participants with the procedure and the *TLX* questionnaire, which has to be filled in after each set to capture the set specific subjective cognitive load. It is thus omitted from further calculations and evaluations. The remaining sets **S1** to **S3** raise in their difficulty level, each requiring new concepts of the Cypher language, and asking increasingly more complex questions. To investigate the learning effect of the participants, the final set **S4** is a permuted copy of **S1** targeting slightly altered variables.

Participant guidance (i.e. what to do next) and all the collection of all questionnaire based metrics was executed using the statistical survey web application *LimeSurvey* [Sch12]. Access to the survey interface was provided on a second independent computer next to the participant as humans react unconsciously to the computers they interact with and transfer experiences from one interaction to another [NMC99]. This data collection approach allows to obtain more honest responses about the system independent from its usage.

The study I conducted also followed the ethical requirements (cf. [Appendix A.4](#) on page 184 for a copy of the successful ethics application) as defined by the *Ethics Committee of the University of Bielefeld*¹, which in turn follows the corresponding rules of the “gemeinsamen Ethischen Richtlinien der Deutschen Gesellschaft für Psychologie und des Berufsverbandes deutscher Psychologinnen und Psychologen”. Informed and voluntary consent in written form was acquired by me from each participants prior to recording and data collection (cf. [Appendix A.5](#) on page 185 for the used consent form). Further, I

¹ <https://uni-bielefeld.de/uni/einrichtungen-organisation/zentrale-organisation/kommissionen/ethik/>

anonymized the data directly at recording time and further sanitized it by separating names and any identifying information from the data into an isolated database linking to participant identifiers. The data is additionally stored on encrypted devices to avoid unauthorized access and breakage of confidentiality.

Summed up, the study procedure for each participant was thus the following:

1. Introduction

- a) The participant is provided with the consent form and has to read, understand, and sign it
- b) The participant is handed all supplemental material and the experimenter leaves the room
- c) The participant fills in the initial demographic questionnaire
- d) The participant reads the introduction material consisting of
 - Cypher Information Sheet
 - Tool Information Sheet (based on the condition)
- e) The participant is able to ask questions of clarification or understanding
- f) The participant reads the task material consisting of
 - Embodied Interaction in Smart Environments Domain Information Sheet
 - Task Sheet
- g) The participant starts the test set S0
- h) The participant fills in the *TLX* questionnaire
- i) The participant can ask the experimenter final questions of clarification or understanding with respect to tools and execution

2. Study Execution

- a) The participant solves each set and fills in the *TLX* questionnaire until all sets are done (with no option to ask the experimenter at any point)
- b) The participant fills in the *SUS*, *UEQ*, and free feedback questionnaire

3. Wrap-Up

- a) The study ends and an open feedback discussion is initialized

7.3.2 Measurements

To properly evaluate the *EISEQD* quantitatively and qualitatively in this study, I made use of the metrics presented in [Section 7.2](#) on page 138. This includes

A Quantitative

- *SUS (System Usability Scale)*: 10 item questionnaire, Likert scale [Lik32] ranging from -2 (“strongly disagree”) to +2 (“strongly agree”)
- *UEQ (User Experience Questionnaire)*: 26 item questionnaire, seven-stage scale ranging from -3 (“attractive”) to +3 (“unattractive”)
- *TLX (NASA Task Load Index)*: 6 item questionnaire, scale ranging from 0 “very low”) to 100 (“very high”)
- Keystrokes and mouse movement
- Error rate
- Demographic data

B Qualitative

- Participant view (screen recording)
- Feedback regarding the used tools, the study itself, and other (free text input after the questionnaire and verbal feedback via interviews)

Due to the nature of each metric’s application, the *TLX* (measuring the subjective cognitive load of a participant) is the only metric which has to be applied during the experiment. It has to be filled out by participants between each task execution and thus classifies as an *obtrusive measurement* [MDF05]. This is necessary as the *TLX* is – to the best of my knowledge – the best metric to obtain meaningful and comparable cognitive load measurements. The other applied metrics are *unobtrusive measurements* and thus do not influence the study execution.

The recorded keystrokes are categorized into the five following categories to allow a separation and more detailed analysis of differences in participant intentions.

1. Insertion

All keystrokes done with the intention to create a query

2. Deletion

All keystrokes done with the intention of deletion of characters or query elements

3. Navigation

Any usage of arrow keys, or other keyboard based navigation

4. Other

All remaining input done during the task which does not belong to previous categories

7.3.3 Study results

Participants of the study were obtained via bulletin within the *Cluster of Excellence Cognitive Interactive Technology at Bielefeld University (CITEC)* institute and satisfied the required knowledge levels (i.e. basic knowledge of the *Structured Query Language (SQL)* query language and basic programming knowledge). Additionally, the study participants were randomly assigned to each of the two conditions. In practice, a successful study run took each participant between 60 and 70 minutes. In total 28 persons participated in the study, however due to execution errors two participants had to be removed from the final data set, resulting in the final sample size of $N = 26$; equally distributed across the condition.

To compare the different conditions and to identify significance in the data, I used the non-parametric *Wilcoxon signed-rank test* over the alternative dependent samples *t-test* [Wil45; FMF12]. This decision is based on the fact the final sample size of 13 participants per condition is too low for a t-test, which is only applicable with a larger sample size (i.e. greater than 20 per condition). Notable significant differences between measurements are indicated via the '*' notation and were calculated with a confidence interval of $p < .05$, unless stated otherwise. Results from all metrics are depicted in [Figures 7.3 to 7.7](#) on pages 148–150.

The two metrics execution duration and *TLX* were measured per set and averaged over all participants for each condition. The error rate \mathcal{E}_Q^C is calculated for each individual query Q (cf. [Equation \(7.3\)](#) on page 144) of each condition C (cf. [Equation \(7.2\)](#) on page 141), which modifies [Equation \(7.1\)](#) on page 140 to

$$\mathcal{E}_Q^C = \frac{1}{N} \left(\sum_{n=0}^N \frac{q_i^n}{(q_i^n + q_c^n)} \right), \quad (7.4)$$

where q_c^n and q_i^n are described by the amount of sent queries with correct and incorrect syntax for participant n in condition C respectively (cf. [Figure 7.5](#) on page 149). \mathcal{E}_Q^C thus describes the rate in which each participant sent syntactically incorrect queries to the database.

I calculated the keystrokes metric ratios (cf. [Figure 7.6](#) on page 149) individually for each set via

$$\mathcal{K} = \sum_{s=0}^4 \frac{\frac{1}{N} (\sum_{n=0}^N I_{n,s}^{MPS})}{\frac{1}{M} (\sum_{m=0}^M I_{m,s}^{Neo4j})} \quad (7.5)$$

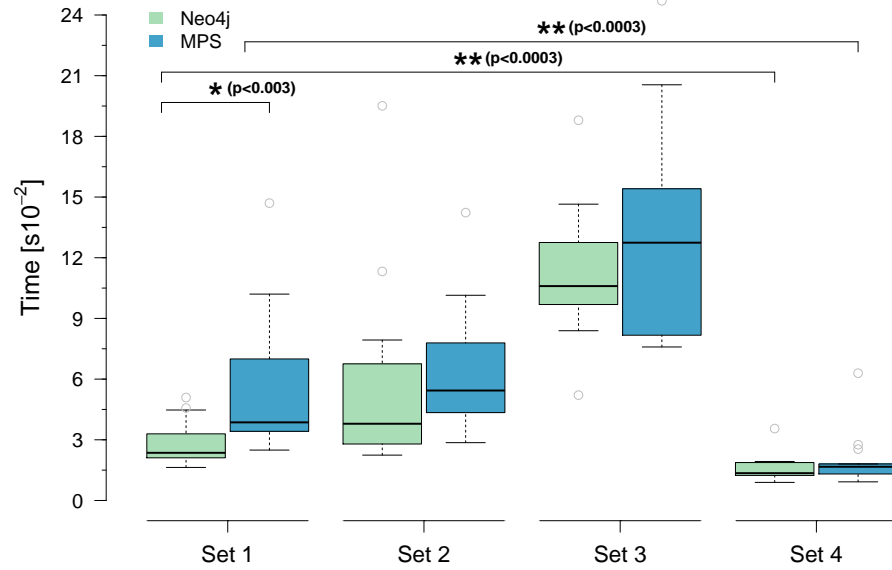


Figure 7.3: Boxplot of the time it took participants to complete the tasks in each set S .

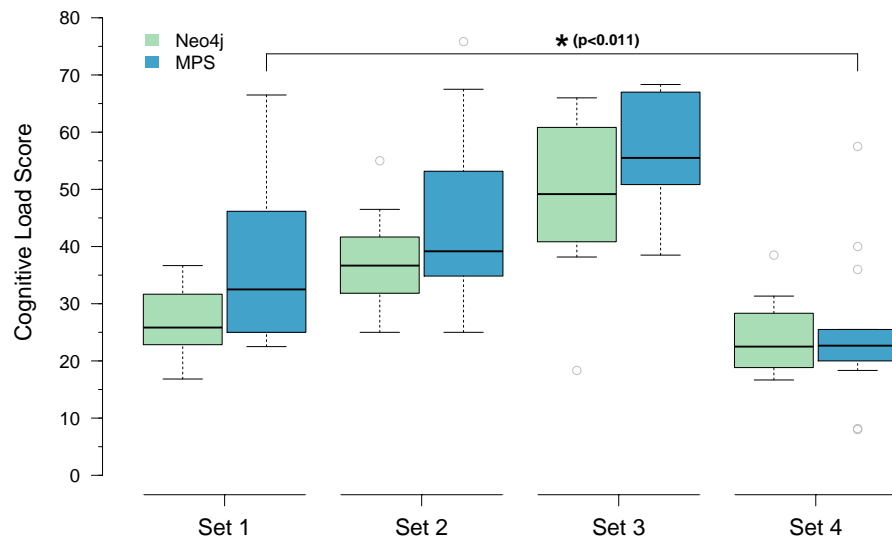


Figure 7.4: Boxplot of the cognitive load measurements obtained via the *TLX* questionnaire for each set S .

where $I_{n,s}^{MPS}$ and $I_{m,s}^{Neo4j}$ represent the inputs of each participant m or n for set s of the two conditions **MPS** and **Neo4j** respectively. This assumes that the average keystrokes in the **Neo4j** condition (the baseline) is a representative measure for the difficulty of the task and hence is usable for normalization and calculation of this ratio measure. To allow a detailed analysis of this assumption, I separated the keystrokes of the participants into the aforementioned categories.

The *UEQ* and *SUS* questionnaire values are calculated according to their documentation and are summarized in Figure 7.7 [LHSo8; Bro96]. All user feedback gathered during the study is listed in Ap-

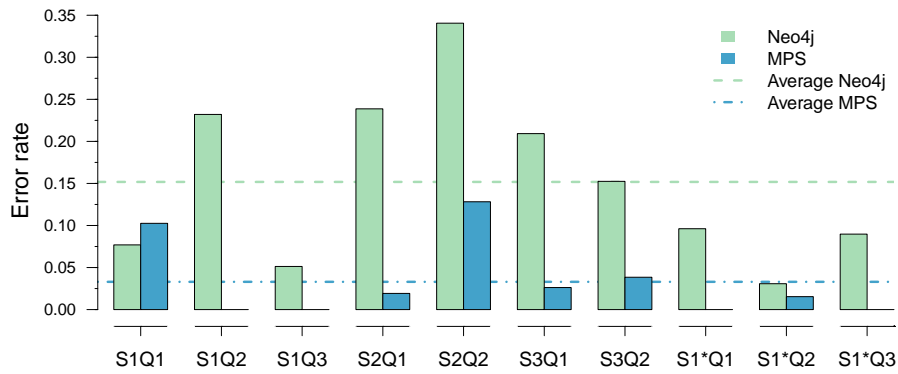


Figure 7.5: Rate of the averaged syntactical errors per query as calculated by Equation (7.1) on page 140.

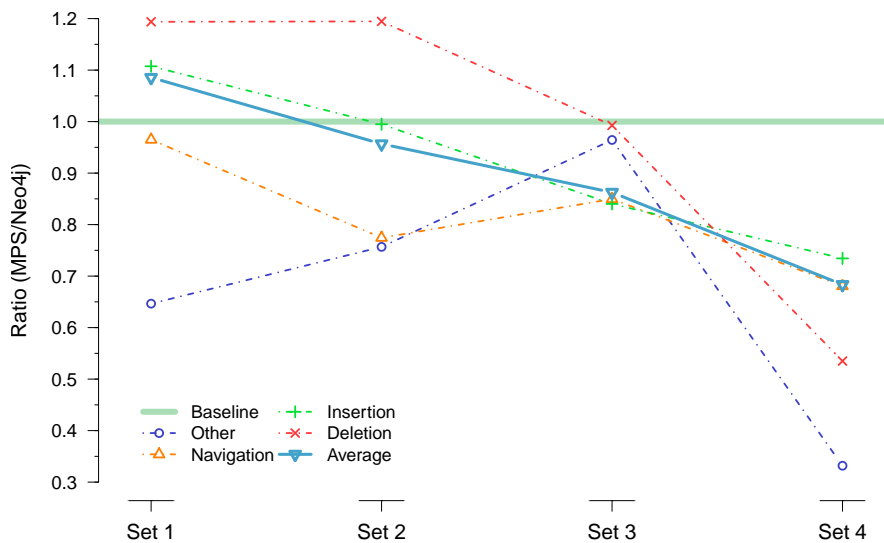


Figure 7.6: Results of the keystrokes metric calculated using Equation (7.5) on page 147. Inputs are separated into five categories and show the relation from condition MPS to the baseline condition Neo4j.

pendix A.6.1 on page 186 and omitted by me at this point due to its length.

7.3.4 Discussion

The duration of task execution and the *TLX* values increase as expected alongside the rising difficulty from **S1** to **S3**, while the permutation **S4** exhibits significant improvements over **S1** within both conditions. Additionally, the keystrokes analysis shows that generally the required user input for condition **MPS** reduces with each set. These results indicate that participants gain expertise between tasks and can perform already seen tasks with less effort, thus confirming the expected learning effect in both conditions.

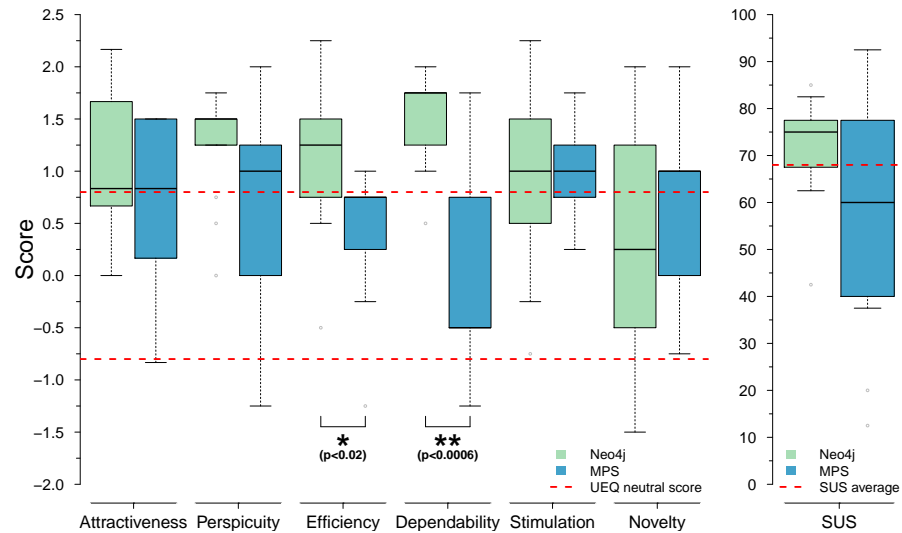


Figure 7.7: Results of the *UEQ* questionnaire categories (left) and *SUS* questionnaire score (right). Questionnaire specific neutral and good areas are indicated using a dashed red line.

Results from the error rate analysis indicate evince that participants within the condition **MPS** execute significantly less queries with errors for all queries except the first query **S1Q1**. Via qualitative analysis of the screen recordings, I attribute the initial query errors to the unfamiliarity with the projectional editing scheme. Users initially explored the *DSL* practically in the first query and became familiar with the input and deletion behavior. Generally, the results from the error rate calculations are expected. Due to the projectional editing and the thus resulting strong link between *concrete syntax* to the *abstract syntax* tree in the background, users of the **MPS** commit less errors. However, it raises the question why these strong constraints do not fully disallow any syntactical errors. I thus qualitatively verified in the recordings that each error recorded by this metric was actually displayed as such within the IDE to the users. The *EISEQD* development prototype warned its users if they were about to send erroneous queries to the database and all participants deliberately executed the queries – against the tool recommendation. In later discussions few participants stated that they “do not trust the tool” and thought “its error reporting is wrong”, thus they force executed queries they considered correct. This is also reflected in the highly varying results for the *UEQ* category *Dependability*. However, the task execution (in terms of duration and cognitive load) is not significantly impacted by the used tooling in either condition. At the same time participants in the **MPS** condition profit from the additional support resulting in less errors and also less inputs required to formulate the queries. Overall, these results suggest that the model driven approach is more effective (with respect to errors and inputs) and validate hypothesis

ERQ_1H_2 and ERQ_2H_2 over their alternative hypotheses H_0 and H_1 respectively.

The *usability* metrics generally exhibit an increased performance of the baseline condition **Neo4j**. The *SUS* score for **MPS** holds a large variance with its average below **Neo4j** and the global *SUS* average of 68 [Bro96]. This score is in the first quartile rating and marginal low acceptance area and thus above the adjective rating “OK” [BKMo8]. The variance in the *SUS* score is also reflected in the *UEQ* categories *dependability* and *perspicuity*, the participants feedback, and individual qualitative analysis of screen recordings: users have the feeling of control loss and thus consider the *EISEQD* as not dependable (e.g. being unable to delete individual tokens but only complete concepts). Some participants also stated that they did not understand the general idea of the underlying *model* of the *EISE* domain. In contrast to this, participants in the **Neo4j** condition actually used production ready tooling with familiar edition scheme, which yields an above average *SUS* score with small variation (i.e. acceptance area with the adjective rating “GOOD” [BKMo8]). Further aligning with these results, the *UEQ* for condition **MPS** metrics additionally hold large variations with significant differences to the baseline in the categories *efficiency* and *dependability*.

However, all above mentioned *usability* results differ from the results obtained from execution duration and *TLX* metrics in which **MPS** participants perform as good as the baseline of **Neo4j**. The error rate and keystrokes even show that **MPS** participants perform better than baseline participants by using less insertions to perform the tasks, reduced deletions, and lower average inputs despite the increasing difficulty of tasks. These results allow to conclude two findings with respect to the *usability* of the *EISEQD* used in **MPS**.

