University of Latvia
Faculty of Computing · Institute of Mathematics and Computer Science

Renārs Liepiņš

Definition Methods and Implementation of Domain-Specific Modeling Language Tools

in partial fulfilment of the requirements
of the doctor degree in Computer Science

Field: Computer Science
Subdiscipline: Programming Languages and Systems

Thesis Supervisor:
Prof. Jānis Bārzdiņš
Dr.habil.Sc.Comp.

Rīga, 2015

The doctoral thesis was carried out at the Institute of Mathematics and Computer Science, University of Latvia, from 2009 to 2015.

IEGULDĪJUMS TAVĀ NĀKOTNĒ

This work has been supported by the European Social Fund
within the project «Support for Doctoral Studies at University of Latvia - 2».

The thesis contains the introduction, 8 chapters, the conclusion, and the list of references.
Form of the thesis: dissertation in Computer Science, subdiscipline of Programming Languages and Systems.

Supervisor:

Prof., Dr.habil.Sc.Comp. Jānis Bārzdiņš
University of Latvia

Reviewers:

1) Dr.sc.comp. J.Bičevskis
2) Dr.habil.sc.ing. J.Grundspeņķis
3) Dr.sc.comp. U.Sarkans

The thesis will be defended at the public session of the Doctoral Committee of Computer Science, University of Latvia, at 16:30 on August 31, 2015, at the Institute of Mathematics and Computer Science (Raiņa blvd. 29, Rīga), Room 413.

The thesis is available at the Library of University of Latvia (Multi-branched Library: Computer Science, Law and Theology), Raiņa blvd. 19, Room 203.

Chairman of the Doctoral Committee: .... /Jānis Bārzdiņš/

Secretary of the Doctoral Committee: .... /Ruta Ikauniece/

ABSTRACT

In this thesis, two new approaches for Domain-specific modeling language tool definition are considered – the model-based approach and the ontology-based approach. The research is done in the context of the tool building platform GRAF developed at IMCS UL, first by investigating the technologies needed for its implementation, and secondly, by developing a vision and base components for its future evolution. In the model-based direction, the main result is a new efficient transformation language lQuery that is specifically designed for tool building tasks. In the ontology-based direction, the author proposes a vision and architecture for the future version of the platform, that will use ontologies as the base metamodeling layer. To approach the vision, the author has developed a conceptually new metamodel and notation for the ontology language OWL and an orthogonal extension of OWL with transformation language expressions for the non-monotonic reasoning tasks.

Keywords: graphical tool building platform, domain-specific languages, transformation language, ontology-based development

CONTENTS

i INTRODUCTION AND THEORETICAL BACKGROUND 1

1 INTRODUCTION 3

- 1.1 Objectives and Tasks of the Research 3
- 1.2 The Clause and Research Directions 4
- 1.3 Research Methods Used 5
- 1.4 Main Results of the Thesis 5
- 1.5 Scientific and Practical Significance of the Results 6
- 1.6 Validation of the Results 8
  - 1.6.1 Publications on the Topic of the Thesis 8
  - 1.6.2 Presentations at Scientific Conferences 10
  - 1.6.3 Scientific Projects in Which the Results Where Used 11
  - 1.6.4 Applying the Results in Practice 12
- 1.7 Size and Structure of the Thesis 12

2 THEORETICAL BACKGROUND 14

- 2.1 Domain Specific Languages 14
- 2.2 Model Driven Engineering 14
- 2.3 Ontologies and Reasoners 16

ii MDE BASED TOOL BUILDING 17

3 MODEL-BASED TOOL BUILDING FRAMEWORK GRAF 18

- 3.1 Graph Diagramming Metamodel and Engine 22
- 3.2 Tool Definition Metamodel: the Core 26
- 3.3 Tool Definition Metamodel: Extensions 32

4 LQUERY – A TRANSFORMATION LANGUAGE FOR TOOL BUILDING 38

- 4.1 Brief Overview of Lua 39
- 4.2 Overview of the Metamodel 41

4.3 lQuery 42
4.3.1 Example Model 42
4.3.2 lQuery Core 44
4.3.3 Selector Combinators 46
4.3.4 Selector Reuse and Custom Selector Combinators 50
4.3.5 Custom Primitive Selectors 50
4.3.6 Shorthand Notation 53
4.3.7 Manipulation with Whole Sets of Objects 55
4.4 Related Work 58
4.5 Conclusions 61
iii TOWARDS ONTOLOGY BASED TOOL BUILDING 62
5 WHY ONTOLOGY BASED TOOL BUILDING – A MOTIVATING EXAMPLE 63
6 UML-INSPIRED METAMODEL AND NOTATION FOR THE OWL ONTOLOGY LANGUAGE 68
6.1 OWL as a Description Layer Above Class Diagrams 69
6.2 Extension of the Core Metamodel 72
6.2.1 Equivalent and Disjoint Classes and Properties 72
6.2.2 Class Expressions 75
6.2.3 Anonymous Classes 76
6.2.4 Enumerated Classes 76
6.2.5 Further Metamodel Extensions 77
6.3 The Editor for the Proposed Notation – OWLGrEd 78
6.4 Related Work 80
6.5 Conclusions 81
7 EXTENDING OWL ONTOLOGIES WITH QUERY-BASED CONSTRUCTOR CLASSES 83
7.1 Motivating Example – a University Ontology 84
7.2 lQuery selectors in Ontologies 87
7.2.1 Integration with Ontology 87

7.2.2 Some Examples from the University Ontology 89
7.2.3 Advanced Example from the University Ontology 91
7.3 Integration with a Reasoner 93
7.4 Related Work 95
7.5 Conclusions 96
8 ontology-based tool building framework: architecture proposal 98
8.1 A Runtime Example in the Proposed Architecture 101
8.2 Conclusions 108

conclusions 109

bibliography 111

LIST OF FIGURES

Figure 1 The graph diagramming metamodel 24
Figure 2 The tool definition metamodel. 27
Figure 3 Tool definition metamodel: extensions. 33
Figure 4 Metamodeling Language Used by lQuery 42
Figure 5 Example model and instances 43
Figure 6 The lQuery selector expression grammar in a BNF notation 55
Figure 7 Flowchart validity constraints written in a natural language and in OWL 64
Figure 8 Validity constraint examples during the editing process 66
Figure 9 "mini-university" ontology (UML notation and OWL Functional Syntax) 71
Figure 10 UMLOWLCoreExtended metamodel; classes in bright yellow are UMLOWLCore metamodel (equivalent to metamodel shown in Figure 4) 73
Figure 11 "mini-university" ontology (UMLOWLCore-Extended notation) 74
Figure 12 An African Wildlife ontology in OWLGrEd editor 79
Figure 13 OWLGrEd downloads by cities outside Latvia from Nov 1st, 2013 till May 31st, 2015 81
Figure 14 Simplified University Ontology. 85
Figure 15 Fragment of the University Ontology with instances. 86
Figure 16 OWL metamodel extension with lQuery constructor classes 88

Figure 17 Demonstration of the OWLGrEd syntax extension for the lQuery expressions 89
Figure 18 lQuery expressions as annotation properties in Manchester syntax 89
Figure 19 Demonstration of the lQuery semantics in contrast to OWL semantics 90
Figure 20 Demonstration of the lQuery aggregation expressions. 91
Figure 21 Extended example from Figure 15. 92
Figure 22 Overall architecture of the MDA based tool building framework from Chapter 3. 99
Figure 23 New overall architecture of OWL-based tool building framework. 100
Figure 24 Fragment of the Tool Definition metamodel in the New Repository – Logical View and Internal Representation 102
Figure 25 Fragment of the tool definition metamodel with activity diagram specific subclasses (Logical View and Internal Representation) 103
Figure 26 The state of the tool definition metamodel with one empty activity diagram (Logical View and Internal Representation) 105
Figure 27 The state of the tool definition metamodel with one empty activity diagram (Logical View and Internal Representation) 107

LIST OF TABLES

Table 1 Compartment type subclasses 29
Table 2 Call points from *XElementType* 34
Table 3 Call points from *XCompartmentType* 36
Table 4 Primitive Selector Constructors 45

Table 5 Selector Combinators 49
Table 6 Custom Primitive Selector Constructors 53
Table 7 Selector Shorthand Notation 54
Table 8 Object Collection Methods 59

Part I

INTRODUCTION AND THEORETICAL BACKGROUND

The research presented here was carried out at the Institute of Mathematics and Computer Science, University of Latvia (IMCS UL) in the Research Laboratory of Modeling and Software Technologies continuing the research in graphical tool building that began at IMCL UL in 1980s.

1 INTRODUCTION

Domain-specific modeling is a software engineering methodology for system specification. Its main principle is using of domain-specific modeling languages [58] (DSMLs) to describe various parts of a system. Compared to traditional system specification approaches, domain-specific modeling requires fewer low-level details and reduces the effort required to specify a system. However, efficient use of domain-specific modeling requires tool support (such as editors and compilers). To develop a proper tool support for a new modeling language from scratch would require a large market to cover the development costs. Thus, GDSLs are mainly developed for large domains, such as BPM for the business process modeling domain.

Since 1990s, there has been an ongoing research to enable rapid, cost-effective development of DSMLs for *small domains*. Moreover, currently there are some commercial and open source tools (called MetaCase Tools [40]) for the creation of DSML tool support. However, a satisfactory solution for the development of DSML support tools still does not exist, because all of the existing MetaCase Tools require substantial additional programming effort to achieve a fully functional and usable domain specific tool.

We will approach this problem in two novel ways. First, by applying the Model Driven Engineering [62] (MDE) principles and technologies, and secondly by exploring how ontologies and reasoning software can be used for DSML support tool development.

1.1 OBJECTIVES AND TASKS OF THE RESEARCH

In 2007 an initiative was started at IMCS UL to develop a graphical tool building framework GRAF [68] that would be based on metamodels and model transformations. From the very beginning of the

initiative, the author participated in the development of the ideas and concepts of this framework. The goal of this Thesis is to provide a significant contribution to the further development of the GRAF framework, and also to advance the principles for the future versions of the framework, that would be based on the ontologies and reasoning software. To achieve these goals the following tasks were formulated:

- provide a significant contribution (at least 20% of the total effort involved) to the further development of the principles and architecture of the tool building framework GRAF;
- develop a new kind of transformation language (lQuery) that is specifically designed for the tasks that occur in tool building frameworks. Create an effective implementation for the language;
- develop principles for the future version of the framework that would be based on ontologies and reasoning software, and to create the necessary services and extensions so that the framework can be used in practice. Specifically:
- develop and implement a graphical notation and metamodel for the ontology language OWL;
- develop an orthogonal extension for the ontology language OWL that augments it with transformation language selector expressions for non-monotonic reasoning tasks.

1.2 THE CLAUSE AND RESEARCH DIRECTIONS

1. Model-based tool building tasks have distinct requirements for transformation languages, in comparison to traditional model-based software development use-cases. Thus, a new transformation language is needed; that is specifically designed for such tasks;
2. A graphical Domain Specific Language for the ontology language OWL can be created by using the ideas from the UML

Class Diagrams notation, thus, solving one of the main problems of adopting ontologies for the task of DSML tool specification.

1.3 RESEARCH METHODS USED

The main research methods where:

- *comparative analysis* of existing solutions;
- *logical deduction* to derive new solutions based on the knowledge of the existing shortcomings and requirements;
- *experimental implementations* to validate the ideas of the proposed solutions.

1.4 MAIN RESULTS OF THE THESIS

- Development of the principles and architecture of the Tool Building framework GRAF. Author's contribution is approximately 20% of the total effort involved;
- Design and implementation of an original transformation language (lQuery): notation, semantics, and compiler software;
- Development of the principles and architecture for a new kind of tool building framework that is based on ontologies and reasoning software. To enable a practical development of such a framework the following two problems were solved:
- A novel metamodel and graphical notation for the ontology language OWL that combines the familiar notation of the UML Class Diagrams with the class expressions and semantics of the OWL;
- An orthogonal extension of the ontology language OWL that merges logical inference strengths of the semantic reasoners with the closed world query and navigation expressions from

a transformation language, thus providing a much richer expressive power than each technology separately.

- A sophisticated editor for the proposed UML-inspired graphical notation for the ontology language OWL; the editor is actively used worldwide (approximately 100 downloads each month);
- A metamodel for the ontology language OWL, which enables easy porting of UML Class Diagram based transformation languages to work with OWL ontologies.

1.5 SCIENTIFIC AND PRACTICAL SIGNIFICANCE OF THE RESULTS

The three scientific main results of the thesis are: a transformation language (lQuery); a novel UML-inspired metamodel and graphical notation for ontology language OWL; and an extension of the ontology language OWL with the query expressions from the transformation language lQuery.

The scientific significance of the transformation language (lQuery) is the demonstration that when model transformations are used at runtime, they have significantly different requirements from the traditional model transformation use-cases, and thus require consideration of different tradeoffs than the traditional transformation languages.

The practical significance of the lQuery: it is the main transformation language used in the GRAF Tool Building framework developed at IMCS UL. In fact, all the transformation runtime software and graphical tool configurator software (developed by Artūrs Sproģis as part of his thesis [67]) were implemented in lQuery. This language is also used for the implementation of all the extension transformations for DSML tools that are defined by using the GRAF Tool Building framework. An example of the specific DSML tool developed in this way: the tool [28] used in the Latvian State Social Insurance Agency (VSAA in Latvian) for business process modeling. Also, lQuery is a demonstration how a transformation language can

be bootstrapped in an existing high-level general purpose programming language (Lua) while being just as expressive as transformation languages developed from scratch.

The scientific significance of the UML-inspired notation and the graphical tool (OWLGrEd) for the ontology language OWL: it is the first notation for OWL that combines the best features of graphical and textual notations in one syntax. That is, the graphics are used for the structural description, and the textual form is used for class expressions. Thus, the resulting notation is both compact and readable.

The practical significance of the OWLGrEd: it facilitates a wider use of OWL among "non-ontologists", thus making it easier to encode their information in machine readable form. The notation has also been used to document the developed ontologies, thus facilitating ontology reuse. As evidence for its usefulness, currently, it is downloaded approximately 100 times each month from its homepage (owlgred.lumii.lv). Furthermore, the existence of the notation makes it possible to adopt OWL as the metamodeling language for the GRAF Tool Building framework.

The scientific significance of the orthogonal extension of the ontology language OWL with the query expressions and semantics from the transformation language lQuery: it shows how the "open world assumption" of the OWL semantics can interoperate with the "closed world assumption" semantics of the transformation language selector expressions. Moreover, the combined language has wider expressive power than each language separately.

The practical significance of the orthogonal extension: its use in the future versions of the GRAF Tool Building framework will enable even more declarative specification of the DSML tools.

1.6 VALIDATION OF THE RESULTS

1.6.1 *Publications on the Topic of the Thesis*

The main results of this thesis have been published in 16 research papers. 12 of the papers are published in editions with recognized citation index (SCOPUS, ACM). The following list contains all of the papers and the level of the author's contribution to each of them:

1. J. Barzdins, A. Zarins, K. Cerans, A. Kalnins, E. Rencis, L. Lace, R. Liepins and A. Sprogis, GrTP: Transformation Based Graphical Tool Building framework, Proc. of MoDELS 2007 Workshop on Model Driven Development of Advanced User Interfaces, MDDAUI 2007; Nashville, TN; USA. (**SCOPUS**)
    - Contribution 15%
2. J. Barzdins, K. Cerans, A. Kalnins, M. Grasmanis, S. Kozlovics, L. Lace, R. Liepins, E. Rencis, A. Sprogis and A. Zarins, Domain Specific Languages for Business Process Management: a Case Study, Proc. of Workshop on Domain-Specific Modeling (Vol. 9, pages 34–40), OOPSLA 2009, Florida, USA.
    - Contribution 15%
3. J. Barzdins, K. Cerans, S. Kozlovics, L. Lace, R. Liepins, E. Rencis, A. Sprogis, A. Zarins, MDE-based Graphical Tool Building Framework, Scientific Papers, University of Latvia, vol. 756, pages 121 – 138, 2010.
    - Contribution 15%
4. J. Barzdins, G. Barzdins, K. Cerans, R. Liepins, A. Sprogis, OWLGrEd: a UML Style Graphical Editor for OWL, Proc. of 1st Workshop on Ontology Repositories and Editors for the Semantic Web, ORES 2010; Hersonissos, Crete; Greece. (**SCOPUS**)
    - Contribution 50%
5. J. Bārzdiņš, G. Bārzdiņš, K. Čerāns, R. Liepiņš, A. Sproğis, OWLGrEd: a UML Style Graphical Notation and Editor for OWL 2,

Proc. of 7th International Workshop on OWL: Experiences and Directions, OWLED 2010; San Francisco, CA; USA. (**SCOPUS**)

- Contribution 50%

6. A.Sproģis, R.Liepiņš, J. Bārzdiņš, K. Čerāns, S. Kozlovičs, L. Lāce, E. Rencis, A. Zariņš, GRAF: a Graphical Tool Building Framework, Proc. of ECMFA 2010, Paris, France.

   - Contribution 15%

7. J. Bārzdiņš, G. Bārzdiņš, K. Čerāns, R. Liepiņš, A.Sproģis, UML Style Graphical Notation and Editor for OWL 2. Lecture Notes in Business Information Processing, Volume 64 LNBIP, 2010, pages 102–114. (**SCOPUS**)

   - Contribution 70%

8. J. Barzdins, K. Cerans, R. Liepins and A. Sprogis, Advanced ontology visualization with OWLGrEd, Proc. of 8th International Workshop on OWL: Experiences and Directions, OWLED 2011; San Francisco, CA; USA. (**SCOPUS**)

   - Contribution 50%

9. R. Liepiņš, lQuery: A Model Query and Transformation Library. In Scientific Papers, University of Latvia, volume 770, pages 27–45, 2011.

   - Contribution 100%

10. R. Liepiņš. Library for model querying – lQuery, Proc. of 12th Workshop on OCL and Textual Modeling, OCL 2012 - Being Part of the ACM/IEEE 15th International Conference on Model Driven Engineering Languages and Systems, MODELS 2012; Innsbruck; Austria. (**SCOPUS**)

    - Contribution 100%

11. R. Liepiņš, K. Čerāns, and A. Sproģis. Visualizing and editing ontology fragments with OWLGrEd, Proc. of 8th International Conference on Semantic Systems; Graz; Austria. (**SCOPUS**)

- Contribution 80%

12. K. Čerāns, R. Liepiņš, J. Ovčiņņikova, and A. Sproğis. Advanced OWL 2.0 ontology visualization in OWLGrEd. Frontiers in Artificial Intelligence and Applications, Volume 249, 2013, pages 41–54. (**SCOPUS**)
    - Contribution 30%
13. K. Čerāns, G. Bārzdiņš, R. Liepiņš, J. Ovčiņņikova, S. Rikačovs, and A. Sproğis. Graphical schema editing for StarDog OWL/RDF databases using OWLGrEd/S, Proc. of OWL: Experiences and Directions Workshop 2012, OWLED 2012; Heraklion, Crete; Greece. (**SCOPUS**)
    - Contribution 20%
14. L. Lāce, R. Liepiņš, and E. Rencis. Architecture and language for semantic reduction of domain-specific models in BPMS. Lecture Notes in Business Information Processing Volume 128 LNBIP, 2012, pages 70–84. (**SCOPUS**)
    - Contribution 40%
15. R. Liepiņš, J. Bārzdiņš, and L. Lāce. OWL orthogonal extension. Lecture Notes in Business Information Processing Volume 128 LNBIP, 2012, pages 13–25. (**SCOPUS**)
    - Contribution 90%
16. R. Liepiņš, M. Grasmanis, and U. Bojars. OWLGrEd ontology visualizer, Proc of ISWC Developers Workshop 2014, ISWC-DEV 2014, Co-Located with the 13th International Semantic Web Conference, ISWC 2014; Riva del Garda; Italy. (**SCOPUS**)
    - Contribution 50%

1.6.2 *Presentations at Scientific Conferences*

The main results of this thesis have been presented at five international conferences:

1. ORES2010, Heraklion, Grece, 31.05 2010, "OWLGrED: a UML Style Graphical Editor for OWL", J. Bārzdiņš, G. Bārzdiņš, K. Čerāns, R. Liepiņš, A. Sproģis

2. OWLED2012, Heraklion, Grece, 27–28 05 2010, "*Graphical Schema Editing for Stardog OWL/RDF Databases using OWLGrEd/S*", K. Čerāns, G. Barzdins, R. Liepiņš, J. Ovčiņņikova, S. Rikačovs and A. Sprogis

3. I-SEMANTICS, September 5–7, 2012, Graz, Austria, "Visualizing and Editing Ontology Fragments with OWLGrEd", R. Liepiņš, K. Čerāns and A. Sprogis

4. BIR2012, September 24–26, 2012, Nizhny Novgorod, Russia, "OWL Orthogonal Extension", R. Liepiņš, J. Barzdins, and L. Lace

5. OCL2012, Sept. 30th - Oct. 5th, 2012, Innsbruck, Austria, "Library for Model Querying – lQuery", R. Liepiņš

1.6.3 *Scientific Projects in Which the Results Where Used*

The main results of the thesis have been developed and applied in the following scientific projects:

- VPP Projekts Nr. 1 „Uz modeļu transformācijām bāzētu sistēmu būves tehnoloģiju izstrāde"

- Projekts VPD1/ERAF/CFLA/05/APK/2.5.1./000009/004: "Jaunas paaudzes sistēmu modelēšanas rīka izstrāde"

- VPP Nr. 2, 5.projekts „Jaunas informācijas tehnoloģijas balstītas uz ontoloģijām un modeļu transformācijām"

- ERAF Nr.2011/0009/2DP/2.1.1.1.0/10/APIA/VIAA/112: „Semantisko datubāzu frameworka nozaru speciālistiem"

- ERAF Nr.2010/0325/2DP/2.1.1.1.0/10/APIA/VIAA/109: „Procesu pārvaldības programmsistēmu būves tehnoloģija un tās atbalsta rīki"

1.6.4 *Applying the Results in Practice*

Industrial DSML tools built by using the GRAF Tool Building framework and the transformation language lQuery:

- OWLGrEd – tool and notation for OWL ontology language;
- ProMod – tool for the Latvian State Social Insurance Agency (VSAA in Latvian), developed at IMCS UL;
- BiLingva – business process management tool, developed by the company "Datorikas Institūts DIVI" (Riga, Latvia).

1.7 SIZE AND STRUCTURE OF THE THESIS

The thesis consists of an introduction, three parts containing eight exposition chapters, and a conclusions chapter. The first part ("Introduction and Theoretical Background") gives an introduction to the problem domain and the context material used throughout the rest of the thesis. The second part ("MDE Based Graphical Tool Building") introduces the GRAF tool building framework and describes a transformation language designed specifically for the task of model interpretation. The third part ("Towards Ontology-Based Graphical Tool Building") presents the motivation and proposal for an ontology-based tool building framework and offers solutions to two fundamental problems for the realization of this.