First, the results indicate the existence of initial fundamental difficulties for participants, which (most) users overcome during the study. This difficulty shows especially in the significant difference between conditions in the execution duration in task set **S1** and high variances for **S2** in execution duration and *TLX*. Once this obstacle is overcome, participants reach a similar cognitive load in **S4** (except individual outliers). I attribute this effect to the use of fundamentally different projectional editors. Developers are used to traditional text based inputs and the common workflow involves only parser-based programming support. Using a projectional editor for the first time consequently requires the developers to change their mental model and think on the *model* or concept layer. The increased learning curve of projectional editors has been previously discussed in literature [Fow05; Völ+14; PS16]. The learning curve for new *MPS* users is exceptionally steep (especially due to the projectional editing) and joint with the specific domain knowledge of the experiment requires participants to familiarize and understand the tools. However, once

cognitive projec-
tional editor gap

developers adjust and adapt their mental representation to the projectional schema, they overcome the what I call *cognitive projectional editor gap* and reach similar or better performance as before.

Second, *usability* strongly impacts the user acceptance and tool understanding. If users understand and can handle the tooling, they benefit greatly (i.e. having less errors and reduced input) and also score high *SUS* values. Most critical cases of confusion can be traced to unexpected tool behavior, most commonly the deletion of concepts. If a deletion is triggered, the *abstract syntax tree (AST)* node under the cursor is deleted rather than its individual tokens. Since the execution of the study, a new version of *MPS* mitigates this issue by a two stroke mechanic that highlights complete editor cells/concepts that will be deleted if the delete key is pressed again. The introduction to the tools via a single sheet of paper was not sufficient to prepare participants for the usage of a projectional *DSL* editor, supporting the alternative hypothesis. Additionally, not all inputs were automatically transformed by the *IDE* and users had to select variables via the completion menu. This was a technical error in the implementation impacting many users to leave negative feedback, describing the completion as not helpful. More language development and improvement is required in terms of *usability* to increase the acceptance of users with the *EISEQD*.

Put together, the results show that participants consider the baseline condition as more intuitive and hence ERQ_3H_1 is confirmed over the null hypothesis ERQ_3H_0 and the alternative hypothesis ERQ_3H_2 . This is not unexpected as the *Neo4j* condition uses a well-engineered state-of-the-art enterprise database tool and interface. Subjective perception seems to be crucial element for user acceptance and further refinement of my provided tooling as well as more detailed evaluations are necessary. I consider this gap addressable with further tool refinement and an analysis of long-term usage of the generated tools. Additionally, I expect that providing a state-of-the-art graphical interface to represent (sub-graphs will address issues reported in participant feedback and hence increase the *usability* and acceptance strongly.

7.4 SUMMARY

This chapter presents a detailed description of the importance of evaluation of *DSLs*, common evaluation metrics, and my study to evaluate the *EISEQD*. In practice, evaluation of *artifacts* and tools resulting from a *MDSE* process is challenging for the developers and researchers and as a result often omitted completely. My literature review showed that the executed evaluations present only low levels of evidence and are mostly anecdotal reports or descriptions of a proof-of-concept test. The metrics chosen to evaluate *DSLs* and tools are also often weak and deliberately chosen to show the classic *MDSE* benefits

(e.g. *LOC* and code generation). I identify key metrics that can show quantitative benefits of *MDSE* tools when compared to a baseline, such as cognitive load, task execution duration, error rate, and *usability* metrics. Additionally, I present a *between-group* study design to answer my three research questions **ERQ1-ERQ3** regarding effort, effectiveness, and intuitiveness with a high level of evidence. From the obtained study results I conclude that my presented approach does not impact user performance once they overcome the quantitatively identified *cognitive projectional editor gap*. The participants performance in task execution exhibits no significant differences as shown by the *TLX* cognitive load scores and execution duration metric. Contrary, the keystrokes values show that participants require reduced inputs for query design when compared to the baseline. Additionally, *IDE* users show a reduced error rate when creating queries. Thus, users of the *EISEQD* require less effort and are more effective when designing queries and confirming research questions hypotheses **ERQ1H2** and **ERQ2H2**. However, the *usability* of my approach is below the baseline condition as shown by the *UEQ* and *SUS* questionnaire results. This low *usability* scores show that the intuitiveness is higher for **Neo4j** condition, confirming the alternative hypothesis **H1** for research question **ERQ3**. However, at the same time users of the *MPS* condition benefit from all advantages emerging from the applied *MDSE*: Their queries are statically checked at design time, they are provided with auto completion for relevant concepts, they can apply quick fixes, and will (by design) produce no syntactical error. Besides the fact that my study shows the applicability of my approach, I conclude that the various metrics, especially the cognitive load, yield strong evidence for the effectiveness and reduced effort of my approach.

Part VI

PERSPECTIVES

The sixth and last part provides an outlook on further research and summarizes the contribution of this work.

“One never notices what has been done; one can only see what remains to be done.”

—Maria Skłodowska-Curie
letter to her brother in 1894 after
receiving her second graduate degree

Assorted possible future work items became apparent during the course of my research, which lie beyond the scope of this thesis. Before concluding this work, I provide the most promising opportunities and potential future efforts which can be categorized either as work which a) directly improves the implemented system with respect to features and *usability*, or b) investigates high interest areas for follow-up future research endeavors.

In terms of technical improvements, various options are available which can advance the presented *vertical prototype* further towards a fully featured *integrated development environment (IDE)*. The most apparent improvement is an implementation of the dedicated Graph language presented in the implementation-independent architecture. The implementation currently does not include a dedicated graph language, as it was chosen to be closely tied to the Cypher query language (and subsequently to the Cypher *Extended Backus–Naur Form (EBNF)*). A proper separation in the implementation would provide a unification and allow for more diverse further extensions.

Apart from modifications to the languages and the composition, further work can revise features which were not fully completed during the course of this work. These lie especially in the areas of *usability* and quality of life functions for users. The evaluation indicates that the users are missing these features and as a result a noticeable *cognitive projectional editor gap* in metrics such as the *NASA Task Load Index (TLX)*. Even though most users overcome this gap within a short amount of time, the level of entry is high and the learning curve is steep. With further improvements targeting this issue, errors related to projection could be minimized and broken model states became less frequent, thus increasing *usability* and acceptance.

Another example for a high-impact *usability* improvement is the *concrete syntax* of time representation used in the temporal query annotations. Further refinements of this interface are possible to provide users with a unified *concrete syntax*, which supports any temporal expansions description.

The model analysis feature could be extended by implementing additional (sub-)graph and query analysis features. For instance, by adding *boolean satisfiability problem (SAT)* or *satisfiability modulo theories (SMT)* solvers into the *IDE*, one could aid the developers in spotting logic errors within their queries [MB11].

The latter example also provides options for further detailed research on how to optimally make use of the known domain-specific properties in the logical solving process. A *SAT* or *SMT* solver could, for example, check for additional constraints by using information from the *DomainDescription* provided by the *EISE Query Designer (EISEQD)* users via the *Domain Description* language. These constraints could either be provided by the developers themselves or could potentially be directly derived from the provided domain abstraction.

An implementation of the proper separated graph language could additionally allow to implement further alternative query languages to be used in the *EISEQD*. Other query languages can, for example, increase acceptance amongst the users. While Cypher was chosen as the ideal candidate for the domain, the modular language composition allows to integrate different languages with comparably low effort. This, however, has a great impact on the users and the difference will have to be analyzed in the future: Other languages provide different semantics and *model-to-model (M2M)* transformations and it may not be possible to fully express constructs of one *graph query language (GQL)* in another. Further research is required to formally grasp and unify these *GQLs* differences and to aid large-scale data management efforts [SW19; ISO19].

In a more domain focused perspective, future research can investigate extended application of the proposed system and tools in the *Embodied Interaction in Smart Environments (EISE)* domain. For example, the temporal query feature could be expanded to match graph patterns materializing in the future. Such a query would thus transform into a *standing query* or a *continuous query* which consequently provides a streaming data interface. Necessary steps include an adaptation of existing languages such as the *Time* language and the design of developer interfaces to access the resulting streaming data within their software components. Such an application can possibly be even made available to the final environment users, who can use the standing queries to design environment controlling rules or agents. Research can investigate to what extend and how an end-user directed *GQL* interface can overcome common problems of this domain, such as the significant discrepancy between user interpretation and reality of a rule [HC15]. A future evaluation can analyze how this approach compares to available approaches which use rule based *domain-specific languages (DSLs)* to create triggers and commonly reduce errors via static analysis [NE16].

Lastly, further evaluation and study based analysis can be executed to validate the approach and tooling presented in this thesis in greater detail. While the presented evaluation already allowed to introspect the efforts with a high level of evidence, an even more detailed longitudinal study will be instrumental [KBM16; GGA10]. Therefore multiple measurements and metrics of study participants need to be recorded over longer time spans. Following a use case or group of use cases over a long period of time can thus allow to gather normative data regarding *usability* or tooling benefits to users.

CONCLUSION

“For a research worker the unforgotten moments of his life are those rare ones which come after years of plodding work, when the veil over nature’s secret seems suddenly to lift & when what was dark & chaotic appears in a clear & beautiful light & pattern.”

—Gerty Cori

Nobel Prize winner in Physiology or Medicine
for her work in metabolizing carbohydrates

In this thesis I applied *Model-driven Software Engineering (MDSE)* techniques to support the design of graph queries targeting interaction relevant knowledge. I therefore considered the perspective of the role of *behavior developers* in the *Embodied Interaction in Smart Environments (EISE)* domain who create complex *human–robot interaction (HRI)* system activities. In total, I attended to five research questions in this thesis and carried out an empirical user evaluation investigating the applicability of the approach.

The contributions of this thesis with respect to the research questions **RQ1-RQ5** are as follows. The volatile and dynamic nature of interactive scenarios in research environments demands that domain modeling and abstraction (**RQ1**) lies in the user model space; *M1* of the *Meta-Object Facility (MOF)* layer. The concepts and their relations, which in sum abstract the domain, evolve frequently and thus need to be easily changeable without continuous language evolution. Orthogonal domains, such as temporal representations, however require the abstraction as *meta-models* in the *M2* layer. Constraints on queries can then be realized as orthogonal annotations to the query model *abstract syntax tree (AST)* with no impact on the query *model*. Further, the proposed development process (**RQ2**) applied in this thesis emphasizes the increased need for subsequent application and evaluation of the developed *domain-specific languages (DSLs)*, as presented in [Section 3.3](#). The objectives and requirements for my approach (**RQ2**) were identified in [Chapter 5](#). It showed, that temporal querying and query analysis at design time pose fundamental tasks in this context. The requirements lay the foundation for the following proposed implementation-independent language composition (**RQ3**). The *DSL* composition is explicitly chosen to provide an extensible query language and further supporting languages, which extensively aid *behavior developers* in the *graph database query (GDQ)* design process. I further describe each individual language in detail alongside corresponding *meta-models* and clarify the intended behavior of the languages by providing

detailed *denotational semantics*. An implementation of these theoretical considerations (RQ4) is presented in Chapter 6. My realization uses the *language workbench JetBrains Meta Programming System (MPS)* to create a fully *integrated development environment (IDE)*, the *EISE Query Designer (EISEQD)*. This *vertical prototype* implements a functioning slice of the proposed system and their behavior, hence providing a set of languages which allow for:

- a) Graph query design,
- b) *M1* domain modeling and *model* grounded querying design,
- c) (Relative) temporal graph querying,
- d) Query design support, query checking, and design time query analysis,
- e) Integration in an existing application system, and
- f) Additional domain-specific features, such as graph visualization or direct query execution.

To validate the approach, I conducted an empirical evaluation of the created tool (RQ5), which analyzes the implementation in comparison to state-of-the-art tooling within a baseline condition. The performed evaluation analyzes multiple metrics and provides a high level of evidence. In terms of *usability*, my implementation does not reach the professional tooling of the baseline condition. Consequently, this confirms hypotheses H₁ of evaluation research question ERQ3: Users perceive the implemented tooling as less *usable* when designing *GDQ*. However, the users recognized the *novelty* and potential of the approach and rated this factor above average. Lastly, it showed that users of the *EISEQD* perform similar to the baseline condition in terms of cognitive load and task execution duration, while (at the same time) exhibiting a significantly reduced input and error rate. Thus, the hypotheses H₂ of evaluation research question ERQ1 and ERQ2 were confirmed respectively over their alternatives: Users of the *EISEQD* require less *effort* and are more *effective* when designing *GDQ* queries.

Part VII
APPENDIX

EVALUATION APPENDIX CHAPTER

A.1 EVALUATION QUESTIONNAIRE

• Please choose for each of the following statements one element that best describes your reactions to the used tool. Choose on a scale from "strongly disagree" to "strongly agree".

	strongly disagree		strongly agree	
I think that I would like to use this system frequently	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I found the system unnecessarily complex	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I thought the system was easy to use	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I think that I would need the support of a technical person to be able to use this system	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I found the various functions in this system were well integrated	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I thought there was too much inconsistency in this system	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I would imagine that most people would learn to use this system very quickly	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I found the system very cumbersome to use	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I felt very confident using the system	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I needed to learn a lot of things before I could get going with this system	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Figure A.1: The *System Usability Scale (SUS)* questionnaire as presented to participants within the *EISE Query Designer (EISEQD)* study.

• Select the point on the scale that best reflects your experience with respect to previous task.

Please click and drag the slider handles to enter your answer.

Mental Demand How mentally demanding was the task?	very low	<input type="range"/>	very high
Physical Demand How physically demanding was the task?	very low	<input type="range"/>	very high
Temporal Demand How hurried or rushed was the pace of the task?	very low	<input type="range"/>	very high
Performance How successful were you in accomplishing what you were asked to do?	very low	<input type="range"/>	very high
Effort How hard did you have to work to accomplish your level of performance?	very low	<input type="range"/>	very high
Frustration How insecure, discouraged, irritated, stressed, and annoyed were you?	very low	<input type="range"/>	very high

Figure A.2: The *NASA Task Load Index (TLX)* questionnaire as presented to participants after each task execution within the *EISEQD* study.

• Please fill in your position about the following pairs of contrasting attributes that may apply to the program. The circles between the attributes represent gradations between the opposites. You can express your agreement with the attributes by selecting the circle that most closely reflects your impression.

annoying	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	enjoyable
not understandable	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	understandable
creative	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	dull
easy to learn	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	difficult to learn
valuable	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	inferior
boring	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	exciting
not interesting	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	interesting
unpredictable	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	predictable
fast	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	slow
inventive	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	conventional
obstructive	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	supportive
good	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	bad
complicated	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	easy
unlikable	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	pleasing
usual	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	leading edge
unpleasant	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	pleasant
secure	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	not secure
motivating	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	demotivating
meets expectations	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	does not meet expectations
inefficient	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	efficient
clear	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	confusing
impractical	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	practical
organized	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	cluttered
attractive	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	unattractive
friendly	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	unfriendly
conservative	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	innovative

Figure A.3: The *User Experience Questionnaire (UEQ)* questionnaire as presented to participants within the *EISEQD* study.

A.2 STUDY INFORMATION MATERIAL

This section holds all information and task related documents handed to participants during the *EISEQD* study.

A.2.1 *Cypher information material*

Cypher Information Sheet

This document serves you as a help and small summary of the most important principles on how to write queries using Cypher.

The language has many more features beyond the ones mentioned here. However, these will not be relevant for any of the tasks presented to you in this study as all tasks can be done solely with this provided information.

1. Cypher: The declarative query language for the graph database Neo4j

Neo4j is a graph database, adopting a labeled property graph model. In Neo4j terminology, vertices are called nodes, and edges are called relationships.

Cypher is a declarative graph query language that allows for expressive and efficient querying and updating of the graph store. It is inspired by a number of different approaches and builds upon established practices for expressive querying. Cypher borrows its general structure from SQL — queries are built up using various clauses. Clauses are chained together, and they feed intermediate result sets between each other. For example, the matching identifiers from one clause will be the context that the next clause exists in.

1.1 Key principles and capabilities of Cypher

- Cypher matches patterns of nodes and relationships in the graph to extract information.
- Cypher has the concept of identifiers which denote named, bound elements and parameters.
- Cypher manages constraints on patterns.

1.2 Basic Read-Query-Structure

Patterns are the fundamental traversal description of Cypher. Designed after ASCII art representing **nodes as circles** and **relationships as arrows**, such as

```
(identifier1)-->(identifier2)
```

Relationship identifiers are specified within square brackets, with an optional type after a colon, like

```
(u)-[r:HAS_ACCESS]->(a)
```

Labels are specified similarly to relationship types, following a colon:

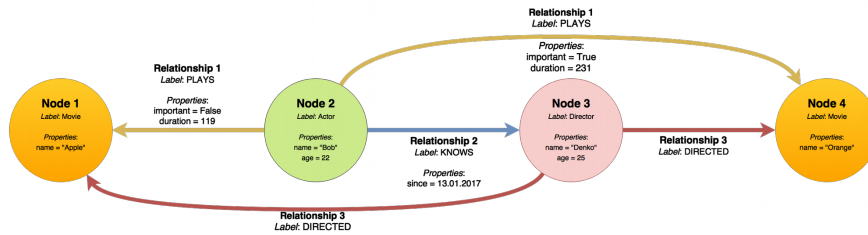
```
(u:User)-->(a:Asset)
```

1.3 Most important query keywords

- **MATCH**: The graph pattern to match. This is the most common way to get data from the graph.
- **WHERE**: Adds constraints to a pattern, or filters the intermediate result.
- **RETURN**: What to return.