Part I ("Introduction and Theoretical Background")

Chapter 1 ("General Description of the Thesis") (this chapter) contains the formal description of the thesis.

Chapter 2 ("Theoretical Background") contains the background material used throughout the rest of the thesis. It introduces the ideas of domain-specific languages, tools, and meta-tools. The chapter also provides information about models, model transformations, and an introduction to ontology concepts needed for the third part of the thesis.

Part II ("MDE Based Graphical Tool Building")

Chapter 3 ("Model-Based Tool Building framework GRAF") presents a detailed description of the Tool Building framework GRAF. Although this framework is not one of the main results of the thesis, the description is included for completeness and because the main results all are developed either for it or by using it.

Chapter 4 ("lQuery – a Transformation Language for Tool Building") presents a transformation language for model interpretation–lQuery. It is the author's main contribution to the Tool Building framework GRAF. Its design enables the runtime extension of the tools build with the framework, interoperation with external data sources and environment. Finally, lQuery serves as an example of how to bootstrap a transformation language in any modern general purpose programming language.

Part III ("Towards Ontology-Based Graphical Tool Building")

Chapter 5 ("Why Ontology-Based Tool Building – a Motivating Example") presents an example showing how ontologies can be used in graphical tool building for constraint specification, style definition, and validity checking.

Chapter 6 ("UML-inspired Metamodel and Notation for the Ontology Language OWL") describes the graphical notation and tool for OWL ontologies – OWLGrEd developed by the author. It is based on the familiar UML class diagram notation. It presents the research behind the notation, its semantics and comparison with alternatives.

Chapter 7 ("Extending OWL Ontologies with lQuery-based Constructor Classes") discusses the shortcomings of OWL for the use in closed systems and offers a solution to this problem by combining ontologies with selector languages. Also, a way is shown how this solution can be implemented by using the lQuery in combination with OWL semantic reasoners.

Chapter 8 ("Ontology-Based Tool Building Framework: Architecture Proposal") proposes the architecture for an ontology-based tool building framework.

The **Conclusions** summarizes the results of the thesis and offers some future research directions.

---

2 THEORETICAL BACKGROUND

This chapter presents the background material needed to understand the rest of the thesis. There are three main themes: the Model-Driven Engineering, the graphical Domain Specific Languages, and the Semantic Web technologies.

2.1 DOMAIN SPECIFIC LANGUAGES

Domain Specific Languages (DSLs) are languages that are using terms native to the domain for which they have been developed [58]. The main benefit of such languages is that they significantly decrease the required effort to specify problems and solutions in their domain. DSL can be either textual or graphical. Examples of textual DSLs are regular expressions [39] and database query language SQL [35]. An examples of a graphical DSL is the well-known Business Process Modeling Notation (BPMN) [73]. In this thesis, we will concentrate on the graphical DSLs, and, therefore, from now on, "DSL" means a graphical DSL.

An efficient use of a DSL requires tool support: first, to help with the creation of syntactically valid models; secondly, for model validation, simulation, and compilation. Over the time, a number of frameworks have been developed to help with the creation of tool support for DSLs. Some examples of such frameworks are MetaEdit+ [70], Microsoft DSL Tools [33], Eclipse GMF [42].

2.2 MODEL DRIVEN ENGINEERING

In 2001, OMG (Object Management Group) started a new initiative – Model Driven Architecture (MDA) [53]. The main idea was to look at a system in three different ways: CIM (Computation Independent

Model), PIM (framework Independent Model), and PSM (framework Specific Model). The CIM describes what the system should do, without specifying how it should be done. The PIM describes the abstract algorithm how the system works without any reference to the implementation environment. Finally, the PSM describes how the abstract algorithm is implemented by using a particular technology. The goal of MDA was to obtain an executable software by successive model transformations: a CIM into a PIM and then, the PIM into a PSM. The premise was that the model transformations could be written once, and then reused for multiple systems. Thus all that the system designers would need to do, was to create a high-level description of the system (CIM) and the rest would be generated automatically by use of already implemented transformations.

As part of the MDA initiative, multiple base concepts and technologies were developed. First, the notion of model was formalized, to make it usable for computers. In this thesis, by a model we understand (adapted from A. Kleppe et all [53]) a description of a system written in a language with fixed syntax and semantics, which is suitable for automated interpretation on a computer. Models can also be used to describe other models, in that case, they are called metamodels.

When the term model is formally defined, the next step and fundamental technology is the model repository. It is a system used for storing and working with models that are stored according to the metamodel. Usually, the MOF standard [4] is used as a metamodeling language.

The final part of the MDA concepts are transformation languages in which models transformations (rules how to transform one model into another) could be written. Many languages were developed for this purpose: ATL [48], ATOM3 [56], Fujaba [60], to name a few. Some of the languages were developed at the IMCS LU, such as Lx [26] and MOLA [50]. Notably most of them were developed for the batch tasks of transforming one model into another.

As it turned out, the original vision of the MDA failed to materialize, but the underlying idea of using models for system development

tasks was retained as useful. Now, the extended use of models as primary artifacts for system development, but no longer restricted to the three standard models (CIM, PIM, PSM), is called Model Driven Engineering [62].

2.3 ONTOLOGIES AND REASONERS

The vision of the Semantic Web [22] predicts that in the future the current Web of unstructured documents (human readable web pages) would be augmented with semantic (structured) data. Thus, the web pages would be understandable not only for humans, but also for computers, which could help humans more easily find, share, and combine information available on the web. The origin of the concept was the famous article by Berners-Lee [30] where the vision of how the existing Web could evolve into a Semantic Web was described.

Since the original formulation of the vision, the Semantic Web community has come up with multiple technologies to make the dream a reality. The first building block is the language standard (Resource Description Framework (RDF) [12]) for publishing the data in a structured form. RDF consists of statements. Each statement is a triplet *subject-predicate-object*. Each resource is identified by a uniform resource identifier (URI). On top of RDF another standard is built – the Web Ontology Language (OWL) [7]. OWL is used as a knowledge representation language. OWL has a formal semantics based on description logic [23]. This enables computers to process the knowledge represented in OWL ontologies in an automated way, similar to the deductive reasoning and inference as done by humans. Thus, computers will be able to gather information and conduct research for humans automatically. The programs that can perform the deduction and inference are called (Semantic) Reasoners. Currently, there a number of such programs, most widely used being Pellet [65], HermiT [63], and FaCT++ [71].

Part II

MDE BASED TOOL BUILDING

3 MODEL-BASED TOOL BUILDING FRAMEWORK GRAF

As was mentioned in the introduction, most of the general purpose modeling languages and tools are often not very useful in everyday situations. Being very complicated, they are, of course, very useful for large systems. However, smaller and more specialized systems usually need only a small part of the facilities provided by the universal languages. Therefore, it is better to develop a specialized language for each domain, and that is where the DSLs (Domain Specific Languages) come into play. Although some of the universal languages make advances towards specific tool builders (e.g., UML offers its stereotype mechanism), they can never provide such a broad spectrum of facilities as DSLs. Moreover, buying and adapting some expensive universal language tool for one's small and concrete use-case can often outweigh the expected benefits of using it afterward. On the other hand, the development of a DSL could give little benefit if its implementation would be very expensive. So a simple and unified way of building domain specific languages and tools is needed.

There already exists a class of tools (called meta-case tools) that are designed for building of DSL tools. Some of the most popular are MetaEdit+ [20], Microsoft DSL Tools [33], and Eclipse GMF [1]. However, most of these tools have significant shortcomings. First, they all are generators, i.e., the DSL tool designer can use a wizard to specify the desired DSL tool configuration, and then the meta-case tool generates the executable program. The generation approach is adequate when the DSL notation and behavior is known beforehand, but to use DSLs in novel contexts, it is increasingly the case that the DSL specification is arrived at incrementally, by rapid prototyping. Thus, there is a desire for an interpreted approach, where the tool specification can be changed at runtime. Secondly, because meta-case tools

are generators, they output C# or Java code, which is compiled to an executable DSL tool. For example, in the Eclipse GMF the metamodels for tool definition are provided explicitly: the domain, graphical definition, tooling, and mapping metamodels. The concrete DSL tool is defined by instances of these metamodels. To obtain an executable tool, metamodel instances are converted to Java code. Thus, for the developer to extend the functionality of a DSL tool, she needs to know the object level structure of the whole system. It would be more convenient for the developer if the tool specification and runtime structure would remain in a declarative metamodel structure that could be queried and changed by model transformations.

In 2006, at IMCS LU, the Laboratory of Modeling and Software Technologies started work on a meta-case tool that is based on model-driven engineering principles. The main insight was that a graphical modeling tool at the core is a *graph diagram*, where each graphical element (node, edge) has attached to it a number of textual labels, and for each type of element there is a dialog form for changing the attached labels. Thus, it should be possible to create a metamodel with which one could describe the specific DSL graphical modeling language and the diagrams the user is creating in this DSL. In this context, a model is a set of graph diagrams consisting of elements: nodes and edges. An element in its turn can contain several compartments (textual labels). At runtime, each visual element (diagram, node, edge, compartment) is attached to exactly one type instance (see classes *DiagramType*, *ElementType*, *CompartmentType*) and to exactly one style instance. Here, types can be perceived as an abstract syntax of the model while the concrete syntax is coded via styles. There is also a metamodel (called the *tool definition metamodel*) whose instances represent concrete tools in the above-mentioned model coding. Apart from types, the tool definition metamodel contains several extra classes describing the tool context (classes such as *Palette*, *MainMenu*, *PopUpMenu*, and *ToolBar*).

Now, what would it take to build a tool based on such a metamodel? It takes four things. First, we need components (called *presentation engines*) that can show the contents of the metamodel to

the users. For example, instances of the graph diagram metamodel are displayed as graphs, instances of the dialog form metamodel are shown as user editable forms. The presentation engines must also catch the user actions that he performs on the representations. These actions then need to be encoded in a metamodel instance for handling by transformations. This brings us to the second thing: we need universal transformation for handling of standard user actions, that is, to recognize the user's goal and transform the metamodel instance accordingly. For example, if a user has clicked on a palette element and then on the diagram, it means that the user wants to create an element of a given type in the diagram. The transformation should find the associated element type, create an element instance of that type in the diagram, and notify the presentation engine that the metamodel instances have changed, and it should redraw the view. The universal transformations should offer extension points, where the tool definer can specify some custom logic for event handling. For example, if the DSL tool is a flowchart editor, then there can be only one start symbol in each diagram. Therefore, an extension is needed for the handling of a start element creation event. The extension must check if the condition (only one start element per diagram) is met and notify the user if it is not satisfied. Thirdly, there must be a unified way to add new presentation engines, for associating transformations to presentation engine events, and for the transformations to notify the presentation engines about the changes to the relevant metamodel instances. Finally, we need a new transformation language in which one could write the universal transformations that handle the common event types, and for implementing of extension transformations. Let us look at each part in more detail.

As it turns out, most of the domain-specific modeling language tools require only two presentation engines – graph diagram engine and dialog form engine. The engines are developed in the standard OOP way and read the things that should be shown from the model repository, and for each user event they create a corresponding event instance in the repository with links to the context, and call transformation handler to process the event.

In our approach, the unified connection of presentation engines with transformations is established by using the Transformation Driven Architecture (TDA) [27]. It was developed specifically for the needs of GRAF Tool Building framework. A detailed exposition is available in the thesis of Sergejs Kozlovičš [54]. For us, the crucial part of TDA is the Head Engine. Its role is to provide services for transformations as well as for presentation engines. For instance, when a user event (such as a mouse click) occurs in some interface engine, the *Head Engine* may be asked to call the corresponding transformation for handling of this event. Transformations may issue commands to presentation engines. This is why the *Core Metamodel* contains classes *Event* and *Command*, and the *Head Engine* is used as an event/command manager. The invocation of transformation or presentation engine is done by linking the corresponding event/ command instance to the singleton instance *Execute* of *Head Engine* metamodel. Thus, transformations do not need to know anything about the implementation details of the engines, and correspondingly the engines do not need to know anything about the implementation details of transformations. All they need is the possibility to work with a common model repository.

Universal transformations for the common user actions (such as *create*, *delete*, *copy*, *cut*, *paste*, and *show properties*) are the central part of the tool building framework. They, together with the tool building metamodel, allow to build the core of a new DSL tool by just providing type instances. The tool building metamodel and the universal transformations are explained in detail in the thesis of Artūrs Sproğis [67]. The crucial part needed for this thesis is the general structure of the universal transformations and the extension points. An example of a universal transformation (creation of a new node from the palette) was already mentioned above. Let us now expand it a bit. First, the graph diagram engine detects a click on a palette element followed by a subsequent click on the diagram background. Such sequence of actions signifies the desire to create a new element. Thus, the engine creates an instance of *NewBoxEvent* class, links it to the *PaletteBox* instance that the user clicked and invokes the han-

dling transformation by using the *Head Engine*. The transformation receives the *NewBoxEvent* instance. From it, the transformation needs to find the *PaletteBox* instance and from it – the *ElementType* instance, that the user wants to create. When the transformation has found this context, it proceeds, by creating a *Node* instance in the active diagram, by linking it to the correct *ElementType* instance from the context. The default compartments are also created by using the *CompartmentType* instances linked to the *ElementType* instance. At the end, the transformation notifies the presentation engine of the changes it has made, that is, the transformation creates an instance of the class *UpdateDgrCmd*, links it to the instance of the changed diagram, and invokes the command using the *Head Engine*. As can be seen from this example, the universal transformations are essentially interpreters of user actions. The program for this interpreter is the declarative DSL specification encoded in the type metamodel.

It was crucial for the implementation of such an interpreter that the language in which it was written was appropriate for the task. It needed to support easy navigation and filtering from an event instance to the relevant context. It needed also to support the dynamic loading of extension points (for rapid prototyping that was one of the core requirements for the framework). As it turned out, none of the existing transformation languages was designed with such objectives. Most of them were designed for a batch translation of data from one metamodel to a different metamodel, and not for incremental transformations inside the same metamodel. Thus, a new transformation language was needed. The language will be described in Chapter 4.

Now we will take a more detailed look at each of the components, starting with the graph diagram metamodel and engine.

3.1 GRAPH DIAGRAMMING METAMODEL AND ENGINE

The tool definition metamodel and its interpreter – the tool building framework is based on some basic presentation services whose interface is described by metamodels. One of the essential services is that of graph diagramming. It is defined by means of *graph diagram*

metamodel (GDMM), and it is implemented by a *graph diagram engine* (GDE) [29]. Another service for which we also have a metamodel and a corresponding engine is that of property editors. The property editor metamodel and engine are used in our implementation of the tool building framework. However, they are not of primary importance in explaining its semantics; therefore they are not considered in detail here.

The aim of GDMM is to describe the graph diagramming functionality that can be offered by GDE and that is common to a broad range of graphical diagramming tasks. Since providing appropriate abstractions in GDMM can considerably ease the tool definition process on the basis of GDE, the emphasis in the design of GDMM has been on properly separating concerns between "purely graphical" tasks that are to be handled by the GDE itself and tasks involving "logic" of tools using GDE.

The GDMM (Figure 1) is built around the classes for visual elements of the presentation, namely *GraphDiagram*, *Element*, *Box*, *Line*, and *Port* together with *Compartment* corresponding to text fields placed in boxes and attached to lines and ports (note that line's start and end can be attached to any elements, not just boxes). Instances of these classes will be diagrams and elements created by the user. Every element, compartment and graph diagram has its style (see classes *ElemStyle*, *CompartStyle* and *GraphDiagramStyle*). For every element, the metamodel allows to specify both its default style and local style (the diagramming engine uses the local style if it is defined; otherwise the default style is used). The *Collection* class contains a single item that is linked to the currently active (selected) elements in the diagram. The *seed/child* link between *Element* and *GraphDiagram* allows specifying of an element to be a seed for the diagram therefore providing means for building of diagram hierarchies.

Besides the classes of visual elements, the GDMM also contains classes describing the tool's environment (*Palette*, *Toolbar* and *Keyboard* classes with the corresponding elements). Instances of these classes are typically created at the tool creation time and do not

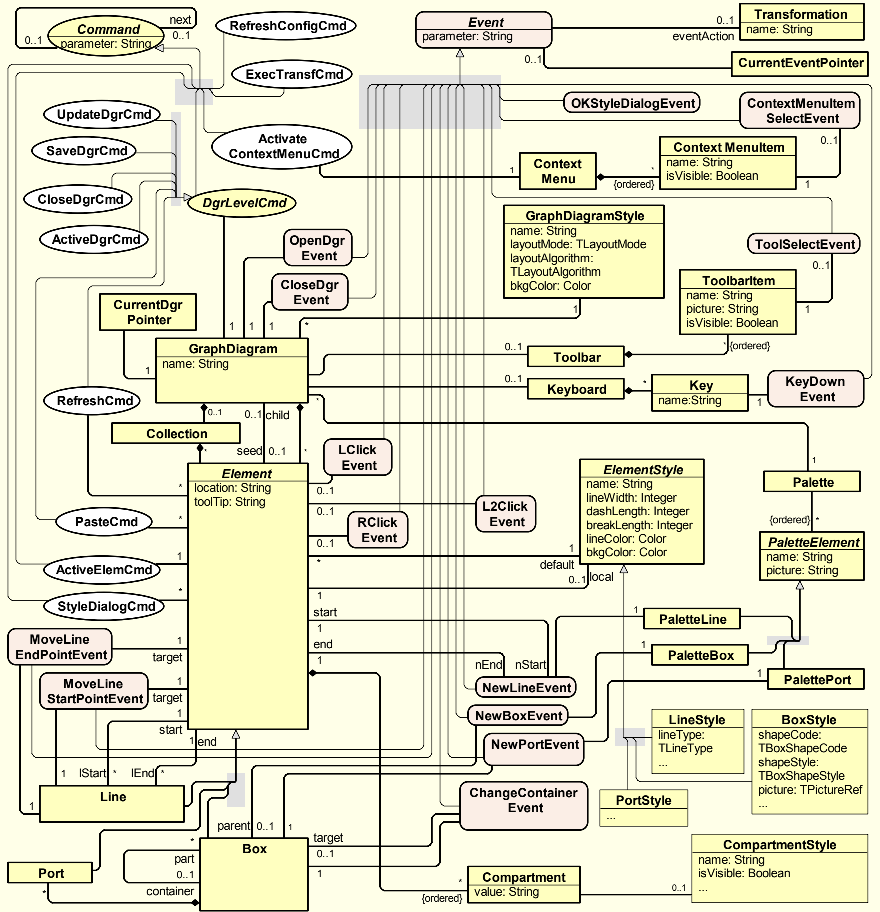


Figure 1: The graph diagramming metamodel

change after that. The context menu (*ContextMenu* class) can also be specified.

GDMM supports the *Event* class whose singleton subclasses correspond to the actions the user may perform on a concrete diagram (the event classes are represented as rounded rectangles), and that are understood by GDE. Upon detecting a current event, GDE invokes the event's *eventAction* transformation that performs the tool's "business logic" in response to this event. The *Command* class describes the requests (commands) that the tool transformations can issue for GDE.

For instance, the creation of a new box in a graph diagram starts by the user making some clicks in the tool, triggering GDE to set *CurrentEventPointer* to the instance of *NewBoxEvent* (the *parent* link from the event is set, if the new box is to be created inside another box). The event's transformation then may, for instance, create a new element of *Box* class (or it may do some extra/other action depending on the tool's specific logic). After that, it creates an instance of *UpdateDgrCmd* and transfers the control back to GDE that processes the command by updating the diagram so that the newly created box becomes visible.

The semantics of other *Command* subclasses is, as follows. The *ActiveDgrCmd* sets editor's focus on the concrete diagram, *RefreshCmd* refreshes the specified elements in the diagram, *PasteCmd* computes coordinates of the elements pasted into the diagram model, *RefreshConfigCmd* rebuilds toolbars and palettes, *ActivateContextMenuCmd* opens the context menu (depending on the collection of elements pointed to by the *Collection* element), *StyleDialogCmd* opens element's style dialogue, *ExecTransfCmd* is used for transformation callbacks.

Although most of the user actions trigger the setting of the current event and invocation of some transformation, there are actions that are performed solely by GDE (e.g. undo/redo, zoom, export to HTML, print diagrams, etc.). The toolbar items responsible for these actions do not have associated *ToolSelectEvent*-s to be triggered when the user selects the item. The context menu item 'Symbol style' is also handled directly by GDE. GDE is also responsible for handling

of element coordinates (the coordinates can be abstracted away while writing the tool defining transformations).

The implementation of GDE has been a considerable programming task of several person-years (A. Zariņš). The relatively simple diagram structure has allowed to implement in GDE advanced graph drawing capabilities [38, 32] that support an automatic initial layout of diagrams as well as serve the interactive diagram editing process. The definition of GDE interface in the form of GDMM allows for reuse of its graph diagramming capabilities in various MDE-related tasks, one of them being meta-tool creation. The architecture of GDE is described in more detail in [29, 27].

3.2 TOOL DEFINITION METAMODEL: THE CORE

In this section, we describe the syntax and semantics of the *core tool definition metamodel* (Core TDMM) that can have (simple) DSML tools as its instances. About 20% of the Core TDMM has been developed by the author. The aim of Core TDMM is to provide basic means for DST definition at the level of graphical presentation. There is a wide range of applications where just the graphical presentation view on the modeled system is sufficient since this is the view of the system directly perceived by the user. The other views on the system, if necessary, can be obtained by model transformations that work either offline by performing export and import tasks, or synchronously, by using tool behavior extension points, described in Section 3.3.

The Core TDMM (Figure 2) is built around the concepts of *GraphDiagramType*, *ElementType* and *CompartmentType* providing type (or, pattern) information for graph diagrams, elements and compartments that are specified in GDMM and that may appear in the concrete tool's visual editor. Therefore, Core TDMM is described as an extension of GDMM. Figure 2 describes both the classes of Core TDMM, as well as a selection of the relevant classes from GDMM in (in Figure 2, these classes are included in two gray rectangles).