2. Examples

Consider this simple example graph:



The most basic query is a match on all nodes and then return them:

```

MATCH (a)
RETURN a
  
```

Optionally, we can add a **LIMIT** to the **RETURN** clause to limit results (for large databases):

```

MATCH (a)
RETURN a LIMIT 50
  
```

Patterns in the graph can be matched using a **MATCH** clause and the results are returned with a **RETURN** clause. The following query will search for nodes *a* and *b* with relationships pointing to each other and return all matched nodes (this matching will ignore the direction of the relationships - to match directions one has to use "-->" or "<--" respectively instead of the undirected "--").

```

MATCH (a)--(b)
RETURN a, b
  
```

One can match multiple sub-graphs at the same time:

```

MATCH (a)--(b), (b)--(c)
RETURN a, b
  
```

Labels can be used to further specify the pattern. The following will return all nodes with the label *Movie* that have a relationship with a node labeled *Actor*:

```

MATCH (a:Actor)--(b:Movie)
RETURN b
  
```

The **MATCH** clause can be further filtered using a **WHERE** clause. In the following example we use the **identifier** *b* and restrict the matching to all nodes which have a string unequal to "Orange" in the **property** called *name* (note: instead of "!=" Cypher uses the "<>" as the unequal operator) :

```

MATCH (a:Actor)--(b:Movie)
WHERE b.name <> "Orange"
RETURN b
  
```

However, it is also possible to define **node properties** in the **MATCH** clause itself (note: it is impossible to do negative matches here). The following query returns the same result as the one above:

```
MATCH (a:Actor)--(b:Movie {name : "Apple"})
RETURN a
```

The idea of **properties** and **labels** is also applicable to relationships. The following defines the relationship between the nodes by matching relationships *r* with the label *PLAYS* and further restrictions on its properties in the **WHERE** clause:

```
MATCH (a:Actor)-[r:PLAYS {important: True}]- (b:Movie)
WHERE r.duration=231 AND a.name = "Denko"
RETURN b
```

Lastly, there are also functions available in Cypher. They allow to influence the results. In this example we simply return the count of the nodes that our query returned:

```
MATCH (a:Actor)-[r:PLAYS {important: True}]- (b:Movie)
WHERE r.duration=231 AND a.name = "Denko"
RETURN COUNT(b)
```

Here is a larger example also using multiple matches to also find the according directors:

```
MATCH (a:Actor)-[r:PLAYS {important: True}]- (b:Movie), (b:Movie)-[:DIRECTED]- (c:Director)
WHERE r.duration=231 AND a.name = "Denko"
RETURN COUNT(b), c
```

3.1 Read Query Structure

MATCH {PATTERN}
WHERE {BOOLEAN_EXPRESSION}
RETURN {RESULT_SET} [LIMIT X]

3.2 MATCH Syntax

Syntax Example	Explanation
MATCH (n:Actor)-[:KNOWS]->(m:Director) WHERE n.name = "Alice"	Nodes and Relationships ins MATCH patterns can contain labels and properties
MATCH (n)-->(m)<--(k)--(o)	Any pattern can be used in MATCH
MATCH (n {name: "Alice"})-->(m)	Pattern that also matches a node property
MATCH (n {name: "Alice", age: 33})-->(m)	Pattern that matches multiple node properties
MATCH (a)-->(b)<--(c), (b)--(o)	Multiple patterns and references to each other are allowed in a single MATCH

3.3 WHERE Syntax

Syntax Example	Explanation
WHERE n.property <> {value}	Use a predicate to filter (<> is the unequal operator in Cypher - it is the same as <i>WHERE NOT (n.property = {value})</i>)
WHERE n.property1 <> {value2} AND NOT (n.property2 = {value2})	Concatenate multiple filters; second filter is negated (see Boolean operators)

3.4 Operators

Type	Operator
Mathematical	+, -, *, /, %, ^
Comparison	=, <>, <, >, <=, >=
Boolean	AND, OR, XOR, NOT
String	+
Regular Expression	=~

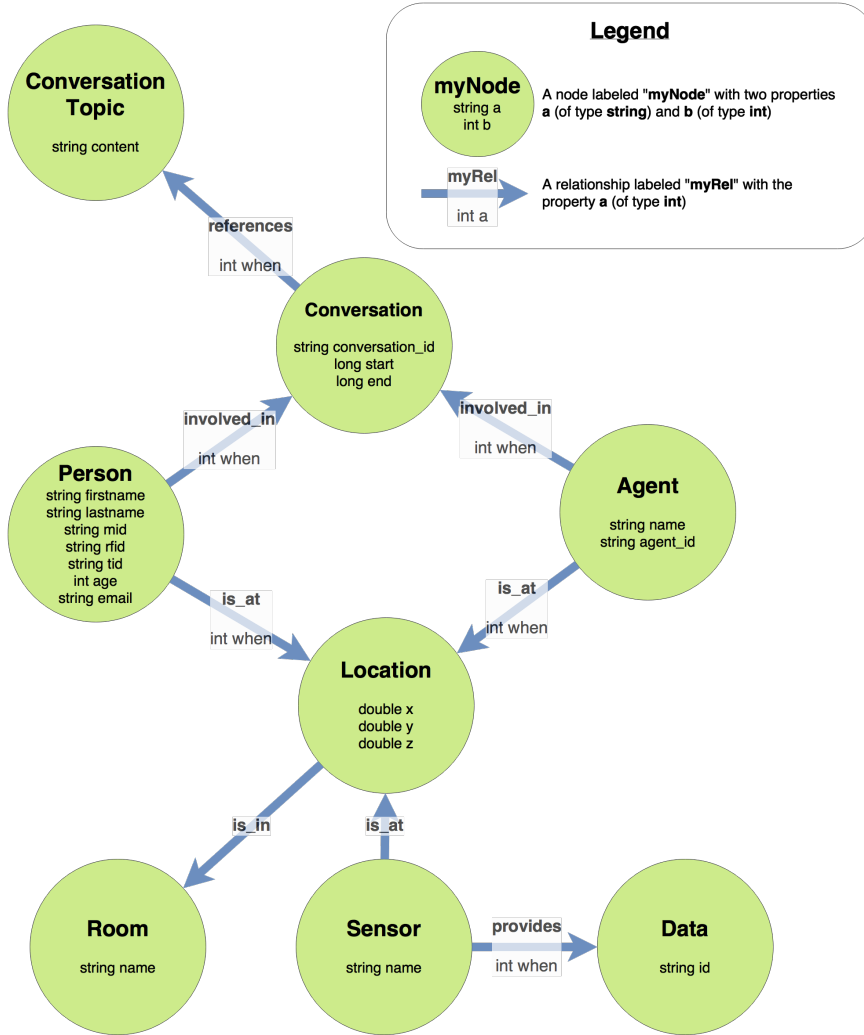
3.5 RETURN Functions

Function	Description
RETURN count (identifier)	The number of non- NULL values (aggregation)

A.2.2 EISE Domain information material

Embodied Interaction in Smart Environments Domain

All queries are to be done within in the “Embodied Interaction in Smart Environments” Domain (EISE Domain). The following image shows you the data meta model of the graph database. Instances of these types of nodes and relationships are stored within the database. Instances that are created which do not provide full details (for example a Person missing the *firstname* property) are **initialised with empty default values**. According defaults are: “” for strings and 0 for int/long/double.



A.2.3 Tool information material

A.2.3.1 Neo4j tool information material

Tool Information Sheet

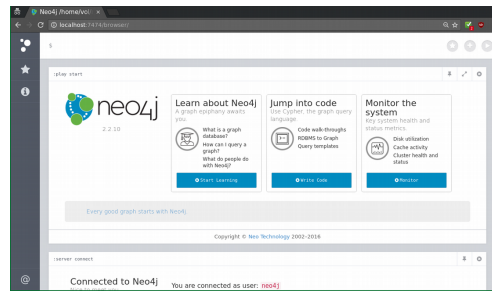
This document serves you as a help and summary of the most important principles of the “Neo4j Web-Interface”. It is the default tool provided by Neo4j and you will use it during this study to write plain Cypher queries which are presented to you as textual questions.

1. Introduction

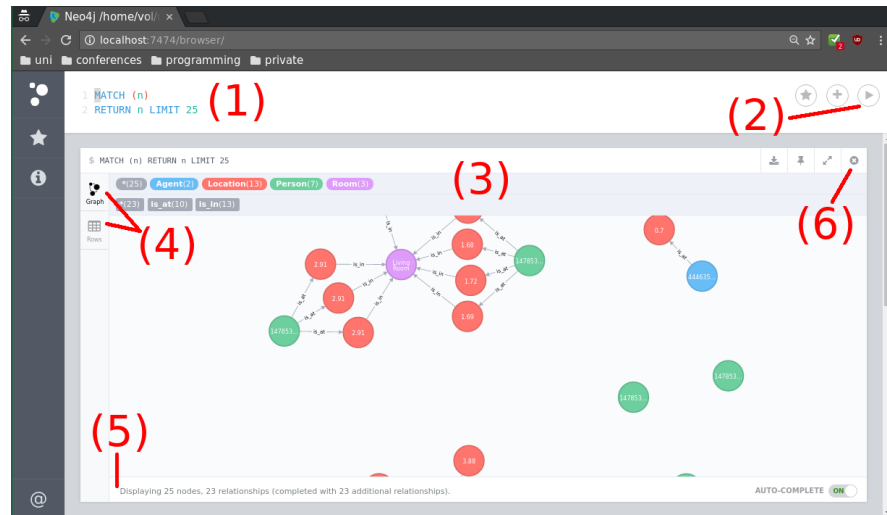
You can reach the web interface via the browser by browsing to <http://localhost:7474/browser/>. In case you are required to provide credentials, please use the following.

User: neo4j | Password: a

You will be greeted by the default interface:



To write and execute a query (see the following picture) you have use the input field (1) and press the play button (2). The result will appear below with a visualised graph (3). You can switch the view from graph to row on the left side of the visualisation (4). Further details about currently displayed results are listed at the bottom (5). You can remove old queries by pressing (6). If you write large queries it may be helpful to limit the results via LIMIT 50 as otherwise large amounts will be displayed which might slow down the browser significantly. Once the query is finished you can remove this restriction to get the final result.



2. Task Workflow

When solving the tasks in the study you can follow this order:

1. Use the web-interface to design and execute queries
2. Evaluate the results and refine query
3. Once you think you finished a query: Copy the query and paste it in the prepared opened text editor
4. Solve the next task

In case you close the browser or text editor by accident, you can find the according links on the desktop to re-open them.

3. Shortcuts

Shift+Enter	Create a new line in query input	Allows to write multi-line queries. Makes the design of long queries easier and gives you more overview
Ctrl+Enter	Execute query	Executes multi-line queries
Up/Down	Previous/Next query	Allows to switch to and re-use the previous queries
Left click on previous query	Put query into input field	Allows you to modify the previously executed query and execute it again.

A.2.3.2 MPS tool information material

Tool Information Sheet

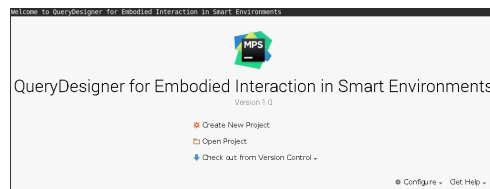
This document serves you as a help and summary of the most important principles of the “Embodied Interaction in Smart Environments Query-Designer” (EISE Query-Designer). Instead of writing plain Cypher queries, you will use the **EISE Query-Designer** which is a tool provided to ease the creation and execution of Cypher queries. It extends the Cypher language with several features such as code completion, syntax checking or direct in-tool execution.

The tool has many more features beyond the ones mentioned here. However, these will not be relevant for any of the tasks presented to you in this study as all tasks can be done solely with this provided information.

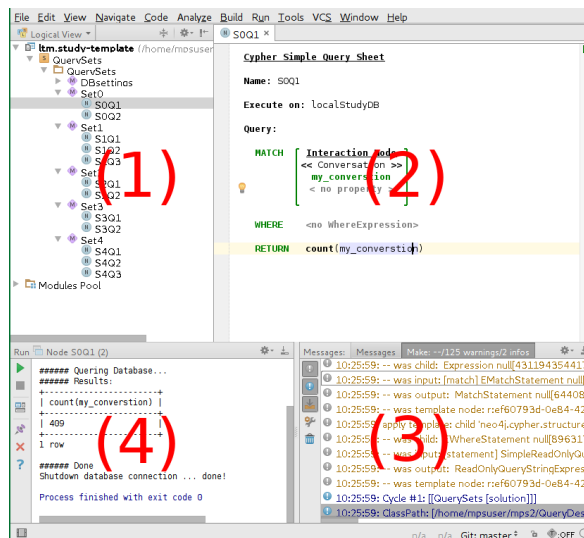
In case you close the tool by accident, you can find the according link on the desktop to start it again. The prepared project to load is also located on the desktop.

1. The EISE Query-Designer: Design and execute domain specific queries

The EISE Query-Designer is an advanced IDE created using **Jetbrains MPS**.

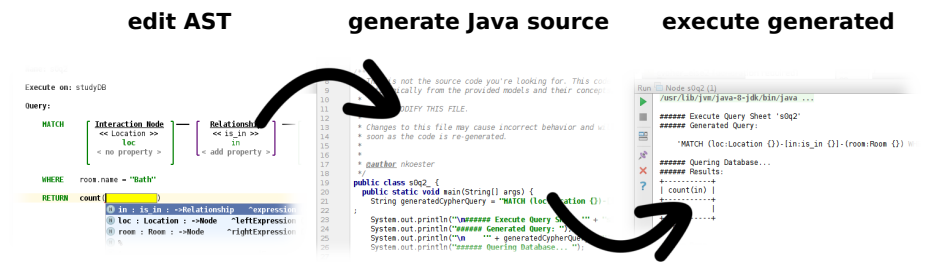


The Query-Designer is just like any other IDE (Eclipse, IntelliJ, Netbeans, etc.). It has a project view (1), an editing area (2), a compile output log (3), and an execution output area (4).



In contrast to other IDEs you do not directly edit the text, instead one edits the abstract syntax tree (AST) of the languages supported by the tool. The Query-Designer is familiar with the Cypher language and can give you appropriate support. So called Query-Sheets already give you the general shape for Cypher queries (MATCH... WHERE... RETURN...) which you will have to fill according to the tasks. Further, it also provides you with auto-completion wherever possible in the editor (to open the auto-completion press **Ctrl+Space**). Once a query is designed, you need to generate the according Java source files and run them afterwards (the connection to the database is done automatically in the background).

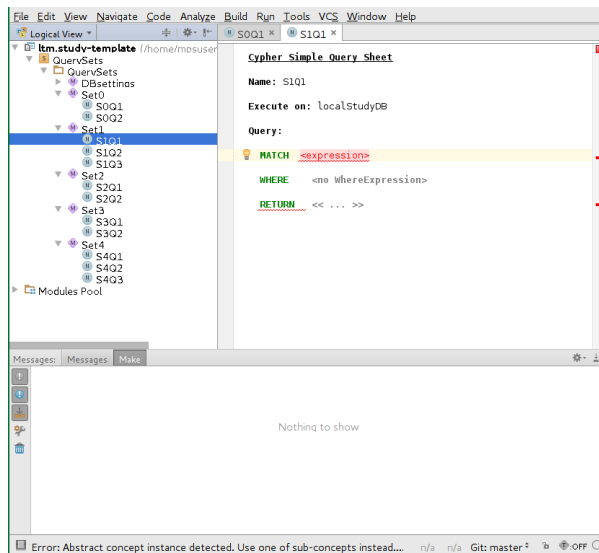
The Query-Designer workflow is as follows:



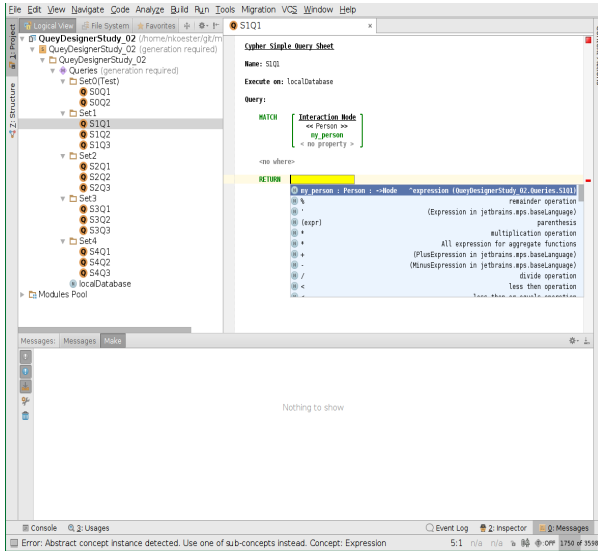
2. Task Workflow

When solving the given tasks in the study you will only use the EISE Query-Designer to design and execute your queries. Once you think you finished a query you save the project and go on with the tasks.

You are provided a project with prepared Query-Sheets in which you can solve each given task (S1Q1, S1Q2, etc.) individually. The general workflow is:

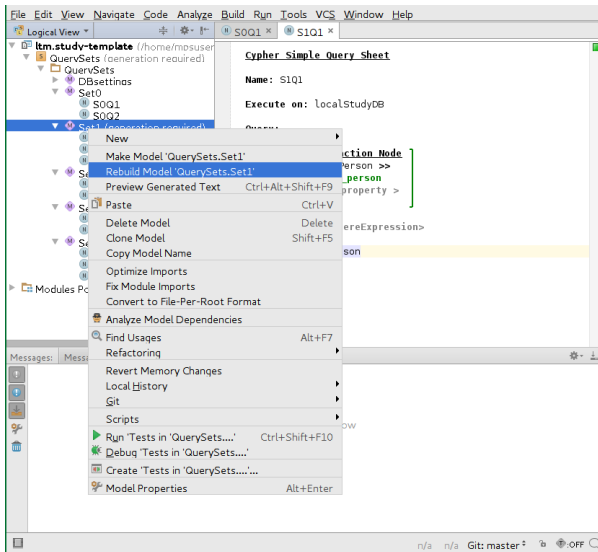


Step 1: Open the according Query-Sheet for the current task.

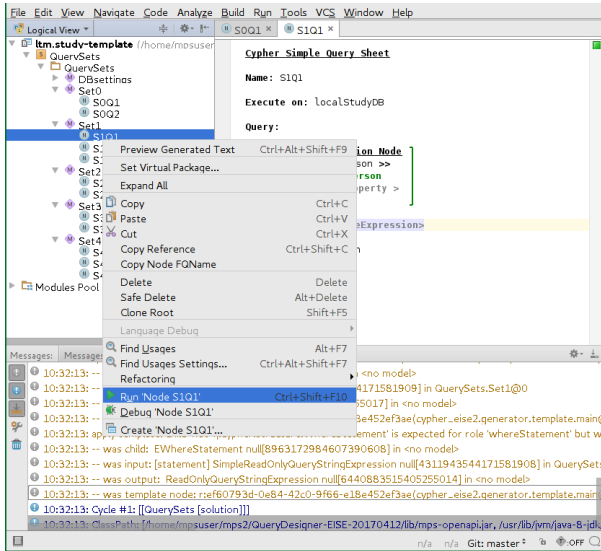


Step 2: Write a query using the provided tool support such as auto-completion (CTRL+Space) and syntax checking.

Errors are highlighted in red and give verbose feedback on mouse over. **Only compile once all errors have been cleared. Compiling with errors will lead to faulty queries.**

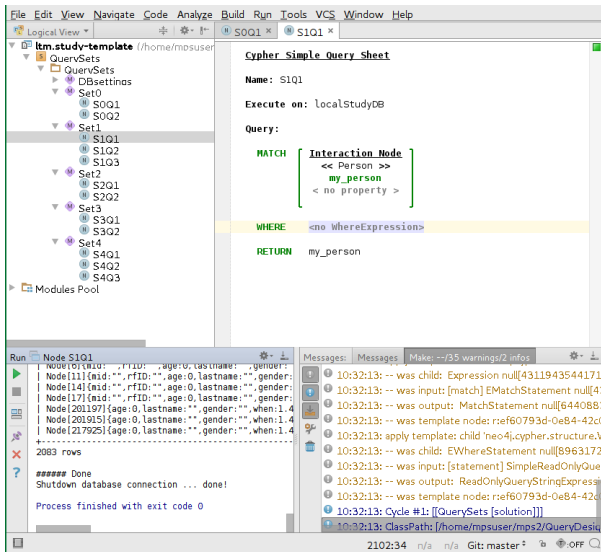


Step 3: Generate the according code via the context menu of the model.



Step 4: Run the generated code via the context menu of the Query-Sheet or shortcut.