The containment hierarchy *Tool GraphDiagramType ElementType CompartmentType* (via *base* link) forms the backbone of TDMM. Every

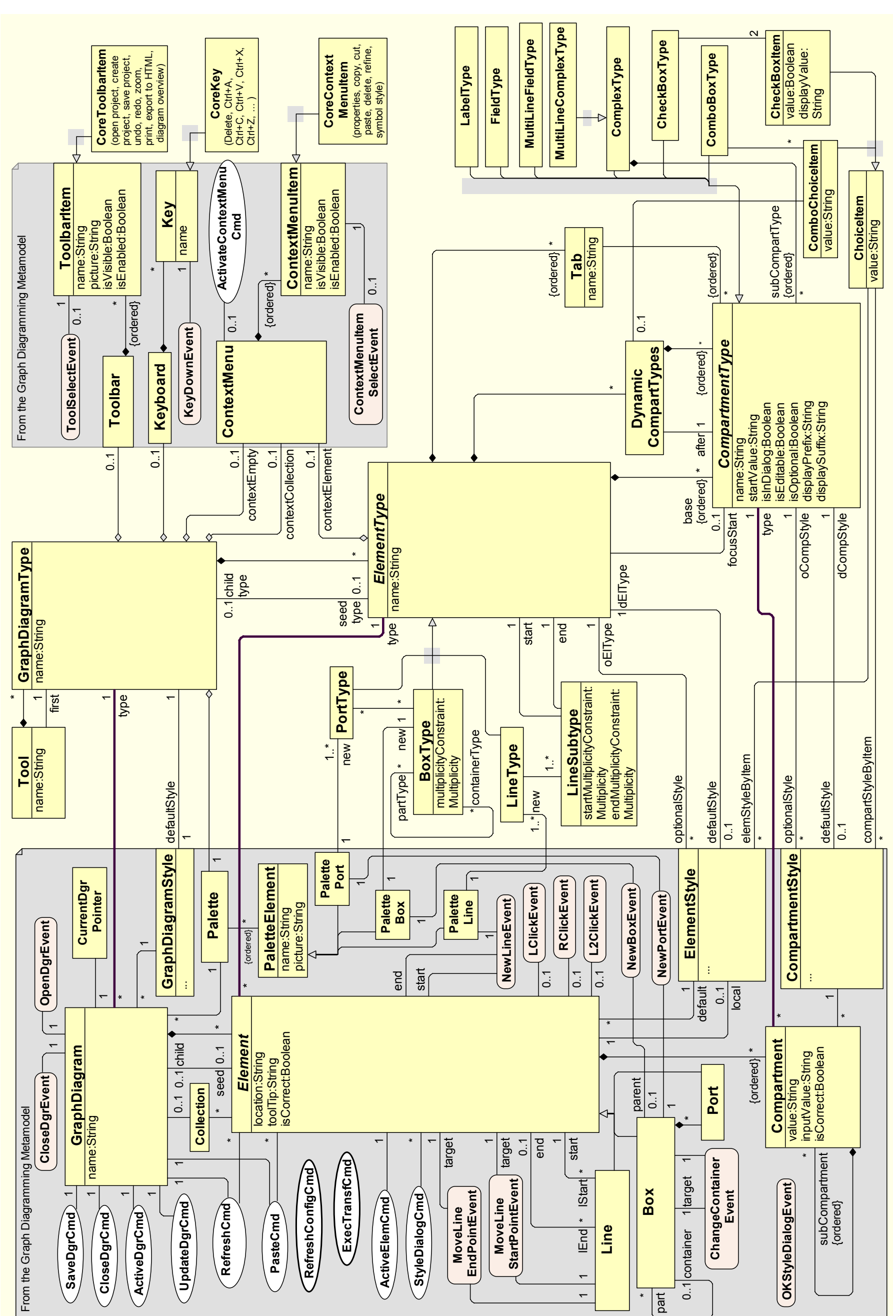


Figure 2: The tool definition metamodel.

tool can serve several graph diagram types (one of these is defined as the *first* diagram type in the tool). Every graph diagram type contains several element types (instances of *ElementType*), each of them being either a box type (e.g., an Action in the activity diagram), a line type (e.g., a Flow), or a port type (e.g., a Pin). Every element type has an ordered collection of *CompartmentType* instances attached via its *base* link. These instances constitute the list of compartment types of the diagram elements of this type.

Note that the correspondence of graph diagram to graph diagram type, element to element type and compartment to compartment type relations is an application of *adaptive object model* [75] patterns to tool definition.

The element type specification (*ElementType* class and its subclasses) allows describing inclusion possibility between boxes of different types (*partType*/*containerType* relation), attachment of ports to boxes, the box type multiplicity constraints (e.g. *0..1* boxes of certain type in a diagram), and line type connectivity rules (pairs of element types for which connection by a line of certain type is allowed as specified by *LineSubtype* class instances).

The *CompartmentType* class is divided into subclasses according to the multiplicity of type's compartments in the elements, as well as the possibilities to work with them in the property editor. Table 1 summarizes these subclasses.

The visualization style of diagrams, elements, and compartments is determined by style instances of GDMM that are connected to the corresponding type instances in TDMM (see Figure 1 for style attributes). Apart from specifying the default style for diagram, element, and compartment types, TDMM allows for the so-called *optional styles* for element and compartment types. These styles can be triggered to become effective for a concrete element/compartment by selection of certain choice item in (possibly another) compartment of *CheckBoxType* or *ComboBoxType* (the links *elemStyleByItem* or *compartStyleByItem* from the *ChoiceItem* to the concrete style instance are used). A classical application of this feature is setting or unsetting

Table 1: Compartment type subclasses

FieldType	Single input field.
MultiLineFieldType	Multi-line input field, with each line corresponding to a compartment. The empty field corresponds to no compartments of this type in the element.
LabelType	Non-editable label. Used, for instance, in property editor to show element names.
CheckBoxType	Check box. The attribute displayValue defines what value will be shown in diagram when the user has selected the corresponding value. For instance, in a class diagram when an attribute is derived (the corresponding check box is selected by activating a CheckBoxItem with the value *true*) it should be displayed in the diagram as `/`.
ComboBoxType	Combo box. The user can choose among certain values that are predefined as ComboChoiceItems.
MultiLineComplexType	Multi-line input field, where each line is a compartment of ComplexType.

the class' name compartment to italics depending on the value of the attribute *isAbstract*. This feature is useful much more widely.

Another form of dynamic behavior supported by Core TDMM is adding of compartments of new types to the elements depending on some value selected in a combo-box. This dynamic behavior is implemented by defining instances of *DynamicCompartTypes* class.

In TDMM, we extend the GDMM *Compartment* class by *inputValue* attribute, so that every compartment has both *inputValue* and *value* attributes. The *value* attribute to be displayed in the diagram is obtained from *inputValue* by prefixing it by compartment type's *displayPrefix* and suffixing by *displaySuffix* (an example of this construction is putting double angle brackets around the stereotype name).

Besides the element and compartment types, every graph diagram type can have an associated *toolbar* consisting of *toolbar elements*. We consider only predefined (core) toolbar elements whose implementation is provided by GDE in the Core TDMM.

The graph diagram type also has an associated *palette* to be shown with concrete diagrams of this type. Each palette element is connected to one or more (in case of ports or lines) element types. This connection determines the type of element being created when a palette element is activated. If several line or port types are connected to one palette element (for instance, in class diagrams it may be convenient to use the same palette element for creation of both associations and links), the type of element is determined by the context of the corresponding *NewLineEvent* or *NewPortEvent* (if there is more than one possible alternative, the list of options is presented to the user).

The context menus (*ContextMenu* instances) can be added to element types, as well as to graph diagram types (there may be different context menus for the same diagram depending on the existence of selected elements in the diagram, therefore there are two associations *contextCollection* and *contextEmpty* from *GraphDiagramType* to *ContextMenu*). In the Core TDMM we consider only items that are implemented by GDE (symbol style), or that are provided by a universal implementation on the level of tool definition framework (properties, copy, cut, paste, delete, refine).

Similarly, we include a *keyboard* with universally implemented keys in the Core TDMM allowing for standard editor functionality (e.g. Ctrl+C for copy, Ctrl+V for paste, etc.), or serving as shortcuts for GDE services (e.g. Ctrl+> for zoom in).

The implementation of the tool definition framework is achieved by developing an interpreter that, relying on the existing implementation of GDE (Section 3.1) interprets a concrete instance of TDMM. This instance specifies the way the corresponding tool has to react from the end user's point of view. As for semantics of Core TDMM and its interpreter, we note that *LClickEvent* does not invoke transformations, *RClickEvent* prepares and opens context menu (via *ActivateContextMenuCmd*) and *L2ClickEvent* opens a property dialogue.

The interpreter also uses the *property dialog engine* (PDE) with a metamodel based interface (the property dialog metamodel, PDMM). This architecture allows to write the interpreter as a collection of model transformations. These transformations are created for all events of GDE, and they handle the "business logic" of the tool that corresponds to the semantics of Core TDMM, outlined here. We have used the model transformation language *lQuery* (Chapter 4) for our implementation; however, other "high-level" transformation languages could have been used for this purpose as well.

An alternative approach to the definition of a concrete tool could be to write the transformations implementing the behavior of the tool directly against the events of GDMM. Then there would remain the option to replace some of the framework-defined transformations by the tool-specific ones (for instance, one may replace 'properties' transformation by 'refine' (navigate from the seed to the child) as a response to *L2ClickEvent* for some element types). Our approach to introducing of tool-specific behavior, however, is via the mechanism of extending universal framework-level transformations instead of replacing them (explained in Section 3.3). In this way the functionality present in the framework-level interpreter is efficiently reused.

As to the expressiveness of the proposed metamodel, a very wide range of graphical tools (int. al., an editor for EMOF [4] class diagrams and UML activity diagrams) can be defined as its instances.

We note that many popular and powerful meta-case tools (see, for instance, MetaEdit [20, 51]) do not offer an explicit tool definition metamodel, but instead explain the tool behavior to the end users by means of some configuration facilities. Some meta-case tools provide the option to use more powerful constraints in some constraint definition language. However, if we want to offer some dynamic behavior, we have to do a serious programming and to understand the implementation of the particular meta-tool very deeply. In our approach, all the relevant information to the DST building and running is captured as an instance of expressive, yet sufficiently simple metamodel (Figure 2), also providing for sufficiently easy means of tool functionality extensions. These extension possibilities are described in the next section.

3.3 TOOL DEFINITION METAMODEL: EXTENSIONS

The implementation of the Core TDMM attaches a fixed universal model transformation to every event of the presentation engine (GDE). However in advanced tool building, there may be situations when such standard universal functionality is not sufficient and certain tool specific behavior is required. For instance, there may be a need to synchronize the "contents" of the graphical editor with data in some other source (e.g., a domain model), or there may be some further restrictions or constraints that must be ensured with respect to elements and values that are introduced during the diagram editing process.

Since the tool to be defined by the tool definition framework conforms to the given tool definition metamodel, it could be in principle possible to allow the tool builder to write his model transformations for handling certain events instead of the transformations built in the framework. However, our approach to the tool functionality extension is more refined in that we allow the tool builder who is willing to introduce the extended functionality still to rely on the basic work done by the transformations implementing the framework. This is achieved by extending the Core TDMM with classes *XElemType* and

XCompartType that are subclasses of *ElemType* and *CompartType* respectively (Figure 3). These classes contain attributes that correspond to certain "call points" at which the framework-level event processing transformation (which is to be adopted to respect these call points) may transfer control to an external tool-specific transformation.

The extended tool definition metamodel also contains classes *AdvancedKey*, *AdvancedContextMenuItem* and *AdvancedToolBarItem* that provide the tool builder with further points where the tool-specific transformations can be attached.

In what follows, we explain the semantics of concrete call points (their placement in the tool interpretation process). We claim that this explanation, together with understanding of the tool definition metamodel is sufficient to use the call point mechanism efficiently in advanced DST building. This is in sharp contrast with the amount of framework specific implementation details that are required for developing advanced tools, for instance, in *Eclipse GMF* framework [19].