(Note: **Always re-compile before you run** - otherwise previously generated code is executed! You can **only run a query if it has been generated before**)



Step 5: Inspect and verify the results (remember to save the project).

3. Shortcuts

Enter or Shift + Enter (in the editor)	Add a node in the AST	Will add an appropriate node at the current location of the cursor if possible. For example: this will add another MATCH clause or RETURN clause if pressed at the end of the according clause.
Ctrl+Space	Invoke code completion	Basic code completion helps you complete the names of nodes, relationships, and keywords within the visibility scope. When you invoke code completion, the context is analysed and choices that are reachable from the current position of the cursor are suggested.
Ctrl + Shift + F9 (in the editor)	Generate the model of the currently open query	Will generate the module in which the opened query resides.
Ctrl + Shift + F10 (in the editor)	Run the currently open query	Will run the opened query against the database. The results of the query will be printed to the console.
Tab / Shift + Tab	Go to next/previous editable cell	Will position the cursor at the next/previous cell you can edit
-- (in MATCH clause)	Add a relationship	Manually add a relationship to the Node where the cursor is. Simply adding two dashes will automatically add a relationship.
AND / OR / = / <> / etc. (in WHERE clause)	Add a logical phrase	Writing and/or/=/<>/etc. followed by a space in the where clause automatically adds the appropriate logical expression.
Ctrl + w / Ctrl + Shift + w	Increase/decrease selection in the editor	Will increase/decrease the selection based on the abstract syntax tree. Alternatively one can also use Shift+Up/Shift+Down.
Alt + 1	Show/Hide project view	Shows or hides the view on the project on the left side of the IDE.

A.2.4 *Task material***Test Set (S0)****Query 1 (S0Q1)** [1 minute]

How many conversations are stored in the database in total?

Hint: This query is already prepared, execute the provided query.

Expected result: 409

Query 2 (S0Q2) [1 minute]

How many locations are located in the Room named "Bath"?

Hint: This refers to the amount of *Locations* that have an *is_in* relationship to the *Room* with name "Bath".

This query is already prepared, execute the provided query.

Expected result: 1414

Set 1 (S1)

Query 1 (S1Q1) [2 minutes]

Which persons are stored in the database in total?

Hint: Refers to the Person instances in the database.

Expected result: 2083 instances of persons

Query 2 (S1Q2) [2 minutes]

Which persons were stored in the database with a first name?

Hint: All persons have a *firstname* property. If not provided an empty string ("") is stored. This query asks for all persons that do not have an empty *firstname* property.

Expected result: 12 instances of persons

Query 3 (S1Q3) [2 minutes]

How many agents are stored in the database in total?

Hint: Refers to the absolute count of all agents in the database. Use the *count()* function.

Expected result: 2

Set 2 (S2)**Query 1 (S2Q1)** [5 minutes]

In which conversations where persons with a known last name involved?

Hint: Refers to all instances of conversations that have an *involved_in* relationship to persons who have a not empty *lastname* property.

Expected result: 214 instances of conversations (or 219 rows)

Query 2 (S2Q2) [5 minutes]

How many conversations are in the database in which persons and agents were active together?

Hint: 1. Refers to the amount of conversations to which persons had an *involved_in* relationship and at the same time agents also have an *involved_in* relationship to.

2. Use multiple relationships within a MATCH clause (alternatively it is also possible to use multiple MATCH clauses).

Expected result: 243

Set 3 (S3)**Query 1 (S3Q1)** [5 minutes]

In which room and when did conversations start in which persons interacted with the agent named "Flobi"?

Hint: Multiple MATCH clauses allow to match against several graphs at the same time. The elements in MATCH clauses can reference to each other (see Cypher Information Sheet, Section 2).

Expected result: Total of 243 (rooms are among „Wardrobe“ and „Kitchen“)

Query 2 (S3Q2) [5 minutes]

Which conversation where the agent with name "Flobi" was involved in have happened last year and which persons were also involved?



Hint: To filter the last year one has to make a restriction on the *start* and *end* property of conversations. The filter has to be done after the timestamp 1451606400 and before timestamp 1481818044.

Expected result: 211 instances of conversations (or 216 rows) and a total of 11 instances of persons

Set 4 (S4)**Query 1 (S4Q1)** [2 minutes]**Which conversations were recorded?****Hint:** Refers to all conversations in the database.**Expected result:** 409 instances of conversations**Query 2 (S4Q2)** [2 minutes]**Which persons age is known?****Hint:** All persons have an age property. If not provided a zero ("0") is stored. This query asks for all persons that have an age property larger than zero.**Expected result:** 12 instances of persons**Query 3 (S4Q3):****How many sensors are stored in the database?****Hint:** Refers to the absolute count of all sensors in the database. Use the *count()* function.**Expected result:** 2

A.3 ETHICS DOCUMENTS

A.4 ETHICS COMMITTEE APPLICATION

 Universität Bielefeld Ethik-Kommission			
<p>Ethik-Kommission der Universität Bielefeld Postfach 10 01 31 D-33501 Bielefeld</p>	<p>Der Vorsitzende</p> <p>Geschäftsstelle: Fatma Akkaya-Willis Raum: T5-239 Tel.: 0521 106-4436 ethikkommission@uni-bielefeld.de Az.: 1266</p> <p>Bielefeld, 13.03.2017 Seite 1 von 1</p>		
<p>Stellungnahme der Ethik-Kommission der Universität Bielefeld zu Antrag Nr. 2017 - 058 vom 13.03.2017</p> <p>Kurzbezeichnung der Studie: "Nutzbarkeit einer DSL zur Beschreibung von Graphdatenbank Anfragen"</p> <p>Hauptansprechpartnerin: Norman Köster Betreuer: Philipp Cimaiano</p> <p>Die Ethik-Kommission der Universität Bielefeld hat den Antrag nach den ethischen Richtlinien der Deutschen Gesellschaft für Psychologie e.V. und des Berufsverbands Deutscher Psychologinnen und Psychologen e.V. begutachtet.</p> <p>Auf der Grundlage der eingereichten Unterlagen hält die Ethik-Kommission der Universität Bielefeld die Durchführung der Studie in der beschriebenen Form für ethisch unbedenklich.</p> <p>Für die Ethik-Kommission</p> <p> Prof. Dr. Gerd Bohner Vorsitzender</p>			
<p>Universität Bielefeld Universitätsstraße 25 33615 Bielefeld</p>	<p>Öffentliche Verkehrsmittel: Stadtbahnlinie 4 Richtung Lohmannshof</p>	<p>Bankverbindung: Landesbank Hessen-Thüringen BLZ: 300 500 00, Konto: 61 036 IBAN: DE 46 3005 0000 0000 061036 SWIFT-BIC: WELADED</p>	<p>Steuernummer: 30515879/0433 USt-IdNr.: DE811307718 Finanzamt Bielefeld Innenstadt → www.uni-bielefeld.de</p>

A.5 CONSENT FORM



Einverständniserklärung

Graph-Query-Design Studie 2017

Hiermit erklären Sie sich bereit, an der Graph-Query-Design Studie teilzunehmen.

Ziel der Studie ist die Dokumentation und wissenschaftliche Untersuchung der Nutzbarkeit von Programmen zur Unterstützung beim Design und Ausführen von Anfragen an Graph-Datenbanken. Aus den Versuchen sollen Erkenntnisse über Anforderungen an Entwicklung und das Design gezogen werden. Diese Studie findet im Rahmen des Projekt „The Cognitive Service Robotics Apartment as Ambient Host (CSRA, LSP-01)“ statt.

Das Experiment wird ca. 60 Minuten dauern. Ihre Daten (Bildschirmaufzeichnung, Tastatureingaben und Daten aus Fragebögen) werden streng vertraulich behandelt, anonymisiert ausgewertet und nicht an Dritte weitergegeben. Eine Zuordnung ihrer personenbezogenen Daten wird nach Abschluss der Studie gelöscht. Sie erklären sich damit einverstanden, dass ihre studienbezogenen Daten aufgezeichnet und anonymisiert für wissenschaftliche Auswertungen verwendet werden. Einer möglichen Veröffentlichung der anonymisierten Daten dieser Studie stimmen Sie mit ihrer Teilnahme zu.

Ihre Teilnahme an der Untersuchung ist freiwillig. Bitte beachten Sie, dass es Ihnen jederzeit frei steht, Ihr Einverständnis zurückzuziehen und die Untersuchung abzubrechen. Daraus werden Ihnen keine Nachteile entstehen.

Sollten Sie sich mit den oben geschilderten Bedingungen einverstanden erklären, so unterschreiben sie bitte wie folgt:

Name	Datum	Unterschrift
------	-------	--------------

Bitte kreuzen Sie außerdem an, in welchem Umfang Sie die Veröffentlichung des Videomaterials gestatten:

- keine Veröffentlichung
- nur im Rahmen von wissenschaftlichen Vorträgen, z.B. anonymisiert auf Konferenzen
- uneingeschränkte Nutzung, z.B. auch im Citec-YouTube-Channel
- Ich möchte über die Nutzung von Fall zu Fall und unter Vorbehalt der Sichtung des Materials entscheiden (Bitte dazu ihre E-Mail Adresse angeben).

Dürfen wir sie ggf. bei Fragen zu einem späterem Zeitpunkt noch einmal kontaktieren? Falls ja, bitte auf diese Weise:

A.6 QUESTIONNAIRE RESULTS

A.6.1 *Feedback*A.6.1.1 *Tool feedback*

Condition: MPS, ID: 7 – “mps immernoch furchtbar”

Condition: MPS, ID: 10 – “Man muss die Autovervollständigung nutzen, damit eingegebene Worte erkannt werden. Einfach nur das Wort einzugeben führt zu einem Fehler. Wenn man gewohnt ist, ohne Autovervollständigung zu arbeiten, passiert das schnell. Und es ist verwirrend, weil das richtige dasteht, aber trotzdem ein Fehler angezeigt wird. Genauso muss man mit Integer Eingaben immer recht lange warten, bis es tatsächlich übernommen wird. InteractionNode ist falsch geschrieben in der Autovervollständigung (Intr...). Wenn man "Inter" eingibt und dann enter drückt, kommt daher kein Vorschlag.”

Condition: MPS, ID: 12 – “Autocomplete manchmal etwas komisch in der benutzung”

Condition: NEO, ID: 15 – “Es ist nicht immer klar, wie die Beziehungen benannt sind, wenn man die Graph Visualisierung benutzt”

Condition: NEO, ID: 17 – “MATCH und RETURN scheinen immer gebraucht zu werden für Anfragen, daher könnten diese als feste Felder bereitgestellt werden, sodass sie nicht als Keywords eingetippt werden müssen”

Condition: MPS, ID: 18 – “Mit Vorerfahrung in MPS richten sich einige meiner Antworten vielleicht auch auf die Benutzbarkeit der IDE an sich.”

Condition: MPS, ID: 20 – “Lediglich die IDE hat so einige Macken. Total nervig: Das man nochmal Cntl-Space drücken muss obwohl bereits das richtige keyword geschrieben war. Auch das man nicht direkt Durchschreiben konnte und immer wieder alles mit Cntl-Space bestätigen müsste war nervig. (Autovervollständigung ist sehr hilfreich wenn man die Domain nicht kennt, wenn bekannt dann sollte eine manuelle Eingabe immer erlaubt sein weil die Assistenz sonst eher störend wirkt. Das Locations eine Liste an Koordinaten sind ist nicht intuitiv.”

Condition: NEO, ID: 23 – “Interaktive Hilfe / Tutorial oder mehr komplexere Beispiele in der Dokumentation.”

Condition: MPS, ID: 24 – “Möglicherweise bei der Darstellung auch Context Menüs über die Maus hilfreich. Variablen eingeben sollte auch akzeptiert werden, ohne Autocomplete zu verwenden.”

Condition: MPS, ID: 26 – “Programm akzeptiert nur Ausdrücke, die mit code-completion erstellt wurden.”

Condition: MPS, ID: 28 – “Autovervollständigung ist meistens nervig und unnötig”

Condition: NEO, ID: 29 – “Könnte man in den eckigen Klammern auch Vergleichsoperatoren nutzen, statt das WHERE?”

Condition: MPS, ID: 32 – “Die Schritte Übersetzen und Ausführen müssen vereinheitlicht werden”

Condition: MPS, ID: 34 – “Unvorhersehbar delete sprengt unrelated entries weg. Autocomplete triggern nervt insgesamt behindert mmn mehr als dsl im texteditor zu schreiben”

A.6.1.2 *Study feedback*

Condition: MPS, ID: 10 – “War das Wort InteractionNode irgendwo eingeführt worden? Die Bezeichnung hat mich anfangs verwirrt, aber vielleicht hab ich es überlesen.”

Condition: NEO, ID: 15 – “Prozentabstufungen (100er) finde ich etwas ueber... 5er Schritte reichen voellig”

Condition: NEO, ID: 17 – ““LIMIT 50” habe ich im Rahmen der Studie nie verwenden müssen und erkenne keinen Grund, warum in den Instruktionen darauf hingewiesen wird, dies sei empfehlenswert.

Sämtliche Anfragen wurden fast auf Knopfdruck verarbeitet. Beim SUS könnte “Menschen” u.U. durch “Informatiker” o.ä. ersetzt werden, zumal “normale Menschen” sicher zunächst mehr lernen bzw. Zeit investieren müssten, um Cypher zu lernen, als im IT-Bereich vorgebildete Leute. Der Eintrag “Keiner” bei der Frage nach Erfahrung mit Programmiersprachen hat sich mir nicht erschlossen. Die Frage “War ihnen die Aufgabe bereits bekannt?” ist mit dieser Skala schwer zu beantworten, zumal “die Aufgabe” nicht klar spezifiziert ist. Wenn die konkreten Testaufgaben gemeint sind, reicht eine binäre Antwortmöglichkeit (ja/nein).”

Condition: MPS, ID: 24 – “Der Schwierigkeitsgrad nimmt von Set 2 zu Set 3 extrem zu. Dies führt zu Frustrationen insbesondere bei dem engen Zeitrahmen.”

Condition: NEO, ID: 25 – “Es war nicht hundertprozentig klar, ob die Zeitvorgabe bei den Aufgaben massgeblich war oder die komplett richtige Lösung.”

Condition: MPS, ID: 28 – “S3 war bockschwer, links- und rechts-Ausdrücke sollte man im Bedienkonzept (Anleitung) besser erklären”

Condition: NEO, ID: 29 – “Beim CYPHER sheet wäre noch statt nur die Queries anzuzeigen auch interessant die tatsächlichen Ergebnisse zu sehen. Das ist dann weniger verwirrend.”

Condition: NEO, ID: 31 – “Das man zwei Zettel hat die man für das konstruieren der Anfragen hat ist nicht ganz optimal, aber lässt sich wohl nicht vermeiden. Ich hätte mir bessere Beispiele zu kombinierten Anfragen, Filtern und vor allem für die Relationen gewünscht, hauptsächlich für große Relationen, die in SET 3 auftauchten.”

Condition: MPS, ID: 32 – “Textuelle Beschreibung nicht konsistent mit der IDE”

Condition: NEO, ID: 33 – “eine vernünftige nicht ruckelnde maus bzw einstellung wäre nett”

Condition: NEO, ID: 35 – “Frage und Hinweis passten meist nicht zusammen, Frage nach ‘which’ aber expected answer war eine Zahl, schien komisch”

A.6.1.3 *Other feedback*

“Condition: MPS, ID: 7 – viel Erfahrung mir keiner programmiersprache :-D”

“Condition: NEO, ID: 15 – eine deutsche und eine englische Tastatur sind etwas verwirrend...”

“Condition: MPS, ID: 18 – Viel Glück!”

“Condition: MPS, ID: 24 – :)”

ACRONYMS

A

ANTLR

ANOther Tool for Language Recognition. *used on: p. 41*

API

Application programming interface. *used on: pp. 28, 77, 102, 119*

AST

abstract syntax tree. *used on: pp. 11, 35, 37, 38, 40, 41, 95, 119, 152, 161*

C

CI

Continuous Integration. *used on: pp. 124, 127, 128, 131*

CITEC

Cluster of Excellence Cognitive Interactive Technology at Bielefeld University. *used on: pp. 53, 147*

CITk

Cognitive Interaction Toolkit. *used on: pp. 124, 125*

CSRA

Cognitive Service Robotics Apartment as Ambient Host. *used on: pp. xii, xiv, 8, 45, 47, 49, 51, 53–58, 61, 64, 65, 72, 73, 75, 79, 101, 107, 131*

D

DBMS

Database Management System. *used on: pp. 4, 5, 11, 17, 18, 21–23, 25, 59*

DIKW

data-information-knowledge-wisdom. *used on: pp. xii, 11–13, 28, 58, 80*

DSL

domain-specific language. *used on: pp. v, xv, xvii, 5–7, 22, 31–36, 38–41, 43–45, 47, 49, 57, 73, 76, 78, 79, 81, 101, 107, 117, 123–126, 130, 131, 135–140, 150, 152, 158, 161*

E

EBNF

Extended Backus–Naur Form. *used on: pp. xv, 33, 38, 41, 85, 86, 112, 157*

EISE

Embodied Interaction in Smart Environments. *used on: pp. xii, 3, 6–8, 11, 19, 21, 22, 24–28, 42, 45, 51–53, 55, 58, 62, 64, 70–73,*

75, 76, 81, 88, 93, 94, 98, 102, 107, 110, 128, 131, 133, 135, 136, 141–143, 151, 158, 161, 162, 165

EISEQD

Embodied Interaction in Smart Environments (EISE) Query Designer. used on: pp. xii, 81, 107, 109, 128, 130, 131, 133, 135, 140, 141, 146, 150–153, 158, 162, 165–167

EMF

Eclipse Modeling Framework. *used on: p. 38*

G

GDB

Graph Database Management System. *used on: pp. xii, 4, 11, 12, 16, 18, 19, 21–24, 28, 44, 77, 96, 100, 101, 112, 119, 141*

GDQ

graph database query. *used on: pp. v, 81, 161, 162*

GPL

General Purpose Language. *used on: pp. 5, 30, 41, 78, 101*

GQL

graph query language. *used on: pp. 7, 12, 18, 22–24, 28, 34, 44, 76–78, 84, 85, 88, 89, 96, 98, 100, 101, 112, 116, 158*

H

HRI

human–robot interaction. *used on: pp. v, 3, 4, 6, 11, 14, 52, 54, 68, 72, 78, 161*

I

IDE

integrated development environment. *used on: pp. 6, 7, 33, 41, 47, 79–81, 83, 101, 102, 107, 119, 126, 128, 131, 135, 136, 152, 153, 157, 158, 162*

L

LOC

(Source) Lines Of Code. *used on: pp. 43, 79, 137, 138, 153*

LOP

Language-oriented Programming. *used on: p. 45*

M

M₂M

model-to-model. *used on: pp. 33, 38, 40, 41, 108, 114, 116, 118, 158*

M₂T

model-to-text. *used on: pp. 33, 40, 41, 108, 114, 116, 117*

MDE

Model-driven Engineering. *used on: p. 29*

MDSE

Model-driven Software Engineering. *used on: pp. v, xii, 5–9, 11, 14, 15, 27, 29–32, 41–48, 58, 65, 72, 75, 76, 97, 102, 103, 131, 135, 136, 138–140, 152, 153, 161*

MOF

Meta-Object Facility. *used on: pp. xii, 31, 38, 41, 161*

MPS

JetBrains Meta Programming System. *used on: pp. xii, 39–41, 101, 107–110, 112–114, 117–120, 123–127, 136–138, 151, 152, 162*

O

OGM

Object Graph Mapping. *used on: p. 22*

OMG

Object Management Group. *used on: pp. xii, 31, 38, 80, 102*

OWL

OWL Web Ontology Language. *used on: pp. 63, 68*

P

PGQL

Property Graph Query Language. *used on: p. 27*

R

RDF

Resource Description Framework. *used on: pp. 19, 25, 63, 68, 78*

ROS

Robot Operating System. *used on: pp. 4, 13, 14*

RSB

Robotics Service Bus. *used on: pp. 56, 101*

RST

Robotics Systems Types. *used on: p. 101*

S

SAT

boolean satisfiability problem. *used on: p. 158*

SLAM

Simultaneous Localization and Mapping. *used on: p. 69*

SMT

satisfiability modulo theories. *used on: p. 158*

SPARQL

SPARQL Protocol and *Resource Description Framework (RDF)* Query Language. *used on: pp. xv, 19, 23–26, 68, 78*

SQL

Structured Query Language. *used on: pp. 5, 17, 19, 23, 25–28, 79, 102, 147*

SUS

System Usability Scale. *used on: pp. [xiii](#), [139](#), [142](#), [145](#), [146](#), [148](#), [150–153](#), [165](#)*

T

TLX

NASA Task Load Index. *used on: pp. [xiii](#), [139](#), [140](#), [144–149](#), [151](#), [153](#), [157](#), [165](#)*

U

UEQ

User Experience Questionnaire. *used on: pp. [xiii](#), [139](#), [142](#), [145](#), [146](#), [148](#), [150](#), [151](#), [153](#), [166](#)*

UML

Unified Modeling Language. *used on: pp. [36](#), [83](#)*

W

W3C

World Wide Web Consortium. *used on: pp. [25](#), [66](#), [70](#), [91](#), [115](#)*

X

XML

Extensible Markup Language. *used on: p. [127](#)*

GLOSSARY

A

abstract syntax

“The abstract syntax is a data structure that can hold the semantically relevant information expressed by a program. It is typically a tree or a graph. It does not contain any details about the notation – for example, in textual languages, it does not contain keywords, symbols or whitespace.” [Völ13a, p. 26] *used on: pp. xvii, 30, 33, 34, 38, 40, 42, 43, 47, 83, 108, 150*

artifact

Artifacts of *domain-specific language (DSL)* generation are commonly defined as the results of *model* transformations. In the case of *model-to-text (M2T)* transformation artifacts often they refer to generated source code or text, while the results of *model-to-model (M2M)* transformations commonly are other *models*. Further, *M2T* artifacts can also be libraries or a compiled *integrated development environment (IDE)*. In the context of software building and *Continuous Integration (CI)* servers, artifacts refer to any result of a successful build process. *used on: pp. 30, 32, 40, 47, 72, 76, 78, 81, 84, 114, 116, 123, 124, 126, 128, 135, 138, 152*

B

behavior developers

Behavior developers are a group of actors commonly involved in the *Cognitive Service Robotics Apartment as Ambient Host (CSRA)* environment who are closely involved in the system development and make use of available capabilities. They are domain experts on how to create suitable interactions for the *naïve users* via the available actuation and interaction mechanisms. Interaction design and creation is their core task in this role and thus this task requires them to have appropriate domain-specific knowledge and programming skills. As a result of their specificity, *behavior developers* do not necessarily know which interaction relevant data, information, or knowledge is stored, and in what format the data is stored and accessed optimally. *used on: pp. v, 4–6, 24, 58–60, 68, 70–72, 76, 79, 81, 128, 161*

C

cognitive projectional editor gap

Developers are used to traditional text based inputs and the common workflow involves only parser-based programming

support. Using a projectional editor for the first time consequently requires the developers to change their mental model and think on the model or concept layer. The learning curve for unexperienced users is thus commonly very steep. However, once developers adjust and adapt their mental representation to the projectional schema, they overcome their *cognitive projectional editor gap* and reach similar or better performance as before. *used on: pp. 152, 153, 157*

concrete syntax

“The concrete syntax defines the notation with which users can express programs. It may be textual, graphical, tabular or a mix of these.” [Völ13a, p. 26] *used on: pp. xii, xvii, 25, 30, 33, 34, 38–40, 42, 47, 83, 92, 94, 101, 108, 109, 113–117, 120, 121, 128, 129, 131, 137, 150, 157*

D

denotational semantics

The *denotational semantics* of a language describe its behavior by formalizing the meanings via mathematical constructs. The syntax independence provides a maximal abstraction to describe the language actions. In this thesis the provided semantics follow the notations as presented by Hennessy [Hen90; Com17]. *used on: pp. xvii, 8, 34, 83, 85, 86, 99, 102, 107, 116, 118, 162*

domain-specific language

A domain-specific language is a programming language designed to have limited expressiveness to provide a focused access to a particular domain. This contrasts to the usual programming approach makes extensive use of *General Purpose Languages (GPLs)*. Within large software system, individual DSLs usually only target one specific aspect of the overall system [Fow10]. *used on: pp. v, xvii, 5–7, 23, 31–36, 38–41, 43–45, 47, 49, 57, 73, 76, 78, 79, 81, 101, 107, 117, 123–126, 130, 131, 135–140, 150, 152, 158, 161*

G

General Purpose Language

“General Purpose Programming Languages (GPLs) are a means for programmers to instruct computers. All of them are Turing complete, which means that they can be used to implement anything that is computable with a Turing machine. It also means that anything expressible with one Turing complete programming language can also be expressed with any other Turing complete programming language. In that sense, all programming languages are interchangeable.” [Völ13a, p. 27] *used on: pp. 5, 30, 41, 78, 101*

H

human–robot interaction

“Human–Robot Interaction (HRI) is a field of study dedicated to understanding, designing, and evaluating robotic systems for use by or with humans. Interaction, by definition, requires communication between robots and humans.” [GS07] *used on: pp. v, 3, 4, 6, 11, 14, 52, 54, 68, 72, 78, 161*

L

labeled property multidigraph

A graph G which is defined by $G = (V, E, \rho, \lambda, \sigma)$. Such a graph allows to (1) contain multiple directed edges, (2) attach multiple properties from finite sets Prop and constants Const to nodes and edges, (3) attach different labels from a finite set Lab to nodes and edges, and (4) contain multiple edges between nodes with identical source, target, and label(s). *used on: pp. 16, 18, 19, 22, 23, 25, 28, 44, 82, 84, 89, 97, 103*

language workbench

“The essential [five] characteristics of a language workbench [are:] [1] Users can freely define new languages which are fully integrated with each other. [2] The primary source of information is a persistent abstract representation. [3] Language designers define a *DSL* in three main parts: schema, editor(s), and generator(s). [4] Language users manipulate a *DSL* through a projectional editor. [5] A language workbench can persist incomplete or contradictory information in its abstract representation.” [Fow05] *used on: pp. xii, 7, 8, 37–39, 41, 42, 47, 83, 100, 101, 107, 112, 124, 136–138, 162*

Language-oriented Programming

“Language Oriented Programming to mean the general style of development which operates about the idea of building software around a set of *DSLs*” [Fow05] *used on: p. 45*

loop

Loops, see *loop* for more information. *used on: p. 15*

M

M₀

Layer 0 of the *Object Management Group (OMG)* meta-modeling layers: “the concrete level (any real situation, unique in space and time, represented by a given model from)” [BG01, p. 3] *used on: p. 30*

M₁

Layer 1 of the *OMG* meta-modeling layers: “the model level (any model with a corresponding meta-model from *M₂*” [BG01, p. 3] *used on: pp. 30, 45, 77, 95, 161, 162*

M₂

Layer 2 of the OMG meta-modeling layers: “the meta-model level (contains any kind of meta-model, including the UML meta-model” [BG01, p. 3]) *used on: pp. 30, 45, 77, 81, 95, 161*

M₃

Layer 3 of the OMG meta-modeling layers[BG01] called the meta-meta-model level. *used on: pp. 30, 81*

meta-model

“A *meta-model* is a model whose instances define the schema for another model.” [Fow10] It thus defines the *abstract syntax* of a language which is used to define valid language concepts [Völ13a]. *used on: pp. xii, 5–7, 14, 30, 33, 37, 41–43, 47, 65, 75, 77, 81, 83–85, 88, 89, 91–95, 107, 110, 112, 114, 115, 130, 131, 137, 161*

model

“A model is a simplification of a system built with an intended goal in mind (...). The model should be able to answer questions in place of the actual system. The answers provided by the model should be the same as those given by the system itself, on the condition that questions are within the domain defined by the general goal of the system.” [BG01, p. 2] *used on: pp. 5–8, 11, 29, 30, 32, 37, 38, 40, 42–44, 47, 49, 51, 53, 59, 61, 65, 70, 72, 75–78, 80, 81, 83, 84, 88, 95, 97, 108, 109, 114, 116, 118, 123, 124, 127, 129, 135, 151, 161, 162*

Model-driven Engineering

Synonym for *Model-driven Software Engineering (MDSE)*. *used on: p. 29*

Model-driven Software Engineering

“MDE is a software engineering approach that considers models not just as documentation artefacts but also as first-class citizens, where models might be used throughout all engineering disciplines and in any application domain.” [Rod15a] *used on: pp. v, 5–9, 11, 14, 15, 27, 29–32, 41–48, 58, 65, 72, 75, 76, 97, 102, 103, 131, 135, 136, 138–140, 152, 153, 161*

module

A JetBrains *JetBrains Meta Programming System (MPS)* module organizes models into higher level groupings. Therefore modules usually consist of multiple *models* along with the required meta information describing the relevant module properties and dependencies. *MPS* differentiates between multiple types of modules: languages, generators, devkits, and solutions. *used on: pp. xii, 100, 101, 108–110, 112, 114, 117–119, 121, 123–126, 131*

O

Object Graph Mapping

A mapping between graph elements (nodes, edges, properties, etc.) and plain old Java objects. *used on: p. 22*

P

pragmatics

The pragmatics of a language are described by the practical concerns and considerations of a *DSL* implementation. They describe “how modeling languages can be used in a more efficient and appropriate way [...] [and] [...] also refers practical aspects of using modeling languages and MDE on real-world projects” [Rod15a, p. 6]. Examples are practical features such as language completion, quick-fixes, or code refactoring. *used on: pp. 7, 38, 41, 42, 47, 76, 83, 92, 94, 100, 107, 108, 116, 118–120, 130, 131*

S

sharding

“Often, a busy data store is busy because different people are accessing different parts of the dataset. In these circumstances we can support horizontal scalability by putting different parts of the data onto different servers – a technique that’s called sharding.” [SF13, p. 46] *used on: p. 22*

smart environment

“Smart environments combine perceptual and reasoning capabilities with the other elements of ubiquitous computing in an attempt to create a human-centered system that is embedded in physical spaces. [...] A smart environment is a small world where all kinds of smart devices are continuously working to make inhabitants’ lives more comfortable” [CD05] *used on: pp. v, 3, 49, 52, 54, 57, 64, 72*

solution

A *MPS* solution is the entry level *module* and represents a set of *models*. End user *models* are often referred to as *sandbox solutions*, while *runtime solutions* allow to provide code to other *modules*, such as Java classes, sources or jar files. Lastly, *plugin solutions* allow to extend the *IDE* functionality by providing menu entries, tool panels, windows, or other features. *used on: pp. xii, 109, 114, 123, 128, 136*

U

usability

Qualitative characteristic of software as defined in [ISO9126] (withdrawn and succeeded by [ISO25010]) describing the extend to which software reaches a certain *quality in use*. *used on: pp. v, 45, 136–140, 151–153, 157, 159, 162*

V

vertical prototype

“As soon as you have a reasonable understanding of the TECHNOLOGY-INDEPENDENT ARCHITECTURE and the TECHNOLOGY MAPPING, make sure you test the non-functional requirements. Build a vertical prototype: an application that uses all of the above and implements it only for a very small subset of the functional requirements. This specifically includes performance and load tests. [...] you have to verify that the programming model does not result in problems with regard to QoS later. You have to make sure the various aspects you define in your architecture really work together.” [Völ+13, p. 269] *used on: pp. v, 6–8, 45, 47, 101–103, 105, 107, 112, 117, 119, 125, 126, 130, 133, 135, 157, 162*

BIBLIOGRAPHY

INVOLVED AND OWN PUBLICATIONS

- [Hol+16b] Patrick Holthaus et al. “How to Address Smart Homes with a Social Robot? A Multi-modal Corpus of User Interactions with an Intelligent Environment.” In: *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016)*. Ed. by Nicoletta Calzolari et al. Paris, France: European Language Resources Association (ELRA), 2016. *used on: pp. 51, 54, 64*
- [KWC18a] Norman Köster, Sebastian Wrede, and Philipp Cimiano. “A Model Driven Approach for Eased Knowledge Storage and Retrieval in Interactive HRI Systems.” In: *2018 Second IEEE International Conference on Robotic Computing (IRC)*. IEEE, 2018, pp. 113–120. DOI: [10.1109/IRC.2018.00025](https://doi.org/10.1109/IRC.2018.00025). *used on: p. 75*
- [KWC18b] Norman Köster, Sebastian Wrede, and Philipp Cimiano. “An Ontology for Modelling Human Machine Interaction in Smart Environments.” In: *Proceedings of SAI Intelligent Systems Conference (IntelliSys) 2016*. Ed. by Yaxin Bi, Supriya Kapoor, and Rahul Bhatia. Vol. 16. Lecture Notes in Networks and Systems. Cham: Springer International Publishing, 2018, pp. 338–350. ISBN: 978-3-319-56990-1. DOI: [10.1007/978-3-319-56991-8_25](https://doi.org/10.1007/978-3-319-56991-8_25). *used on: p. 51*
- [KWC18c] Norman Köster, Sebastian Wrede, and Philipp Cimiano. “Evaluating a Graph Query Language for Human-Robot Interaction Data in Smart Environments.” In: *Software Technologies: Applications and Foundations*. Ed. by Martina Seidl and Steffen Zschaler. STAF 2017 Collocated Workshops. Vol. 10748. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 263–279. ISBN: 978-3-319-74729-3. DOI: [10.1007/978-3-319-74730-9_24](https://doi.org/10.1007/978-3-319-74730-9_24). *used on: pp. 135, 136*
- [KWC19] Norman Köster, Sebastian Wrede, and Philipp Cimiano. “Evaluation of a Model-driven Knowledge Storage and Retrieval IDE for Interactive HRI Systems.” In: *International Journal of Semantic Computing* 13 (02 2019), pp. 207–227. DOI: [10.1142/S1793351X19400099](https://doi.org/10.1142/S1793351X19400099). *used on: p. 136*
- [Wie+18] Johannes Wienke et al. “Model-Based Performance Testing for Robotics Software Components.” In: *2018 Second IEEE International Conference on Robotic Computing (IRC)*. IEEE, 2018, pp. 25–32. DOI: [10.1109/IRC.2018.00013](https://doi.org/10.1109/IRC.2018.00013).

GENERAL

- [All84] James F. Allen. "Towards a general theory of action and time." In: *Artificial Intelligence* 23 (2 1984), pp. 123–154. ISSN: 00043702. DOI: [10.1016/0004-3702\(84\)90008-0](https://doi.org/10.1016/0004-3702(84)90008-0). used on: pp. 91, 93, 100
- [Ang+17] Renzo Angles et al. "Foundations of Modern Query Languages for Graph Databases." In: *ACM Computing Surveys* 50 (5 2017), pp. 1–40. ISSN: 03600300. DOI: [10.1145/3104031](https://doi.org/10.1145/3104031). used on: pp. 23–26
- [Ang12] Renzo Angles. "A Comparison of Current Graph Database Models." In: *2012 IEEE 28th International Conference on Data Engineering Workshops*. IEEE, 2012, pp. 171–177. DOI: [10.1109/ICDEW.2012.31](https://doi.org/10.1109/ICDEW.2012.31). used on: pp. 17, 18
- [Atk+00] C. G. Atkeson et al. "Using humanoid robots to study human behavior." In: *IEEE Intelligent Systems* 15 (4 2000), pp. 46–56. ISSN: 1094-7167. DOI: [10.1109/5254.867912](https://doi.org/10.1109/5254.867912). used on: p. 3
- [Atz+13] Paolo Atzeni et al. "The relational model is dead, SQL is dead, and I don't feel so good myself." In: *ACM SIGMOD Record* 42 (1 2013), p. 64. ISSN: 01635808. DOI: [10.1145/2503792.2503808](https://doi.org/10.1145/2503792.2503808). used on: p. 17
- [Bad+09] David A. Bader et al. "Hpc scalable graph analysis benchmark." In: *Citeseer*. Citeseer 2009 (2009), pp. 1–10. used on: p. 21
- [BAG18] Ankica Barišić, Vasco Amaral, and Miguel Goulão. "Usability driven DSL development with USE-ME." In: *Computer Languages, Systems & Structures* 51 (2018), pp. 118–157. ISSN: 14778424. DOI: [10.1016/j.cl.2017.06.005](https://doi.org/10.1016/j.cl.2017.06.005). used on: pp. 44–46, 137
- [Bal+17] Ferenc Balint-Benczedi et al. "Storing and retrieving perceptual episodic memories for long-term manipulation tasks." In: *2017 18th International Conference on Advanced Robotics (ICAR)*. IEEE, 2017, pp. 25–31. DOI: [10.1109/ICAR.2017.8023492](https://doi.org/10.1109/ICAR.2017.8023492). used on: pp. 38, 68, 78, 91
- [Bar+12] Ankica Barišić et al. "How to reach a usable DSL? Moving toward a Systematic Evaluation." Multi-Paradigm Modeling 2011. In: *Electronic Communications of the EASST* 50 (2012). DOI: [10.14279/tuj.eceasst.50.741](https://doi.org/10.14279/tuj.eceasst.50.741). used on: pp. 44, 45, 137
- [Bar13] Ankica Barišić. "Iterative evaluation of domain-specific languages." In: *CEUR Workshop Proceedings* 1115 (2013), pp. 100–105. ISSN: 16130073. used on: pp. 45, 137
- [BBL76] Barry W. Boehm, John R. Brown, and Mlity Lipow. "Quantitative evaluation of software quality." In: *Proceedings of the 2nd international conference on Software engineering*. IEEE Computer Society Press, 1976, pp. 592–605. used on: p. 135
- [BC10] Jonathan Bohren and Steve Cousins. "The SMACH High-Level Executive [ROS News]." In: *IEEE Robotics & Automation Magazine* 17 (4 2010), pp. 18–20. ISSN: 1070-9932. DOI: [10.1109/MRA.2010.938836](https://doi.org/10.1109/MRA.2010.938836). used on: p. 14