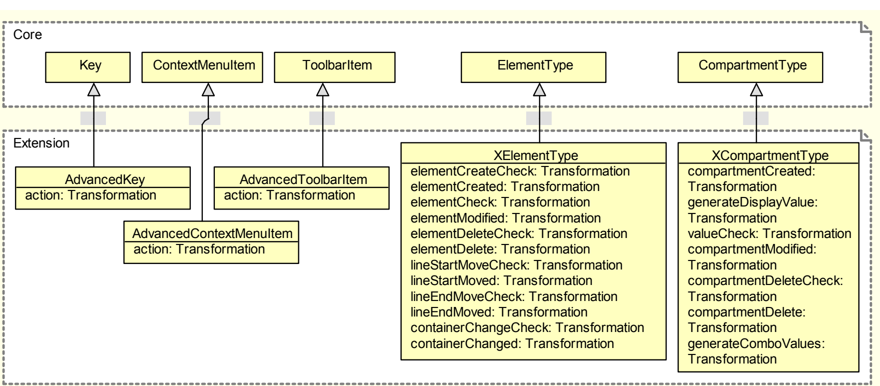


Figure 3: Tool definition metamodel: extensions.

Table 2 summarizes the call points in *XElementType* class that arise in connection with element creation, contents modification and deletion (if not specified otherwise, each transformation accepts the corresponding instance e: *Element* as its only argument; the call points are designed to have transformations that either do or do not have a (Boolean) return value).

Note. Moving of a line start or end, or changing a container do not invoke initial deletion and further creation of elements; therefore

Table 2: Call points from *XElementType*

elementCreateCheck: Boolean	Called before the creation of an element (instance of *Element* class). If the function returns false, the element creation process is cancelled. Recommended for initial correctness constraints (e.g. whether a new element of given type is possible in the diagram).
elementCreated	Called after creation of the element, after *elementCreateCheck*, before the addition of compartments.
elementCheck: Boolean	Called upon completion of value change of element's compartments. The result of the function is recorded in the element's *isCorrect* attribute. The user is notified, if the transformation returns *false*.
elementModified	Called upon completion of value change of element's compartments, after *elementCheck*.
elementDeleteCheck: Boolean	Called upon user's request to delete an element, after system's own checks for delete possibility are completed. If the return value is false, the delete action is cancelled.
elementDelete	Called upon user's request to delete an element, after elementDeleteCheck, before (unconditional) delete of the element.
lineStartMoveCheck(e, OldStart, NewStart: Element): Boolean	Called upon user request to move line's start point, after system's own checks for action possibility are completed. If the procedure returns false, the action is cancelled.
lineStartMoved(e, OldStart, NewStart: Element)	Called after the line start has been moved.

the corresponding call points for element deletion and element and compartment creation are not activated.

Table 3 summarizes the call points in *XCompartmentType* class (each transformation accepts a corresponding instance c:*Compartment* as its argument).

Note. The *compartmentDeleteCheck* and *compartmentDelete* transformations are not invoked when deleting a whole element.

The introduced tool extension mechanism, although simple, is sufficient for a large range of tasks arising in DST building. We mention some of them here:

- synchronization with an abstract user-defined domain model;
- constraints of potentially arbitrary logical complexity;
- support of dynamic contents in the tool (e.g. drop-down values in a combo-box);
- advanced dependencies in tool's presentation behavior;
- integration with other data engines (e.g. data from relational databases, provided the data access interface has been created).

The synchronization of model contents with a user-defined domain model can be performed by transformations *elementCreated*, *elementModified* and *elementDelete*, as well as *compartmentCreated*, *compartmentModified* and *compartmentDelete* that provide the tool builder for the points at which a corresponding action (e.g., creation, modification or delete of a structure, corresponding to an element or compartment at the presentation level) can be defined for the domain model. If necessary, the *lineStartMoved*, *lineEndMoved* and *containerChanged* transformations can be used for this purpose as well.

The constraints can be implemented in the tool via the transformations *elementCreateCheck*, *elementCheck*, *elementDeleteCheck*, *lineStartMoveCheck*, *lineEndMoveCheck*, *containerChangeCheck*, *compartmentDeleteCheck* and *valueCheck*. All of these transformations, except *elementCheck* and *valueCheck*, in the case of returning *false*, cancel the action initiated by the user. The result of *elementCheck* and *valueCheck*

Table 3: Call points from *XCompartmentType*

compartmentCreated	Called after creation of compartment and setting its context (link to the element or containing compartment), before setting up compartment's value and processing sub-compartments.
generateDisplayValue	If specified, is used instead of the Core mechanisms for generating compartment's value (as seen in the diagram) from input value (as entered in the property editor). Called after the input value of the compartment is prepared (e.g. in the property editor).
valueCheck: Boolean	Called upon completion of value change of compartment, after *generateDisplayValue*. The result of the procedure is recorded in the compartment's *isCorrect* attribute. The user is notified, if the transformation returns *false*.
compartmentModified	Called upon completion of value change of compartment, after *valueCheck*.
compartmentDelete	Called upon user's request to delete a compartment, after *compartmentDeleteCheck*, before (unconditional) delete of the compartment.
generateComboValues	Procedure for dynamic generation of values in the compartment's combo box in the property editor. If not specified, the combo box is filled up by means specified in the Core.

transformations is placed in the element's or compartment's attribute *isCorrect*, and the user is notified to take a correcting action if the result was *false*. Note that both the structure of the model created in the editor (the presentation), and the tool-specific domain model information can be accessed by the procedures implementing the constraints.

Since the DST conforms to the (extended) tool definition metamodel, the transformations attached to the call points, as well as the user-defined event-processing transformations (in case of *AdvancedKey*, *AdvancedContextMenuItem* and *AdvancedToolBarItem*) can be in principle defined in any high level model transformation language. This means that we have reached a point where an advanced DST including the user-defined extensions can be fully implemented within MDE framework, without the need to resort to structures and constructs that are typical of traditional programming languages.

Furthermore, the definition of the call points within the tool interpretation process hides the details of this process from the user (it allows the user to seamlessly re-use the implemented process). It allows focusing just on adding the tool-specific advanced functionality and relying on that these will be called at the right time and place. The only requirement for the tool builder (the writer of extension transformations) is not to introduce inconsistencies in the metamodel of Figure 2.

Now we will turn to the language in which the extension transformation can be written.

4

LQUERY – A TRANSFORMATION LANGUAGE FOR TOOL BUILDING

As concluded in the previous chapter, the development of a universal interpreter and the extension point transformations for the GRAF Tool Building framework requires a new kind of transformation language. This language must, among other things, allow dynamic loading of extension transformations at runtime. Moreover, it must be designed for the task of development of incremental transformations that modify a common metamodel (in contrast to existing transformation languages, which are mainly tailored for batch model conversions from one metamodel to another). Also, this language must provide options for integration with other parts of the system (databases, compilers, simulators, etc.) in which the DSL tool is only a component.

The analysis of the types of transformations required for the tool-building framework revealed that the majority of them are context based. Namely, the transformation starts with a single instance element (event, graphical element, form field); afterward it must find the context of the instance, and finally, it must make some adjustments in the instance graph (create or delete an instance; add or remove a property link; modify an instance attribute). Thus, the necessary steps are navigation, filtration and modification of the instance graph. When this conclusion is combined with the need to integrate with the external infrastructure, it seems that it would be very desirable if an existing scripting language could be used as the base for the new language. This would make it possible to reuse the existing libraries and the interpreter of the scripting language for the dynamic loading of extension transformations. The crucial question is: "could an easy to use abstraction layer for writing transformations be developed in an existing programming language"?

The first step should be selection of a scripting language on which the transformation layer could be based. There are many alternatives available: Python [72], Ruby [37], JavaScript [36], and Lua [46]. The scripting language Lua was particularly suitable for our requirements. It is tailored for embedding and extending existing systems, and thus provides a natural fit for writing of extension transformations. It is also one of the smallest and theoretically purest languages [47].

To achieve the expressivity of the existing model transformation languages in a general-purpose scripting language we use the ideas from the functional programming, specifically from combinator parsing [45]. Thus, the result is a set of functions for querying and modifying of the models stored in a model repository. Functions are built in progressive layers, where every next layer is based on the previous one. The first layer is built directly on the repository API. From now on, we will refer to the resulting transformation language as lQuery.

4.1 BRIEF OVERVIEW OF LUA

Before going into details about *lQuery*, we will first give a brief outline of the Lua scripting language and the API of the model repository for which *lQuery* is designed.

Lua is a dynamically typed scripting language, i.e. variables do not have types, but each value carries its type. Comments start with double hyphens ('- -') and run till the end of the line. In the following examples, we use comments beginning with '- ->' to indicate the result of the preceding code.

Lua has only a few primitive value types: *nil*, *strings*, *numbers*, *booleans*, and *functions*. Moreover, there is only one data structure: an associative array, commonly called *table*. The indices and values in a table can be any Lua values: *strings*, *numbers*, *booleans*, *functions*, or other *tables*. Lua has a special syntax for creating tables: *{}* creates an empty table, and *{x=1, y="a"}* creates a table where the index *"x"* has value *1* and the index *"y"* has value *"a"*. There are two syntaxes for getting the value that is associated with a given key in a table: *t.y* and *t["y"]*, the former is just a shorthand for the later:

```
t = {x=1, y="a"}
print(t.x)  --> 1
print(t["y"])   --> "a"
```

Lua has a standard set of control structures: *if* for conditions and *for* for iterations. All control structures have an explicit terminator: *end*.

```
if a < 2 then
    print("a less than 2")
else
    print("a greater than 2")
end

t = {"a", 100, true}
for i, v in ipairs(t) do
    -- i is index, v is value, .. is concatenation operator
    print("the value of index " .. i .. " is " .. v)
end
```

Functions in Lua are first-class values meaning that functions can be constructed at runtime, assigned to variables, passed as arguments and returned as results from other functions. All functions in Lua are anonymous. The statement *function (x) . . . end* is a function constructor, just as *{}* is a table constructor. For example, to create a function that adds one to its argument, we write:

```
add_one = function(n)
    return n + 1
end
add_one(3)  --> 4
```

In the above example, *add_one* is a variable to which we assign the constructed anonymous function.

Tables can also be used as objects. To make it more convenient, there is a special syntax for calling methods: *obj:foo(args)*. It gets the anonymous function stored at key *"foo"* in the table *obj* and calls it

passing the table itself as the first argument. In this case, the table plays the role of *self* or *this* that are used in other object-oriented languages.

4.2 OVERVIEW OF THE METAMODEL

lQuery, like other model transformation languages, works on a model repository. The repository can be divided into two parts (Figure 4): the schema part (upper part of the figure) and the data part (lower part of the figure). The data part is the actual part with which lQuery works, and the schema part serves as annotations that help to understand what each data item means. The schema part consists of three things: *classes*, *attributes*, and *links* (more commonly known as associations but the word "link" reads more naturally in our examples). Classes are used to group objects together, and the *super/sub* relation between classes is used to state that if an object belongs to a subclass then it also belongs to the superclass. Attributes are used as keys for associating string values to objects. Links are used for associating objects with other objects. The data part consists of: *objects*, *attribute values*, and *link assertions*. Objects are the actual values that are stored in the repository. Each object is an instance of some class. Attribute values are strings that are associated with some object with a particular attribute name. Each object can possess at most one attribute value for a particular attribute. Link assertions represent a collection of objects that are associated with a particular object for a given link. An example of a repository contents is provided in the next chapter.

Each schema entity has a unique string id, and there is an API function to get an entity with a particular id. There are also functions to get all objects that belong to a given class, check whether an object belongs to a specific class, create an object, delete an object, get the value of an object attribute, set the value of an object attribute, get linked objects, add link between two objects, and delete link between two objects.

In theory, these functions are sufficient to write any transformation, but in practice the resulting code would be very repetitive, i.e. some

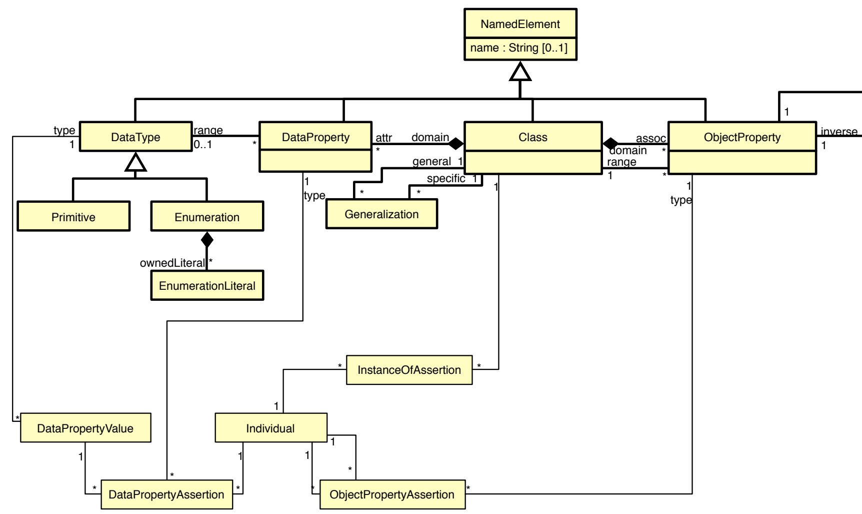


Figure 4: Metamodeling Language Used by lQuery

patterns would repeat, e.g. navigation through multiple link chains, or filtering by some condition. To make transformations more readable, the redundant parts need to be abstracted away. lQuery functions help to do it.

4.3 LQUERY

4.3.1 *Example Model*

In Figure 5 we can see a simple model and an instance diagram. We will use it throughout the rest of the chapter for demonstrating lQuery constructs. The model is on the left; it consists of two classes: *Person* and *Animal*. A person has *name* and *age* attributes and associations to other persons that are his *parents* and *children*, and an association to *Animals* that are his *pets*. On the right side, we can see some instances of this model.

Typical queries that we would like to make on this model are: get instances of a particular class (e.g. all persons), get instances with a particular attribute value (e.g. persons with name "John"), or get all pets of a person's children. If we needed to perform these queries using only the repository API, then the code would mostly contain

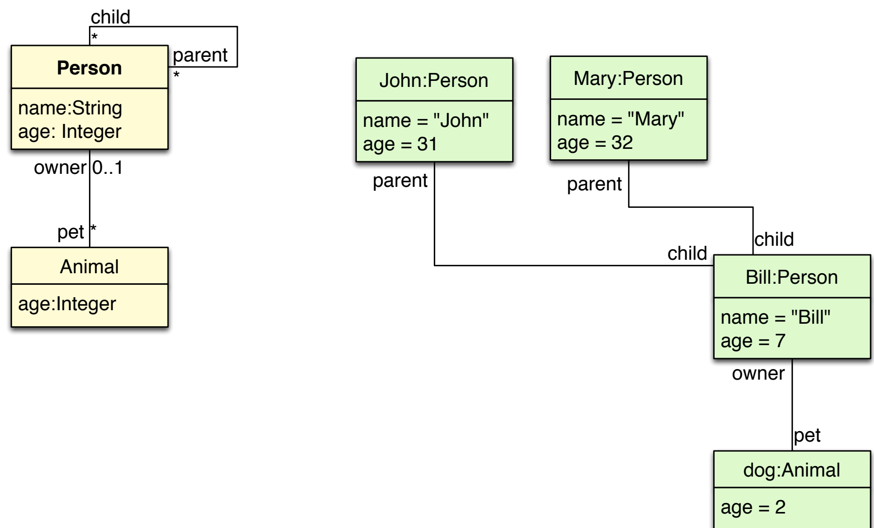


Figure 5: Example model and instances

iterator constructs. For example, to get persons that are 42 years old, we would need to write:

```
persons_with_age_42 = {} --empty table for storing results
for i, o in ipairs(allObjects()) do --iterate over all objects
    --check that object is a person and the value of age is 42
    if isKind(o, "Person") and getAttrVal(o, "age") == 42 then
        --insert person into results table
        insert(persons_with_age_42, o)
    end
end
```

Such a code is far from readable, even for such a simple query, especially if we compare it with path expressions from XPath language [18], where it would look something like ".Person[@age=42]". Our goal is to create a query language where selector expressions would be as compact as that. One way to do it is to create a function that receives an XPath-like selector string and returns the resulting object collection, but this approach is too limiting because there are common queries that cannot be adequately represented as strings (e.g. getting objects with a link to a particular instance, because in our repositories an instance does not have an externally accessible id, so it cannot be encoded in a string). That is why we take another

approach: we define selector functions, and function combinators so that we can easily reference objects and object collections by passing them as arguments to those functions. For the common cases, where string expressions would suffice, we will define an XPath-like selector shorthand notation (string expressions) that can be easily mixed with selector functions and combinators. The result is lQuery.

4.3.2 *lQuery Core*

The core of lQuery is a single function: *query*. It has two arguments: an ordered collection of repository objects and a selector. The selector specifies what will be the result of the query operation on the source collection. There are two types of selectors: *filters* and *navigators*. Filters are used to return a subset of the initial collection based on some condition. Navigators are used for getting a new ordered collection of objects from the initial collection. Examples of filter selectors: filter by class membership, and filter by attribute value. Examples of navigation selectors: get the collection of objects that are reachable from the current collection by a given ling name, and get the collection of values of some attribute. For each of these primitive selectors, there is a constructor function that creates it. Constructor function names have been chosen to improve readability when used as arguments in query calls. The list of built-in primitive selector constructors is given in Table 4. For example, to get all persons from Figure 5 that are 42 years old we can write:

```
persons = query(allObjects(), kind("Person"))
 query(persons, hasAttrValue("age", 42))
```

Such a query is much more concise than the same query written by using the base repository API with an explicit *for* loop (see the previous chapter). However, there are still some problems, e.g. we need to introduce a temporary variable: persons and we have to call the query function twice. It would be better if we could combine these two query steps into one. In that way, we do not have to introduce a temporary variable, and we can call the query function only once.

Table 4: Primitive Selector Constructors

Selector Constructor	Description
`kind(className)`	returns a filter selector that will match only those elements that are instances of a class with id *className* or instances of some class in its subclass chain
`hasAttrValue(attrName, attrValue)`	returns a filter selector that will match only those objects that have an attribute with id *attrName* whose value is equal to *attrValue*
`linked(linkName)`	returns a navigator selector that will match all those objects that are reachable by a link with id *linkName*
`attrValue(attrName)`	returns a navigator selector that will return a collection of values that are associated to attribute with id *attrName*

Another problem is how to perform filters on more complex conditions. Currently, there are only two primitive filters: filter by kind and filter by attribute value. If we need to make a more complex query, e.g. select persons that have at least one child, we have to resort to an explicit iterator.

```
parents = {}
for p in query(allObjects(), kind("Person")) do
    children = query(p, linked("child"))
    if #children > 0 then
        parents:insert(p)
    end
end
```

In the next chapter, we look at selector combinators that address these problems.

4.3.3 *Selector Combinators*

In the previous chapter, we introduced the query function and some primitive selectors for filtering and navigation object collections, but they were not powerful enough to cover many typical use-cases. To solve those problems, we introduce functions (*selector combinators*) for building new selectors from existing ones. They will receive selectors as arguments and return a new selector that can be used elsewhere as if it was a primitive. Let us look at a couple of selector combinators in more detail (the complete list of selector combinators is shown in Table 5).

One of the most frequently used selector combinators is *chain*. It receives any number of selectors as arguments, and returns a new selector that, when evaluated in a query function, will apply the first selector to the initial collection, then pass the result of that evaluation to the next selector and so on through all the selectors that were passed to it. In this way, we can write long selector expressions in a very readable way because we do not need to call query functions separately and to pass them the arguments. For example, to get all

persons and then to get all animals that are pets of those persons, we can write:

```
query(allObjects(), chain( kind("Person"), linked("pet") )
```

Another frequently needed task is filtering not just by a predefined selector (like filter by kind, or filter by attribute value), but by the result of another selector. For this task, there are two selector combinators: *has* and *hasNot*. Selector combinator *has* accepts a selector as an argument and creates a filter selector, that when applied to collection of repository objects will return a new collection with only those objects for which the passed selector returns a *non*-empty collection. The selector combinator *hasNot* works similarly but returns the objects for which the passed selector returns an empty collection. For example, to select persons that have children, we can write:

```
query(allObjects(), chain(kind("Person"), has(linked("child")))
```

Another pair of selector combinators is *union* and *intersect*. Both receive one or more selectors and return a new selector. In the case of *union*, the returned selector returns a union of object collections (multi-set) of all the results of applying each selector to the initial collection. The *intersect* selector returns an intersection of object collection that are returned by all of the passed selectors. For example, let us say a person is *responsible for someone* if that someone is either his child or his pet. To get all persons that are *responsible for someone* we would use a *filter* and *union*:

```
query(allObjects(), chain(kind("Person"),
                          has(union(linked("child"),
                                    linked("pet"))))
```

The selector combinators *chain* and *intersect* can be interchanged in some situations, but in general they are different. When combining selectors with *chain*, each selector will be performed on the result of the previous selector, but when they are combined with *intersect* then all selectors are carried out on the original collection and only then the results are intersected. When all the selectors are *filters* then *chain*

and *intersect* can be interchanged and *chain* is actually a better option, because it is more efficient, i.e. every subsequent selector will be applied to a smaller collection of objects. However, *chain* and *intersect* will return a different result if some of the selectors are *navigators*, because then the *intersect* will perform each selector in the context of a source collection, but the chain will navigate through a chain of links. For example, *intersect(linked("children"), linked("pet"))* will return objects that are children and pets (hopefully an empty collection), while *chain(linked("children"), linked("pets"))* will return children's pets.