- [BE17] Jasmin Bernotat and Friederike Eyssel. "An Evaluation Study of Robot Designs for Smart Environments." In: *Proceedings of the Companion of the 2017 ACM/IEEE International Conference on Human-Robot Interaction - HRI '17*. Ed. by Bilge Mutlu et al. New York, New York, USA: ACM Press, 2017, pp. 87–88. DOI: [10.1145/3029798.3038429](https://doi.org/10.1145/3029798.3038429). used on: p. 56
- [BE18] Jasmin Bernotat and Friederike Eyssel. "Can('t) Wait to Have a Robot at Home? - Japanese and German Users' Attitudes Toward Service Robots in Smart Homes." In: *2018 27th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)*. IEEE, 2018, pp. 15–22. DOI: [10.1109/ROMAN.2018.8525659](https://doi.org/10.1109/ROMAN.2018.8525659). used on: p. 54
- [Bee+18] Michael Beetz et al. "KnowRob 2.0. A 2nd generation knowledge processing framework for cognition-enabled robotic agents." In: *2018 IEEE International Conference on Robotics and Automation (ICRA) : 21-25 May 2018*. Ed. by Kevin Lynch. [Piscataway, NJ]: IEEE, 2018, pp. 512–519. ISBN: 978-1-5386-3081-5. used on: pp. 4, 5, 14, 44, 61, 63, 67, 78
- [Ben73] Melvin A. Benarde. *Our precarious habitat*. WW Norton & Company, 1973. used on: p. 75
- [BES19] Jasmin Bernotat, Friederike Eyssel, and Janik Sachse. "The (Fe)male Robot: How Robot Body Shape Impacts First Impressions and Trust Towards Robots." In: *International Journal of Social Robotics* 85 (4 2019), p. 768. ISSN: 1875-4791. DOI: [10.1007/s12369-019-00562-7](https://doi.org/10.1007/s12369-019-00562-7). used on: p. 4
- [BG01] J. Bezivin and O. Gerbe. "Towards a precise definition of the OMG/MDA framework." In: *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*. IEEE Comput. Soc, 2001, pp. 273–280. DOI: [10.1109/ASE.2001.989813](https://doi.org/10.1109/ASE.2001.989813). used on: pp. 29, 195, 196
- [BKMo8] Aaron Bangor, Philip T. Kortum, and James T. Miller. "An Empirical Evaluation of the System Usability Scale." In: *International Journal of Human-Computer Interaction* 24 (6 2008), pp. 574–594. ISSN: 1044-7318. DOI: [10.1080/10447310802205776](https://doi.org/10.1080/10447310802205776). used on: pp. 139, 151
- [Bla09] Elizabeth Blackburn. "A conversation with Elizabeth Blackburn. Interview by Misia Landau." In: *Clinical chemistry* 55 (4 2009), pp. 835–841. DOI: [10.1373/clinchem.2008.119578](https://doi.org/10.1373/clinchem.2008.119578). eprint: [19233908](https://doi.org/10.1093/clinchem/55.4.835). used on: p. 11
- [Brio8] Robert Bringhurst. *The elements of typographic style*. Version 3.2. Point Roberts, Wash.: Hartley & Marks, 2008. ISBN: 978-0-88179-206-5. used on: p. 221
- [Bro90] Rodney A. Brooks. "Elephants don't play chess." In: *Robotics and Autonomous Systems* 6 (1-2 1990), pp. 3–15. ISSN: 09218890. DOI: [10.1016/S0921-8890\(05\)80025-9](https://doi.org/10.1016/S0921-8890(05)80025-9). used on: p. 52
- [Bro96] John Brooke. "SUS-A quick and dirty usability scale." In: *Usability evaluation in industry* 189 (194 1996), pp. 4–7. used on: pp. 135, 139, 148, 151

- [BTW15] Michael Beetz, Moritz Tenorth, and Jan Winkler. "Open-EASE." In: *2015 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2015, pp. 1983–1990. DOI: [10.1109/ICRA.2015.7139458](https://doi.org/10.1109/ICRA.2015.7139458). used on: pp. 63, 68
- [But+19] Arvid Butting et al. "Systematic Composition of Independent Language Features." In: *Journal of Systems and Software* (2019). ISSN: 01641212. DOI: [10.1016/j.jss.2019.02.026](https://doi.org/10.1016/j.jss.2019.02.026). used on: p. 37
- [Cav+14] Filippo Cavallo et al. "Improving Domiciliary Robotic Services by Integrating the ASTRO Robot in an AmI Infrastructure." In: *Gearing up and accelerating cross-fertilization between academic and industrial robotics research in Europe*. Ed. by Florian Röhrbein, Germano Veiga, and Ciro Natale. Vol. 94. Springer Tracts in Advanced Robotics. Cham: Springer International Publishing, 2014, pp. 267–282. ISBN: 978-3-319-02933-7. DOI: [10.1007/978-3-319-02934-4_13](https://doi.org/10.1007/978-3-319-02934-4_13). used on: p. 53
- [CB76] Donald D. Chamberlin and Raymond F. Boyce. "SEQUEL." In: *Proceedings of the 1976 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control - FIDET '76*. Ed. by Gene Altshuler, Randall Rustin, and Bernard Plagman. New York, New York, USA: ACM Press, 1976, pp. 249–264. DOI: [10.1145/800296.811515](https://doi.org/10.1145/800296.811515). used on: p. 17
- [CD05] Diane J. Cook and Sajal K. Das. *Smart environments. Technologies, protocols, and applications*. Vol. v.43. Wiley Series on Parallel and Distributed Computing. Hoboken, NJ: John Wiley, 2005. ISBN: 978-0-471-54448-7. DOI: [10.1002/047168659X](https://doi.org/10.1002/047168659X). used on: pp. 3, 52, 197
- [Cer+15] Mario Cervera et al. "On the usefulness and ease of use of a model-driven Method Engineering approach." In: *Information Systems* 50 (2015), pp. 36–50. ISSN: 03064379. DOI: [10.1016/j.is.2015.01.006](https://doi.org/10.1016/j.is.2015.01.006). used on: p. 138
- [Cla66] Berge Claude. *Théorie des graphes et ses applications*. French. Dunod, Paris, 1966. used on: p. 15
- [Cod70] E. F. Codd. "A relational model of data for large shared data banks." In: *Communications of the ACM* 13 (6 1970), pp. 377–387. ISSN: 00010782. DOI: [10.1145/362384.362685](https://doi.org/10.1145/362384.362685). used on: p. 17
- [Com+12] Michael Compton et al. "The SSN ontology of the W3C semantic sensor network incubator group." In: *Journal of Web Semantics* 17 (2012), pp. 25–32. ISSN: 15708268. DOI: [10.1016/j.websem.2012.05.003](https://doi.org/10.1016/j.websem.2012.05.003). used on: pp. 66, 70
- [Com17] Benoit Combemale. *Engineering modeling languages*. Chapman & Hall/CRC Innovations in Software Engineering and Software Development. Boca Raton, Florida, London, [England], and New York: CRC Press, 2017. ISBN: 978-1-315-38793-2. used on: pp. 5, 29, 33–35, 44–47, 81, 194

- [CSW16] Birte Carlmeyer, David Schlangen, and Britta Wrede. "Exploring self-interruptions as a strategy for regaining the attention of distracted users." In: *Proceedings of the 1st Workshop on Embodied Interaction with Smart Environments - EISE '16*. Ed. by Unknown. New York, New York, USA: ACM Press, 2016, pp. 1–6. DOI: [10.1145/3008028.3008029](https://doi.org/10.1145/3008028.3008029). used on: p. 4
- [Dat12] Chris J. Date. *SQL and relational theory. How to write accurate SQL code*. 2nd ed. Theory in practice. Sebastopol, Calif.: O'Reilly, 2012. ISBN: 978-1-4493-1640-2. used on: p. 17
- [Dat84] C. J. Date. "A critique of the SQL database language." In: *ACM SIGMOD Record* 14 (3 1984), pp. 8–54. ISSN: 01635808. DOI: [10.1145/984549.984551](https://doi.org/10.1145/984549.984551). used on: p. 17
- [Dat87] C. J. Date. "Where SQL falls short." In: *Datamation* 33 (9 1987), pp. 83–89. used on: p. 17
- [Dom+10] D. Dominguez-Sal et al. "Survey of Graph Database Performance on the HPC Scalable Graph Analysis Benchmark." In: *Web-Age Information Management*. Ed. by Heng Tao Shen et al. Vol. 6185. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 37–48. ISBN: 978-3-642-16719-5. DOI: [10.1007/978-3-642-16720-1_4](https://doi.org/10.1007/978-3-642-16720-1_4). used on: pp. 20, 21
- [Dup07] Lyn Dupré. *BUGS in writing. A guide to debugging your prose*. Rev ed., 10. print. Boston: Addison-Wesley, 2007. ISBN: 978-0-201-37921-1. used on: p. 221
- [DZK15] Andre Dietrich, Sebastian Zug, and Jorg Kaiser. "SELECTSCRIPT: A query language for robotic world models and simulations." In: *2015 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2015, pp. 6254–6260. DOI: [10.1109/ICRA.2015.7140077](https://doi.org/10.1109/ICRA.2015.7140077). used on: p. 79
- [EGR12] Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. "Language composition untangled." In: *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications - LDTA '12*. Ed. by Anthony Sloane and Suzana Andova. New York, New York, USA: ACM Press, 2012, pp. 1–8. DOI: [10.1145/2427048.2427055](https://doi.org/10.1145/2427048.2427055). used on: p. 37
- [Erd+13] Sebastian Erdweg et al. "The State of the Art in Language Workbenches." In: *Software Language Engineering*. Ed. by David Hutchison et al. Vol. 8225. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2013, pp. 197–217. ISBN: 978-3-319-02653-4. DOI: [10.1007/978-3-319-02654-1_11](https://doi.org/10.1007/978-3-319-02654-1_11). used on: pp. 35, 81
- [Erd+15] Sebastian Erdweg et al. "Evaluating and comparing language workbenches." In: *Computer Languages, Systems & Structures* 44 (2015), pp. 24–47. ISSN: 14778424. DOI: [10.1016/j.cl.2015.08.007](https://doi.org/10.1016/j.cl.2015.08.007). used on: pp. 38, 42
- [ES12] Ayssam Elkady and Tarek Sobh. "Robotics Middleware: A Comprehensive Literature Survey and Attribute-Based Bibliography." In: *Journal of Robotics* 2012 (5 2012), pp. 1–15. ISSN: 1687-9600. DOI: [10.1155/2012/959013](https://doi.org/10.1155/2012/959013). used on: p. 5

- [EU17] RobMoSys EU. *H2020 Project RobMoSys: Composable Models and Software for Robotics Systems-Towards an EU Digital Industrial Platform for Robotics*. Tech. rep. EU, 2017. used on: p. 44
- [EV06] Sven Efftinge and Markus Völter. “oAW xText: A framework for textual DSLs.” In: *Workshop on Modeling Symposium at Eclipse Summit*. 2006, p. 118. used on: p. 38
- [FBoo] Martin Fowler and Kent Beck. *Refactoring: Improving the design of existing code*. 4. print. Object technology, software engineering. Reading, Mass. [u.a.]: Addison-Wesley, 2000. ISBN: 978-0-201-48567-7. used on: p. 32
- [FB14] Normen E. Fenton and James Bieman. *Software metrics. A rigorous and practical approach / Norman E. Fenton and James Bieman*. Third edition. Chapman & Hall/CRC Innovations in Software Engineering and Software Development. Boca Raton: CRC Press, 2014. ISBN: 978-1-4398-3823-5. used on: p. 135
- [Fer15] Ghofrane Fersi. “Middleware for Internet of Things: A Study.” In: *2015 International Conference on Distributed Computing in Sensor Systems*. IEEE, 2015, pp. 230–235. DOI: [10.1109/DCOSS.2015.43](https://doi.org/10.1109/DCOSS.2015.43). used on: p. 5
- [Fis+18] Tobias Fischer et al. “iCub-HRI: A Software Framework for Complex Human–Robot Interaction Scenarios on the iCub Humanoid Robot.” In: *Frontiers in Robotics and AI* 5 (2018), p. 4807. DOI: [10.3389/frobt.2018.00022](https://doi.org/10.3389/frobt.2018.00022). used on: p. 5
- [FMF12] Andy P. Field, Jeremy Miles, and Zoë Field. *Discovering statistics using R*. London: SAGE, 2012. ISBN: 978-1-4462-0046-9. used on: p. 147
- [FND03] Terrence Fong, Illah Nourbakhsh, and Kerstin Dautenhahn. “A survey of socially interactive robots.” In: *Robotics and Autonomous Systems* 42 (3-4 2003), pp. 143–166. ISSN: 09218890. DOI: [10.1016/S0921-8890\(02\)00372-X](https://doi.org/10.1016/S0921-8890(02)00372-X). used on: p. 3
- [Foo13] Tully Foote. “tf: The transform library.” In: *2013 IEEE Conference on Technologies for Practical Robot Applications (TePRA)*. IEEE, 2013, pp. 1–6. DOI: [10.1109/TePRA.2013.6556373](https://doi.org/10.1109/TePRA.2013.6556373). used on: pp. 4, 14
- [Fou+17] Dehann Fourie et al. “SLAMinDB: Centralized graph databases for mobile robotics.” In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2017, pp. 6331–6337. DOI: [10.1109/ICRA.2017.7989749](https://doi.org/10.1109/ICRA.2017.7989749). used on: pp. 11, 69
- [Fow10] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010. used on: pp. 5, 31, 194, 196
- [Fra+18] Nadime Francis et al. “Cypher.” In: *Proceedings of the 2018 International Conference on Management of Data - SIGMOD '18*. Ed. by Gautam Das, Christopher Jermaine, and Philip Bernstein. New York, New York, USA: ACM Press, 2018, pp. 1433–1445. DOI: [10.1145/3183713.3190657](https://doi.org/10.1145/3183713.3190657). used on: p. 24
- [Gam07] Erich Gamma. *Design patterns. Elements of reusable object oriented software*. Addison-Wesley professional computing series. Boston [u.a.]: Addison-Wesley, 2007. ISBN: 978-0-201-63361-0. used on: p. 127

- [Gar+07] E. Garcia et al. "The evolution of robotics research." In: *IEEE Robotics & Automation Magazine* 14 (1 2007), pp. 90–103. ISSN: 1070-9932. DOI: [10.1109/MRA.2007.339608](https://doi.org/10.1109/MRA.2007.339608). used on: p. 3
- [GGA10] Pedro Gabriel, Miguel Goulão, and Vasco Amaral. "Do Software Languages Engineers Evaluate their Languages?" In: *Proceedings of the XIII Congreso Iberoamericano en "Software Engineering" (CIbSE'2010)* (2010). used on: pp. 45, 135, 136, 159
- [Groo8] Jonathan L. Gross, ed. *Handbook of graph theory*. [Online-aus.] Discrete mathematics and its applications. Boca Raton: CRC Press, 2008. ISBN: 978-0-203-49020-4. used on: p. 15
- [GSo7] Michael A. Goodrich and Alan C. Schultz. "Human-Robot Interaction: A Survey." In: *Foundations and Trends® in Human-Computer Interaction* 1 (3 2007), pp. 203–275. ISSN: 1551-3955. DOI: [10.1561/1100000005](https://doi.org/10.1561/1100000005). used on: pp. 3, 195
- [Guo+14] Yong Guo et al. "How Well Do Graph-Processing Platforms Perform? An Empirical Performance Evaluation and Analysis." In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 2014, pp. 395–404. DOI: [10.1109/IPDPS.2014.49](https://doi.org/10.1109/IPDPS.2014.49). used on: p. 22
- [Har06] Sandra G. Hart. "NASA-task load index (NASA-TLX); 20 years later." In: *Proceedings of the human factors and ergonomics society annual meeting*. Sage publications Sage CA: Los Angeles, CA, 2006, pp. 904–908. DOI: [10.1037/e577632012-009](https://doi.org/10.1037/e577632012-009). used on: p. 140
- [HC15] Justin Huang and Maya Cakmak. "Supporting mental model accuracy in trigger-action programming." In: *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing - UbiComp '15*. Ed. by Kenji Mase et al. New York, New York, USA: ACM Press, 2015, pp. 215–225. DOI: [10.1145/2750858.2805830](https://doi.org/10.1145/2750858.2805830). used on: p. 158
- [Hebo9] John Hebler. *Semantic Web programming*. Indianapolis, IN: Wiley, 2009. ISBN: 978-0-470-41801-7. used on: p. 25
- [Hen74] Nicholas L. Henry. "Knowledge Management: A New Concern for Public Administration." In: *Public Administration Review* 34 (3 1974), p. 189. ISSN: 00333352. DOI: [10.2307/974902](https://doi.org/10.2307/974902). used on: p. 17
- [Hen90] Matthew Hennessy. *The semantics of programming languages. An elementary introduction using structural operational semantics*. Chichester: Wiley, 1990. ISBN: 978-0-471-92772-3. used on: pp. 33, 34, 194
- [HFB16] Florian Häser, Michael Felderer, and Ruth Breu. "An integrated tool environment for experimentation in domain specific language engineering." In: *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering - EASE '16*. Ed. by Sarah Beecham, Barbara Kitchenham, and Stephen G. MacDonell. New York, New York, USA: ACM Press, 2016, pp. 1–5. DOI: [10.1145/2915970.2916010](https://doi.org/10.1145/2915970.2916010). used on: p. 137

- [Hoc+16] Nico Hochgeschwender et al. "Graph-based software knowledge: Storage and semantic querying of domain models for run-time adaptation." In: *2016 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAN)*. IEEE, 2016, pp. 83–90. DOI: [10.1109/SIMPAN.2016.7862379](https://doi.org/10.1109/SIMPAN.2016.7862379). used on: pp. 11, 43
- [Hol+16a] Patrick Holthaus et al. "1st international workshop on embodied interaction with smart environments (workshop summary)." In: *Proceedings of the 18th ACM International Conference on Multimodal Interaction - ICMI 2016*. Ed. by Yukiko I. Nakano et al. New York, New York, USA: ACM Press, 2016, pp. 589–590. DOI: [10.1145/2993148.3007628](https://doi.org/10.1145/2993148.3007628). used on: p. 3
- [Hon+12] Sungpack Hong et al. "Green-Marl: A DSL for Easy and Efficient Graph Analysis." In: *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '12*. Ed. by Tim Harris and Michael L. Scott. New York, New York, USA: ACM Press, 2012, p. 349. DOI: [10.1145/2150976.2151013](https://doi.org/10.1145/2150976.2151013). used on: p. 43
- [HP04] Jerry R. Hobbs and Feng Pan. "An ontology of time for the semantic web." In: *ACM Transactions on Asian Language Information Processing* 3 (1 2004), pp. 66–85. ISSN: 15300226. DOI: [10.1145/1017068.1017073](https://doi.org/10.1145/1017068.1017073). used on: pp. 70, 91, 115
- [HP18] Olaf Hartig and Jorge Pérez. "Semantics and Complexity of GraphQL." In: *Proceedings of the 2018 World Wide Web Conference on World Wide Web - WWW '18*. Ed. by Pierre-Antoine Champin et al. New York, New York, USA: ACM Press, 2018, pp. 1155–1164. DOI: [10.1145/3178876.3186014](https://doi.org/10.1145/3178876.3186014). used on: pp. 27, 28
- [HR00] David Harel and Bernhard Rumpe. "Modeling languages: Syntax, semantics and all that stu." In: *N/A n/a* (2000), pp. 1–28. used on: pp. 33, 83
- [HRH16] Martin Hoppen, Juergen Rossmann, and Sebastian Hiester. "Managing 3D Simulation Models with the Graph Database Neo4j." In: *DBKDA 2016* (2016), p. 88. used on: p. 11
- [HS88] Sandra G. Hart and Lowell E. Staveland. "Development of NASA-TLX (Task Load Index): Results of Empirical and Theoretical Research." In: *Human Mental Workload*. Ed. by N. Meshkati and P. A. Hancock. Vol. 52. Advances in Psychology. Elsevier, 1988, pp. 139–183. ISBN: 978-0-444-70388-0. DOI: [10.1016/S0166-4115\(08\)62386-9](https://doi.org/10.1016/S0166-4115(08)62386-9). used on: p. 139
- [Hut+11] John Hutchinson et al. "Empirical assessment of MDE in industry." In: *Proceeding of the 33rd international conference on Software engineering - ICSE '11*. Ed. by Richard N. Taylor, Harald Gall, and Nenad Medvidović. New York, New York, USA: ACM Press, 2011, p. 471. DOI: [10.1145/1985793.1985858](https://doi.org/10.1145/1985793.1985858). used on: p. 42
- [ISO14543] KNX. 14543. ISO/IEC. 2008. URL: <https://www.knx.org/> (visited on 2018-10-08). used on: pp. 53, 56

- [ISO25010] ISO/IEC JTC 1/SC 7 Software and systems engineering. *Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE)*. 25010. ISO/IEC. Version 2011. 2011. URL: <https://www.iso.org/standard/35733.html> (visited on 2019-07-10). used on: pp. 137, 197
- [ISO9126] ISO/IEC JTC 1/SC 7 Software and systems engineering. *Software engineering – Product quality*. 9126. ISO/IEC. Version 1:2001. 2001. URL: <https://www.iso.org/standard/22749.html> (visited on 2019-07-10). used on: pp. 137, 197
- [Jon+14] Matt Jones et al., eds. *Proceedings of the 32nd annual ACM conference on Human factors in computing systems - CHI '14*. the 32nd annual ACM conference (Toronto, Ontario, Canada). New York, New York, USA: ACM Press, 2014. ISBN: 978-1-4503-2473-1. DOI: [10.1145/2556288](https://doi.org/10.1145/2556288). used on: p. 57
- [KBM16] Tomaž Kosar, Sudev Bohra, and Marjan Mernik. “Domain-Specific Languages: A Systematic Mapping Study.” In: *Information and Software Technology* 71 (2016), pp. 77–91. ISSN: 09505849. DOI: [10.1016/j.infsof.2015.11.001](https://doi.org/10.1016/j.infsof.2015.11.001). used on: pp. 45, 135–137, 159
- [Kel84] J. F. Kelley. “An iterative design methodology for user-friendly natural language office information applications.” In: *ACM Transactions on Information Systems* 2 (1 1984), pp. 26–41. ISSN: 10468188. DOI: [10.1145/357417.357420](https://doi.org/10.1145/357417.357420). used on: p. 64
- [KRV10] Holger Krahn, Bernhard Rumpe, and Steven Völkel. “MontiCore: a framework for compositional development of domain specific languages.” In: *International Journal on Software Tools for Technology Transfer* 12 (5 2010), pp. 353–372. ISSN: 1433-2779. DOI: [10.1007/s10009-010-0142-1](https://doi.org/10.1007/s10009-010-0142-1). used on: p. 37
- [KTK09] Juha Kärnä, Juha-Pekka Tolvanen, and Steven Kelly. “Evaluating the use of domain-specific modeling in practice.” In: *9th OOPSLA workshop on Domain-Specific Modeling*. 2009. used on: p. 136
- [KWB03] Anneke Kleppe, Jos B. Warmer, and Wim Bast. *MDA explained. The model driven architecture ; practice and promise*. Safari online books. Reading, Mass. and Sebastopol, CA: Addison-Wesley and Safari Books Online, 2003. ISBN: 978-0-321-19442-8. used on: p. 30
- [Lem+10] Séverin Lemaignan et al. “ORO, a knowledge management platform for cognitive architectures in robotics.” In: *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2010, pp. 3548–3553. DOI: [10.1109/IR05.2010.5649547](https://doi.org/10.1109/IR05.2010.5649547). used on: pp. 4, 14, 44, 68, 78
- [Len95] Douglas B. Lenat. “CYC.” In: *Communications of the ACM* 38 (11 1995), pp. 33–38. ISSN: 00010782. DOI: [10.1145/219717.219745](https://doi.org/10.1145/219717.219745). used on: p. 62

- [LHSo8] Bettina Laugwitz, Theo Held, and Martin Schrepp. "Construction and Evaluation of a User Experience Questionnaire." In: *HCI and Usability for Education and Work*. Ed. by Andreas Holzinger. Vol. 5298. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 63–76. ISBN: 978-3-540-89349-3. DOI: [10.1007/978-3-540-89350-9_6](https://doi.org/10.1007/978-3-540-89350-9_6). used on: pp. [135](#), [139](#), [148](#)
- [Lie+14a] Grischa Liebel et al. "Assessing the State-of-Practice of Model-Based Engineering in the Embedded Systems Domain." In: *Model-Driven Engineering Languages and Systems*. Ed. by Juer-gen Dingel et al. Vol. 8767. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014, pp. 166–182. ISBN: 978-3-319-11652-5. DOI: [10.1007/978-3-319-11653-2_11](https://doi.org/10.1007/978-3-319-11653-2_11). used on: p. [42](#)
- [Lie+14b] Florian Lier et al. "The Cognitive Interaction Toolkit – Improving Reproducibility of Robotic Systems Experiments." In: *Simulation, Modeling, and Programming for Autonomous Robots*. Ed. by Davide Brugali et al. Vol. 8810. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014, pp. 400–411. ISBN: 978-3-319-11899-4. DOI: [10.1007/978-3-319-11900-7_34](https://doi.org/10.1007/978-3-319-11900-7_34). used on: p. [124](#)
- [Lik32] Rensis Likert. "A technique for the measurement of attitudes." In: *Archives of psychology* (1932). used on: p. [146](#)
- [LLM15] Ruijiao Li, Bowen Lu, and Klaus D. McDonald-Maier. "Cognitive assisted living ambient system: a survey." In: *Digital Communications and Networks* 1 (4 2015), pp. 229–252. ISSN: 23528648. DOI: [10.1016/j.dcan.2015.10.003](https://doi.org/10.1016/j.dcan.2015.10.003). used on: pp. [55](#), [61](#)
- [Loo+14] Markus Look et al. "Black-box Integration of Heterogeneous Modeling Languages for Cyber-Physical Systems." In: *CoRR* abs/1409.2388 (2014). used on: p. [37](#)
- [Lou+15] João Ricardo Lourenço et al. "Choosing the right NoSQL database for the job: a quality attribute evaluation." In: *Journal of Big Data* 2 (1 2015), p. 12. DOI: [10.1186/s40537-015-0025-0](https://doi.org/10.1186/s40537-015-0025-0). used on: p. [22](#)
- [Lüt+11] Ingo Lütkebohle et al. "The Task-State Coordination Pattern, with applications in Human-Robot-Interaction." In: *Learning, Planning and Sharing Robot Knowledge for Human-Robot Interaction*. Ed. by Rachid Alami et al. Dagstuhl Seminar Proceedings. Dagstuhl, Germany: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2011. used on: p. [14](#)
- [Mai83] David Maier. *The theory of relational databases*. Computer science press Rockville, 1983. ISBN: 978-0-914894-42-1. used on: p. [17](#)
- [Mal+15] Anupama Mallik et al. "Ontology based context aware situation tracking." In: *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*. IEEE, 2015, pp. 687–692. DOI: [10.1109/WF-IoT.2015.7389137](https://doi.org/10.1109/WF-IoT.2015.7389137). used on: p. [67](#)
- [Mat+06] Cynthia Matuszek et al. "An introduction to the syntax and content of Cyc." In: *UMBC Computer Science and Electrical Engineering Department Collection* (2006). used on: p. [63](#)

- [MB11] Leonardo de Moura and Nikolaj Bjørner. "Satisfiability modulo theories." In: *Communications of the ACM* 54 (9 2011), p. 69. ISSN: 00010782. DOI: [10 . 1145 / 1995376 . 1995394](https://doi.org/10.1145/1995376.1995394). used on: p. 158
- [McC+14] Robert Campbell McColl et al. "A performance evaluation of open source graph databases." In: *Proceedings of the first workshop on Parallel programming for analytics applications - PPAA '14*. Ed. by Manoj Kumar, Joefon Jann, and Priya Nagpurkar. New York, New York, USA: ACM Press, 2014, pp. 11–18. DOI: [10 . 1145/2567634.2567638](https://doi.org/10.1145/2567634.2567638). used on: p. 22
- [MCD12] L. M. McAvoy, Liming Chen, and Mark Donnelly. "An ontology-based context management system for smart environments." In: *UBICOMM 2012 - 6th International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies* (2012), pp. 18–23. used on: p. 66
- [MDF05] Geoffrey R. Marczyk, David DeMatteo, and David Festinger. *Essentials of research design and methodology*. Essentials of behavioral science series. Hoboken, N.J: John Wiley & Sons, 2005. ISBN: 978-0-471-47053-3. used on: pp. 142, 146
- [Mén+16] David Méndez-Acuña et al. "Leveraging Software Product Lines Engineering in the development of external DSLs: A systematic literature review." In: *Computer Languages, Systems & Structures* 46 (2016), pp. 206–235. ISSN: 14778424. DOI: [10 . 1016/j . cl . 2016 . 09 . 004](https://doi.org/10.1016/j.cl.2016.09.004). used on: p. 35
- [MFN06] Giorgio Metta, Paul Fitzpatrick, and Lorenzo Natale. "YARP: yet another robot platform." In: *International Journal of Advanced Robotic Systems* 3 (1 2006), p. 8. used on: p. 13
- [Mik15] Peter Mika. "On Schema.org and Why It Matters for the Web." In: *IEEE Internet Computing* 19 (4 2015), pp. 52–55. ISSN: 1089-7801. DOI: [10 . 1109/MIC . 2015 . 81](https://doi.org/10.1109/MIC.2015.81). used on: p. 67
- [MK14] Dan McCreary and Ann Kelly. *Making sense of NoSQL. A guide for managers and the rest of us / Dan McCreary, Ann Kelly*. Shelter Island, New York: Manning, 2014. ISBN: 978-1-61729-107-4. used on: p. 18
- [Mor+13] Meg E. Morris et al. "Smart-home technologies to assist older people to live well at home." In: *Journal of aging science* 1 (1 2013), pp. 1–9. used on: p. 53
- [Mpi+15] Steve Ataky Tsham Mpinda et al. "Evaluation of graph databases performance through indexing techniques." In: *International Journal of Artificial Intelligence & Applications (IJAAIA) Vol 6* (2015), pp. 87–98. used on: p. 22
- [Msh+18] Haider Mshali et al. "A survey on health monitoring systems for health smart homes." In: *International Journal of Industrial Ergonomics* 66 (2018), pp. 26–56. ISSN: 01698141. DOI: [10 . 1016/ j . ergon . 2018 . 02 . 002](https://doi.org/10.1016/j.ergon.2018.02.002). used on: p. 66

- [NE16] Chandrakana Nandi and Michael D. Ernst. “Automatic Trigger Generation for Rule-based Smart Homes.” In: *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security - PLAS’16*. Ed. by Toby Murray and Deian Stefan. New York, New York, USA: ACM Press, 2016, pp. 97–102. DOI: [10.1145/2993600.2993601](https://doi.org/10.1145/2993600.2993601). used on: p. 158
- [Net+08] A. D. Neto et al. “Improving Evidence about Software Technologies: A Look at Model-Based Testing.” In: *IEEE Software* 25 (3 2008), pp. 10–13. ISSN: 0740-7459. DOI: [10.1109/MS.2008.64](https://doi.org/10.1109/MS.2008.64). used on: pp. 45, 136, 140
- [NHW14] Arne Nordmann, Nico Hochgeschwender, and Sebastian Wrede. “A Survey on Domain-Specific Languages in Robotics.” In: *Simulation, Modeling, and Programming for Autonomous Robots*. Ed. by Davide Brugali et al. Vol. 8810. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014, pp. 195–206. ISBN: 978-3-319-11899-4. DOI: [10.1007/978-3-319-11900-7_17](https://doi.org/10.1007/978-3-319-11900-7_17). used on: p. 43
- [NLK12] Nhan Nguyen-Duc-Thanh, Sungyoung Lee, and Donghan Kim. “Two-Stage Hidden Markov Model in Gesture Recognition for Human Robot Interaction.” In: *International Journal of Advanced Robotic Systems* 9 (2 2012), p. 39. DOI: [10.5772/50204](https://doi.org/10.5772/50204). used on: p. 14
- [NM+01] Natalya F. Noy, Deborah L. McGuinness, et al. *Ontology development 101: A guide to creating your first ontology*. Tech. rep. Stanford knowledge systems laboratory technical report KSL-01-05: Stanford University, 2001. used on: p. 65
- [NMC99] Clifford Nass, Youngme Moon, and Paul Carney. “Are People Polite to Computers? Responses to Computer-Based Interviewing Systems¹.” In: *Journal of Applied Social Psychology* 29 (5 1999), pp. 1093–1109. ISSN: 0021-9029. DOI: [10.1111/j.1559-1816.1999.tb00142.x](https://doi.org/10.1111/j.1559-1816.1999.tb00142.x). used on: p. 144
- [NN91] Riis Hanne Nielson and Flemming Nielson. *Semantics with applications. A formal introduction*. Chichester [u.a.]: Wiley, 1991. ISBN: 978-0-471-92980-2. used on: pp. 33, 34
- [Nor16] Arne Nordmann. “Modeling of motion primitive architectures using domain-specific languages.” Doctoral dissertation. Bielefeld: Universität Bielefeld, 2016. used on: pp. 44–46
- [Omg08] Q. V.T. Omg. “Meta object facility (mof) 2.0 query/view/transformation specification.” In: *Final Adopted Specification (November 2005)* (2008). used on: pp. 30, 31
- [Pes+15] Ana Pescador et al. “Pattern-based development of Domain-Specific Modelling Languages.” In: *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 2015, pp. 166–175. DOI: [10.1109/MODELS.2015.7338247](https://doi.org/10.1109/MODELS.2015.7338247). used on: p. 76
- [Pic10] Robert Pickering. “Language-Oriented Programming.” In: *Beginning F#*. Ed. by Robert Pickering. Berkeley, CA: Apress, 2010, pp. 327–349. ISBN: 978-1-4302-2389-4. DOI: [10.1007/978-1-4302-2390-0_12](https://doi.org/10.1007/978-1-4302-2390-0_12). used on: pp. 35, 81

- [PS16] Nick Papoulias and Serge Stinckwich. "Towards projection: mapping reflection onto the userland." In: *Companion Proceedings of the 15th International Conference on Modularity - MODULARITY Companion 2016*. Ed. by Lidia Fuentes, Don Batory, and Krzysztof Czarnecki. New York, New York, USA: ACM Press, 2016, pp. 172–175. DOI: [10.1145/2892664.2892696](https://doi.org/10.1145/2892664.2892696). used on: p. [151](#)
- [Qui+09] Morgan Quigley et al. "ROS: an open-source Robot Operating System." In: *ICRA workshop on open source software*. Kobe, Japan, 2009, p. 5. used on: p. [13](#)
- [Ric+06] Vincent Ricquebourg et al. "The Smart Home Concept : our immediate future." In: *2006 1ST IEEE International Conference on E-Learning in Industrial Electronics*. IEEE, 2006, pp. 23–28. DOI: [10.1109/ICELIE.2006.347206](https://doi.org/10.1109/ICELIE.2006.347206). used on: p. [52](#)
- [Ric+16] Viktor Richter et al. "Are you talking to me? Improving the Robustness of Dialogue Systems in a Multi Party HRI Scenario by Incorporating Gaze Direction and Lip Movement of Attendees." In: *Proceedings of the Fourth International Conference on Human Agent Interaction - HAI '16*. Ed. by Wei Yun Yau et al. New York, New York, USA: ACM Press, 2016, pp. 43–50. DOI: [10.1145/2974804.2974823](https://doi.org/10.1145/2974804.2974823). used on: pp. [4](#), [54](#)
- [RK18] Viktor Richter and Franz Kummert. "Continuous Interaction Data Acquisition and Evaluation." In: *Companion of the 2018 ACM/IEEE International Conference on Human-Robot Interaction - HRI '18*. Ed. by Takayuki Kanda et al. New York, New York, USA: ACM Press, 2018, pp. 217–218. DOI: [10.1145/3173386.3177005](https://doi.org/10.1145/3173386.3177005). used on: p. [54](#)
- [Rod+14] Natalia Díaz Rodríguez et al. "A survey on ontologies for human behavior recognition." In: *ACM Computing Surveys* 46 (4 2014), pp. 1–33. ISSN: 03600300. DOI: [10.1145/2523819](https://doi.org/10.1145/2523819). used on: p. [66](#)
- [Rod15a] Alberto Rodrigues da Silva. "Model-driven engineering: A survey supported by the unified conceptual model." In: *Computer Languages, Systems & Structures* 43 (2015), pp. 139–155. ISSN: 14778424. DOI: [10.1016/j.cl.2015.06.001](https://doi.org/10.1016/j.cl.2015.06.001). used on: pp. [5](#), [14](#), [29](#), [30](#), [38](#), [42](#), [76](#), [100](#), [196](#), [197](#)
- [Rod15b] Marko A. Rodriguez. "The Gremlin graph traversal machine and language (invited talk)." In: *Proceedings of the 15th Symposium on Database Programming Languages - DBPL 2015*. Ed. by James Cheney and Thomas Neumann. New York, New York, USA: ACM Press, 2015, pp. 1–10. DOI: [10.1145/2815072.2815073](https://doi.org/10.1145/2815072.2815073). used on: p. [26](#)
- [Row07] Jennifer Rowley. "The wisdom hierarchy: representations of the DIKW hierarchy." In: *Journal of Information Science* 33 (2 2007), pp. 163–180. ISSN: 0165-5515. DOI: [10.1177/0165551506070706](https://doi.org/10.1177/0165551506070706). used on: pp. [12](#), [13](#)
- [Sax+14] Ashutosh Saxena et al. *RoboBrain: Large-Scale Knowledge Engine for Robots*. Tech. rep. Cornell University and Stanford University, 2014. URL: <http://arxiv.org/pdf/1412.0691v2>. used on: p. [69](#)

- [SF13] Pramod J. Sadalage and Martin Fowler. *NoSQL distilled. A brief guide to the emerging world of polyglot persistence*. Always learning. Upper Saddle River, NJ: Addison-Wesley/Pearson, 2013. ISBN: 978-0-321-82662-6. used on: pp. 17, 197
- [Sfa+13] G. Sfakianakis et al. “Interval indexing and querying on key-value cloud stores.” In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 2013, pp. 805–816. DOI: [10.1109/ICDE.2013.6544876](https://doi.org/10.1109/ICDE.2013.6544876). used on: p. 96
- [SGK17] Sebastian Schneider, Michael Goerlich, and Franz Kummert. “A framework for designing socially assistive robot interactions.” In: *Cognitive Systems Research* 43 (2017), pp. 301–312. ISSN: 13890417. DOI: [10.1016/j.cogsys.2016.09.008](https://doi.org/10.1016/j.cogsys.2016.09.008). used on: p. 14
- [SLS17] Christian Schlegel, Alex Lotz, and Dennis Stampfer. *Composable models and software for robotics systems. Deliverable D2.2: Initial preparation of (meta-)models, prototypical DSLs, tools and implementation*. Tech. rep. EU, 2017. URL: http://robmosys.eu/wp-content/uploads/2017/03/D2.2_Final.pdf (visited on 2019-10-25). used on: p. 44
- [SP16] Konstantinos Semertzidis and Evaggelia Pitoura. “Time Traveling in Graphs using a Graph Database.” In: *EDBT/ICDT Workshops*. 2016. used on: p. 96
- [SS09] Steffen Staab and Rudi Studer. *Handbook on Ontologies*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. ISBN: 978-3-540-70999-2. DOI: [10.1007/978-3-540-92673-3](https://doi.org/10.1007/978-3-540-92673-3). used on: p. 65
- [ŞvV18] Ana Maria Şutfii, Mark van den Brand, and Tom Verhoeff. “Exploration of modularity and reusability of domain-specific languages: an expression DSL in MetaMod.” In: *Computer Languages, Systems & Structures* 51 (2018), pp. 48–70. ISSN: 14778424. DOI: [10.1016/j.cl.2017.07.004](https://doi.org/10.1016/j.cl.2017.07.004). used on: p. 35
- [SW09] William Strunk and Elwyn B. White. *The elements of style*. 4. ed. New York, NY: Longman, 2009. ISBN: 978-0-205-30902-3. used on: p. 221
- [SW19] Joshua Shinavier and Ryan Wisnesky. *Algebraic Property Graphs*. Tech. rep. Uber, 2019. URL: <http://arxiv.org/pdf/1909.04881v1>. used on: pp. 16, 24, 26, 158
- [TB09] Jonas Tappolet and Abraham Bernstein. “Applied Temporal RDF: Efficient Temporal Querying of RDF Data with SPARQL.” In: *The Semantic Web: Research and Applications*. Ed. by Lora Aroyo et al. Vol. 5554. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 308–322. ISBN: 978-3-642-02120-6. DOI: [10.1007/978-3-642-02121-3_25](https://doi.org/10.1007/978-3-642-02121-3_25). used on: pp. 91, 96
- [TB11] Bogdan George Tudorica and Cristian Bucur. “A comparison between several NoSQL databases with comments and notes.” In: *2011 RoEduNet International Conference 10th Edition: Networking in Education and Research*. IEEE, 2011, pp. 1–5. DOI: [10.1109/RoEduNet.2011.5993686](https://doi.org/10.1109/RoEduNet.2011.5993686). used on: pp. 18, 22

- [TB13] Moritz Tenorth and Michael Beetz. "KnowRob: A knowledge processing infrastructure for cognition-enabled robots." In: *The International Journal of Robotics Research* 32 (5 2013), pp. 566–590. ISSN: 0278-3649. DOI: [10.1177/0278364913481635](https://doi.org/10.1177/0278364913481635). used on: pp. 4, 14, 44, 61, 63, 67, 78
- [The18] The Economist. *Style Guide. The Bestselling Guide to English Usage*. First US edition. New York: Public Affairs, 2018. ISBN: 978-1-61039-538-0. used on: p. 221
- [Vic+10] Chad Vicknair et al. "A comparison of a graph database and a relational database." In: *Proceedings of the 48th Annual Southeast Regional Conference on - ACM SE '10*. Ed. by H. Conrad Cunningham, Paul Ruth, and Nicholas A. Kraft. New York, New York, USA: ACM Press, 2010, p. 1. DOI: [10.1145/1900008.1900067](https://doi.org/10.1145/1900008.1900067). used on: p. 22
- [vKVoo] Arie van Deursen, Paul Klint, and Joost Visser. "Domain-specific languages." In: *ACM SIGPLAN Notices* 35 (6 2000), pp. 26–36. ISSN: 03621340. DOI: [10.1145/352029.352035](https://doi.org/10.1145/352029.352035). used on: pp. 5, 31, 32
- [Völ+12] Markus Völter et al. "mbeddr: an extensible C-based programming language and IDE for embedded systems." In: *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*. ACM. 2012, pp. 121–140. used on: pp. 136, 138
- [Völ+13] Markus Völter et al. *Model-Driven Software Development. Technology, Engineering, Management*. 1. Aufl. Wiley Software Patterns Series. s.l.: Wiley, 2013. ISBN: 978-0-470-02570-3. used on: pp. 29, 198
- [Völ+14] Markus Völter et al. "Towards User-Friendly Projectional Editors." In: *Software Language Engineering*. Ed. by Benoît Combemale et al. Vol. 8706. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014, pp. 41–61. ISBN: 978-3-319-11244-2. DOI: [10.1007/978-3-319-11245-9_3](https://doi.org/10.1007/978-3-319-11245-9_3). used on: p. 151
- [Völ+19] Markus Völter et al. "Lessons learned from developing mbeddr: a case study in language engineering with MPS." In: *Software & Systems Modeling* 18 (1 2019), pp. 585–630. ISSN: 1619-1366. DOI: [10.1007/s10270-016-0575-4](https://doi.org/10.1007/s10270-016-0575-4). used on: pp. 136, 138
- [Völ06] Markus Völter. "Software Architecture - A pattern language for building sustainable software architectures." In: *Eleventh European Conference on Pattern Languages of Programs*. 2006, pp. 31–66. used on: pp. 44, 79
- [Völ09] Markus Völter. "Best practices for DSLs and model-driven development." In: *Journal of Object Technology* 8 (6 2009), pp. 79–102. used on: pp. 44, 45
- [Völ13a] Markus Völter. *DSL engineering. Designing, implementing and using domain-specific languages*. In collab. with Sebastian Benz et al. Lexington, KY: CreateSpace Independent Publishing Platform, 2013. ISBN: 978-1-4812-1858-0. used on: pp. 5, 30, 32, 35, 42, 44, 46, 81, 193, 194, 196

- [Völ13b] Markus Völter. “Language and IDE Modularization and Composition with MPS.” In: *Generative and Transformational Techniques in Software Engineering IV*. Ed. by Ralf Lämmel, João Saraiva, and Joost Visser. Vol. 7680. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 383–430. ISBN: 978-3-642-35991-0. DOI: [10.1007/978-3-642-35992-7_11](https://doi.org/10.1007/978-3-642-35992-7_11). used on: pp. 35–37, 41
- [Völ18] Markus Völter. “The Design, Evolution, and Use of KernelF.” In: *Theory and Practice of Model Transformation*. Ed. by Arend Rensink and Jesús Sánchez Cuadrado. Vol. 10888. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 3–55. ISBN: 978-3-319-93316-0. DOI: [10.1007/978-3-319-93317-7_1](https://doi.org/10.1007/978-3-319-93317-7_1). used on: p. 32
- [VP12] Markus Völter and Vaclav Pech. “Language modularity with the MPS language workbench.” In: *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 1449–1450. used on: pp. 35, 40
- [Wal07] Danny P. Wallace. *Knowledge management. Historical and cross-disciplinary themes*. Libraries Unlimited knowledge management series. Westport, Conn: Libraries Unlimited, 2007. ISBN: 978-1-59158-502-2. used on: p. 65
- [War95] M. P. Ward. “Language Oriented Programming.” In: *Software—Concepts and Tools* 15 (1995), pp. 147–161. used on: pp. 45, 46
- [Weg+13] Timo Wegeler et al. “Evaluating the benefits of using domain-specific modeling languages.” In: *Proceedings of the 2013 ACM workshop on Domain-specific modeling - DSM '13*. Ed. by Jeff Gray, Steven Kelly, and Jonathan Sprinkle. New York, New York, USA: ACM Press, 2013, pp. 7–12. DOI: [10.1145/2541928.2541930](https://doi.org/10.1145/2541928.2541930). used on: pp. 45, 137
- [Wes01] Douglas Brent West. *Introduction to graph theory*. 2. ed. Upper Saddle River, NJ: Prentice Hall, 2001. ISBN: 978-0-13-014400-3. used on: pp. 15, 16
- [Wig+17] Dennis Leroy Wigand et al. “Domain-Specific Language Modularization Scheme Applied to a Multi-Arm Robotics Use-Case.” In: *Journal of Software Engineering for Robotics* 8 (1 2017), pp. 45–64. ISSN: 2035-3928. used on: pp. 32, 37
- [Wil45] Frank Wilcoxon. “Individual Comparisons by Ranking Methods.” In: *Biometrics Bulletin* 1 (6 1945), p. 80. ISSN: 00994987. DOI: [10.2307/3001968](https://doi.org/10.2307/3001968). used on: p. 147
- [Wil99] Robin J. Wilson. *Introduction to graph theory*. 4. ed., repr. Harlow: Longman, 1999. ISBN: 978-0-582-24993-6. used on: pp. 15, 16
- [Won+12] Konlakorn Wongpatikaseree et al. “Activity Recognition Using Context-Aware Infrastructure Ontology in Smart Home Domain.” In: *2012 Seventh International Conference on Knowledge, Information and Creativity Support Systems*. IEEE, 2012, pp. 50–57. DOI: [10.1109/KICSS.2012.26](https://doi.org/10.1109/KICSS.2012.26). used on: p. 67

- [Wre+17] Sebastian Wrede et al. “The Cognitive Service Robotics Apartment.” In: *KI - Künstliche Intelligenz* 31 (3 2017), pp. 299–304. ISSN: 0933-1875. DOI: [10.1007/s13218-017-0492-x](https://doi.org/10.1007/s13218-017-0492-x). used on: pp. 4, 53
- [WW11] Johannes Wienke and Sebastian Wrede. “A middleware for collaborative research in experimental robotics.” In: *2011 IEEE/SICE International Symposium on System Integration (SII)*. IEEE, 2011, pp. 1183–1190. DOI: [10.1109/SII.2011.6147617](https://doi.org/10.1109/SII.2011.6147617). used on: p. 13
- [YD04] H. A. Yanco and J. Drury. “Classifying human-robot interaction: an updated taxonomy.” In: *2004 IEEE International Conference on Systems, Man and Cybernetics (IEEE Cat. No.04CH37583)*. IEEE, 2004, pp. 2841–2846. DOI: [10.1109/ICSMC.2004.1400763](https://doi.org/10.1109/ICSMC.2004.1400763). used on: pp. 70, 72
- [Zha+18] Yunming Zhang et al. “GraphIt: a high-performance graph DSL.” In: *Proceedings of the ACM on Programming Languages* 2 (OOPSLA 2018), pp. 1–30. DOI: [10.1145/3276491](https://doi.org/10.1145/3276491). used on: p. 43

ONLINE RESOURCES

- [Bie13] Bielefeld University. *The Cognitive Service Robotics Apartment as Ambient Host*. 2013. URL: <https://cit-ec.de/en/cognitive-service-robotics-apartment-ambient-host> (visited on 2019-02-12). used on: p. 53
- [BM14] Dan Brickley and Libby Miller. *FOAF Vocabulary Specification*. 2014. URL: <http://xmlns.com/foaf/spec/> (visited on 2020-01-10). used on: p. 70
- [Fow05] Martin Fowler. *Language Workbenches: The Killer-App for Domain Specific Languages?* 2005. URL: <https://www.martinfowler.com/articles/languageWorkbench.html> (visited on 2018-10-08). used on: pp. 38, 151, 195
- [Gui11] Erico Guizzo. *Meka Robotics Announces Mobile Manipulator With Kinect and ROS*. IEEE. 2011. URL: <https://spectrum.ieee.org/automaton/robotics/humanoids/meke-robotics-announces-mobile-manipulator-with-kinect-and-ros> (visited on 2019-08-01). used on: p. 56
- [ISO19] ISO Graph Query Language Proponents. *Graph Query Language GQL. GQL Standard*. 2019. URL: <https://www.gqlstandards.org/gql-blogs/critical-milestone-for-iso-graph-query-standard-gql> (visited on 2019-11-08). used on: pp. 27, 28, 158
- [Jet18] JetBrains. *JetBrains MPS*. 2018. URL: <https://github.com/JetBrains/MPS> (visited on 2018-10-08). used on: p. 39
- [Mah17] Alaa Mahmoud. *No more joins: An overview of Graph database query languages*. IBM. 2017. URL: <https://developer.ibm.com/dwblog/2017/overview-graph-database-query-languages/> (visited on 2019-04-16). used on: p. 24
- [Neo19] Neo Technology. *The GQL Manifesto - One Property Graph Query Language*. 2019. URL: <https://gql.today/> (visited on 2019-11-08). used on: pp. 27, 28

- [Obj14] Object Management Group. *Model Driven Architecture (MDA). Guide Rev. 2.0*. 2014. URL: <https://www.omg.org/mda/> (visited on 2019-08-16). *used on: pp. 44, 45*
- [ope13] openHAB Community. *openHAB*. 2013. URL: <https://www.openhab.org/> (visited on 2019-10-07). *used on: p. 57*
- [PU03] Woody Pidcock and Michael Uschold. *What are the differences between a vocabulary, a taxonomy, a thesaurus, an ontology, and a meta-model*. 2003. URL: <https://web.archive.org/web/20090310162812/http://www.metamodel.com/article.php?story=20030115211223271> (visited on 2019-09-21). *used on: p. 65*
- [RDS17] Jonathan Robie, Dyck Michael, and Spiegel Josh, eds. *XML Path Language (XPath)*. W3C. 2017. URL: <http://www.w3.org/TR/2017/REC-xpath-31-20170321/> (visited on 2019-08-09). *used on: p. 26*
- [sol19] solid IT. *DB-Engines Ranking. DB-Engines Ranking of Graph DBMS*. 2019. URL: <https://db-engines.com/en/ranking/graph+dbms> (visited on 2019-04-16). *used on: pp. 20, 21*
- [Typ11] Typefox. *Xtext - Modernized Java*. 2011. URL: <http://www.xtend-lang.org/> (visited on 2018-10-08). *used on: p. 39*
- [W3Co8] W3C. *SPARQL Query Language for RDF. W3C Recommendation 15 January 2008*. 2008. URL: <https://www.w3.org/TR/rdf-sparql-query/> (visited on 2019-04-11). *used on: p. 25*
- [W3C17] W3C. *Time Ontology in OWL*. 2017. URL: <http://www.w3.org/TR/owl-time/> (visited on 2019-08-26). *used on: pp. 91, 115*

SOFTWARE PACKAGES

- [Apa00] *Ant*. Version 1.10.5. Apache Software Foundation. 2000. URL: <https://ant.apache.org/> (visited on 2018-10-08). *used on: p. 124*
- [Ecl] *Xtext. Language Engineering For Everyone*. Eclipse Project. URL: <http://xtext.org> (visited on 2018-10-08). *used on: pp. 38, 39, 41*
- [Fac16] *GraphQL. GraphQL Specification*. Facebook Inc. 2016. URL: <http://facebook.github.io/> (visited on 2018-10-08). *used on: p. 27*
- [ite17] *mbeddr. An embedded development system*. Version 2017.2.0. itemis. 2017. URL: <http://mbeddr.com> (visited on 2018-11-16). *used on: p. 136*
- [Jen] *Jenkins*. Jenkins. URL: <http://jenkins.io> (visited on 2018-11-16). *used on: pp. 124, 125*
- [Jet] *MPS. Meta Programming System*. Version 2018.2.1. JetBrains. URL: <https://www.jetbrains.com/mps> (visited on 2018-11-16). *used on: pp. 8, 39, 107, 109, 117, 121–123, 126, 128, 129, 141, 142*
- [Neo07] *Neo4j. The Graph Platform*. Neo Technology. 2007. URL: <https://neo4j.com/> (visited on 2018-10-08). *used on: pp. 4, 16, 18, 22, 24, 25, 28, 44, 69, 101, 102, 112–114, 117, 119, 121, 122, 141, 142*

- [Neo11] *Cypher*. Neo Technology. 2011. URL: <https://neo4j.com/developer/cypher-query-language/> (visited on 2018-10-08). *used on: pp. xv, 19, 20, 22–28, 34, 35, 44, 69, 85, 87, 88, 91, 98, 99, 101, 102, 112–114, 116, 117, 121, 122, 129, 141, 143–145, 157, 158*
- [Neo15] *openCypher*. Neo Technology. 2015. URL: <http://www.opencypher.org/> (visited on 2018-10-08). *used on: pp. xv, 19, 20, 22, 24, 25, 28, 85, 86, 112*
- [Sch12] *LimeSurvey. An open source survey tool*. Schmitz, Carsten. 2012. URL: <http://www.limesurvey.org> (visited on 2019-08-06). *used on: p. 144*
- [Ter] *ANTLR. ANother Tool for Language Recognition*. Terence Parr and others. URL: <https://www.antlr.org/> (visited on 2018-10-08). *used on: p. 41*

DECLARATION OF AUTHORSHIP

According to the Bielefeld University's doctoral degree regulations §8(1)g: I hereby declare to acknowledge the current doctoral degree regulations of the Faculty of Technology at Bielefeld University. Furthermore, I certify that this thesis has been composed by me and is based on my own work, unless stated otherwise. Third parties have neither directly nor indirectly received any monetary advantages in relation to mediation advises or activities regarding the content of this thesis. Also, no other person's work has been used without due acknowledgment. All references and verbatim extracts have been quoted, and all sources of information, including graphs and data sets, have been specifically acknowledged. This thesis or parts of it have neither been submitted for any other degree at this university nor elsewhere.

Norman Köster

Place, Date

COLOPHON

This thesis is typeset using the classicthesis typographical look-and-feel developed by André Miede and is based on the template created by Johannes Wienke. The applied style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*" [Bri08]. The writing style has been influenced by Strunk and White [SW09], Dupré [Dup07], and The Economist [The18].