The last combinator is *closure*. It receives a selector and returns a new selector that, when applied to a collection of repository objects, will return a new collection with all the objects from the initial collection together with objects that can be found by a repeated application of the passed selector to the resulting collection until no new objects are found. It is impossible to go into an infinite loop here because *closure* will detect cycles and will not evaluate the passed selector on them again. A typical example for *closure* is getting all descendants of a person (here we must assume that each person is a descendant of himself, in the next section we will see how to implement a combinator *closure_plus* that will not have this problem). The *closure* will first find all the person's children then it will find the children of these children, and so on until no more children can be found. It can be written as follows, assuming that *p* is the collection of persons for whom we want to find descendants:

```
query(p, chain(kind("Person"),
               closure(linked("child")))
```

The combinator *closure* can be used not only with simple selectors like *navigation*, but also with more complex selectors: like a chain of links or links followed by filters. For example, if the class *Person* would have the attribute *gender*, then we could create a selector for getting only male descendants by writing:

```
closure(chain(linked("child"),
              hasAttrValue("gender", "Male")))
```

Table 5: Selector Combinators

Selector Combinators	Description
`chain(sel1, sel2, ..., selN)`	creates a selector that applies each of the supplied selectors in order, the first selector is applied to the initial collection, and each subsequent selector is applied to the result of the previous selector
`has(sel)`	creates a selector that filters the initial collection based on the result of supplied selector: if the result is a non-empty collection or a non-false value, then the object will be in the result collection, otherwise it will be dropped
`hasNot(sel)`	creates a selector that returns the complement collection of the result the *sel* selector would have returned
`union(sel1, sel2, ..., selN)`	creates a selector that returns the union of all supplied selector results
`intersect(sel1, sel2, ..., selN)`	creates a selector that returns the intersection of all the selector results
`closure(sel)`	returns a transitive closure by repeatedly applying the selector to the initial collection and then to each of the results until no new object is added

4.3.4 *Selector Reuse and Custom Selector Combinators*

When building any reasonably complex application, there usually are some selector patterns that repeat multiple times, e.g. the compound selector from previous chapter that gets persons that are responsible for someone, i.e. that have a child or a pet. One way to avoid the repetition is to create this selector once and assign it to a variable. Later, when we need to use that selector, we can pass the variable instead of building it from scratch, like this:

```
responsible_persons = chain(kind("Person"),
                            has(union(linked("child"),
                                      linked("pet")))
query(allObjects(), responsible_persons)
```

This works, when the pattern is constant, but what if the pattern is like a template? For example, we could want to get all objects that are reachable via a selector chain with length at least one. We can use functions to create these selectors. In a way, the selector combinators from the previous chapter did just that. For example, to define a new selector combinator (*closure_plus*) that will receive a navigation selector and return a new selector that will match all objects that are reachable via a navigation chain with length at least 1, we write:

```
function closure_plus(selector)
    return chain(selector, closure(selector))
end
```

Now we can use this new selector combinator just as if it was a library primitive. In real life tasks, this allows the programmer to build a task-specific selector library on top of the primitive selectors and selector combinators that are tailored for his problem domain.

4.3.5 *Custom Primitive Selectors*

Although the ability to create higher-level selector combinators is very powerful, it is not enough because we are still bound by the primitives

that came with the library. There are situations when we need a genuinely new kind of selector that cannot be expressed by means of the existing primitives, e.g. get all persons from Figure 5 whose name starts with the letter 'B'. Of course, we could always resort to explicit *for* loops, but then we could not use them in our selector chains, i.e. we would have to split our chains in parts and return to the *for* loops. The situation is even worse, if we want to use such a selection in a combinator (e.g. *closure*), because there is no way to do this, and we would be forced to re-implement *closure* specifically for this case. To alleviate these problems, in lQuery there is a mechanism for constructing new primitive selectors. In fact, all of the primitive selectors of lQuery have been implemented through it.

There are two primitive selector constructor functions (Table 6). The first one operates in the context of a single repository object, like the primitive selectors returned by *linked* and *kind* constructors. The second one operates in the context of a repository object collection. The *closure* selector is implemented through it.

New selectors with single object context can be created by using the function *soloSelector* that accepts a one-argument function as an argument (remember that functions are first-class objects in Lua, and therefore can be passed as arguments). When the resulting selector is used in a query invocation, it will apply the passed function to each element from the initial collection of objects. The function must return either a repository object, an object collection or a boolean. If it returns an object or a collection, then all results are collected in a list that is flattened afterward. If the function returns a boolean, then it acts as a filter, i.e. only those objects for which the function returned *true* are included in the result collection. For example, if we were working with the repository that is shown in the Figure 5 and needed to get all persons who have underage children, then we would have a problem, because there is no selector for checking is an attribute value less than a given integer, and we would have to introduce an explicit *for* loop. However, now we are able to construct such a selector and use it with other combinators:

```
underage = soloSelector(function(p)
```

```
                  age = getAttrValue(p, "age")
                  if age < 18 then
                    return true
                  else
                    return false
                  end
                end)
```

query(allObjects(), chain(kind("Person"), has(chain(linked("child"), underage)))

In fact, all of the primitive selectors are implemented through *soloSelector*. For example, the primitive selector *kind(className)* is implemented like this:

```
function kind(className)
  return soloSelector(function(o)
                        return isKindOf(o, className)
                      end)
end
```

The second primitive selector constructor creates a selector from a one-argument function that will work on all of the initial collection at once. Thus, its only argument is the initial collection of objects. The result of the passed function, when called with the initial collection, is the result of the whole selector. This selector constructor is useful for creation of custom selectors that must have the whole object collection, e.g. getting the first object from a collection, getting the number of objects in a collection, or checking whether an object collection contains a specific object. For example, to get the first child of every person we would first define a new primitive selector *first* (it is universal and can be used in other situations as well) and then use it to get the first child:

```
first = collSelector(function(coll)
                       return coll[1] -- table value by index
                     end)
```

```
query(allObjects(), chain(kind("Person"),
                          chain(linked("child"),
                                first))
```

Table 6: Custom Primitive Selector Constructors

Custom Selector Constructors	Description
`soloSelector(fn)`	creates a selector from a one-argument function; when the selector is used, the function will be applied to each element in the collection; if it returns an object or an object collection, then all the results will be collected and flattened; if it returns a boolean then it will act as a filter
`collSelector(fn)`	creates a selector from a one-argument function, in contrast to *soloSelector*, the whole object collection is passed to the function; the result of the function is the result of the selector

4.3.6 *Shorthand Notation*

The primitive selectors and selector combinators allow us to write complex query expressions in a modular and readable way, but in cases where the selector is constant and simple, the combinator approach yields expressions that are a bit longer than the analogous expressions in OCL [5] or XPath. To reach the maximum compactness and readability, we introduce a shorthand string notation for most common primitive selectors and combinators. The string form can be used anywhere in place of the selector: when the query function gets a string in place of a selector, it will compile it to the corresponding primitive selector constructor or selector combinator calls. This allows

us to mix the shorthand string notation together with ordinary selectors to achieve the maximum of compactness and expressiveness.

The shorthand notation is adapted from the XPath navigation language. Function *compile(shorthand_string)* compiles a shorthand string into the corresponding selectors. It works as follows: a string that starts with a dot followed by an alphanumeric string, e.g. *".ClassName"*, is compiled to the selector constructor *kind("ClassName")*; a string that starts with a slash, e.g. *"/linkName"*, is compiled to *linked("linkName")*, and a string that starts with brackets followed by '@' and a name, e.g. *"[@attrName = value]"*, is compiled to *hasAttrValue("attrName", "value")*. The shorthand notation for selector combinators is as follows: *":has(sel)"* and is compiled to selector combinator *has(compile("sel"))*. The complete list of shorthand notation is given in Table 7 and the selector expression grammar is shown in Figure 6.

Table 7: Selector Shorthand Notation

Shorthand Notation	Equivalent Form
`".ClassName"`	`kind("ClassName")`
`"/linkName"`	`linked("linkName")`
`"[@attrName = value]"`	`hasAttrValue("attrName", "value")`
`":has(sel)"`	`has(compile("sel"))`
`"sel1 sel2 ... selN"`	`chain(compile("sel1"), compile("sel2"), ..., compile("selN"))`
`"sel1, sel2, ..., selN"`	`union(compile("sel1"), compile("sel2"), ..., compile("selN"))`

Let us consider, how some of the examples from the previous chapters can be rewritten by using the shorthand notation. The first example was: get all persons that are 42 years old. Using the shorthand notation we can write:

```
query(allObjects(), ".Person[@age=42]")
```

```
           <lQuery_expr> ::= <object_selector_expr>
   <object_selector_expr> ::= <class_name>
                            | <object_selector_expr> "/" <role_name>
                            | <object_selector_expr> "." <class_name>
                            | <object_selector_expr> "["
                                      <object_selector_expr>
                                      <obj_op>
                                      <object_selector_expr>
                                      "]"
                            | <object_selector_expr> "["
                                      <data_sub_selector_expr>
                                      <data_op>
                                      <constant>
                                      "]"
                            | <object_selector_expr> "not("
                                      <object_sub_selector_expr>
                                      ")"
<object_sub_selector_expr> ::= "/" <role_name>
                            | "." <class_name>
                            | <object_sub_selector_expr>
                              <object_selector_expr>
  <data_sub_selector_expr> ::= <object_sub_selector_expr> ":count()"
                            | <object_sub_selector_expr> ":sum()"
                  <obj_op> ::= "==" | "!="
                 <data_op> ::= "==" | "!=" | "<" | ">" | "=<" | ">="
```

Figure 6: The lQuery selector expression grammar in a BNF notation

The shorthand notation can also be used in selector combinators. For example, to get the descendants of the person collection *p*, we can write:

```
query(p, closure("/child"))
```

In this way, we can use the shorthand where possible, but fall back to selector combinators or custom selectors when the shorthand is not expressive enough.

4.3.7 *Manipulation with Whole Sets of Objects*

The selection of repository objects is only one part of the model interpretation task. The other one is operating with the selected objects. Usually, the operating and the selection is intertwined, i.e. we select some objects, operate with them and then use the resulting collection to find the next collection, and operate with it, etc. Because the repository API supports functions only for manipulating one object at a time, we would have to use explicit iterators for manipulation, and it would break up the *selection-manipulation-selection* chain into multiple

statements. To avoid this problem, we define a number of methods for repository object collections that will allow us to manipulate sets of objects at once and intermix selection and manipulation steps. The list of methods is given in Table 8. We use the Lua object notation, where ':' is used for method invocation. Now let us consider each method in more detail.

There are three manipulation methods: *setFeatures*, *deleteLinks*, and *delete*. The *setFeatures* method receives a Lua table as an argument. Each key in the table is a property (attribute or link) name, and the corresponding value is either a string for an attribute value or an object or an object collection for a link value. The method adds the given features to each object in the source collection. In case of an attribute value, the current value is replaced with the given value. In case of a link, a new link assertion is created for the given object, or for each object in the object collection. For example, to set the attribute "age" of all persons from Figure 5 to 18 and add a link "pets" to some object p, we would write:

```
p = createObject("Animal") -- create a new animal
query(allObjects(), ".Person")
  :setFeatures({
                age = 18,
                pets = p
              })
```

The *deleteLinks* method receives a Lua table as an argument, where each key is a link name, and the corresponding value is either a single repository object or a repository object collection. The method deletes link assertions that correspond to the given key from each object in source collection to the corresponding key value. If there are no link assertions, then nothing is done. The result of this method call is the same collection on which it was called so that further selection or modification operations can be done. For example, to delete the link "child" from all persons in Figure 5 to the person whose name is "Bill", we would write:

```
persons_with_name_bill = query(allObjects(),
```

```
                                  ".Person[@name = Bill]")
query(allObjects(), ".Person")
  :deleteLinks({
                 child = persons_with_name_bill
               })
```

The *delete* method removes all objects that are in the source collection from the repository and returns an empty collection.

There is also a higher-order method *each(fn, args)*, i.e. a method that receives a function as an argument. It can be used to call some function on each object from the source collection for its side-effects, like making some changes in the repository. The result of the method *each* is the same collection on which it was called. This allows us to make multiple calls of this kind one after another. The supplied function *fn* will be called on each object in the source collection: its first argument will be the current object, and the rest arguments will be *args*, which were passed to the *each* method. For example, if we have defined a function for incrementing the attribute "age" by a given number, then we can make every person two years older as follows:

```
function increment_age (person, n)
  current_age = getAttrValue(person, "age")
  setAttrValue(person, "age", current_age + n)
end
 query(allObjects(), ".Person"):each(increment_age, 2)
```

To allow mixing manipulation and selection steps, there is a method *find(selector)* that returns the result of the function query on the given collection and selector, i.e. *p:find(sel)* is equivalent to *query(p, sel)*. This method also creates a selection stack, so that each collection that is a result of the *find* method remembers from which collection it was derived. This information is used by the method *back*, to return the collection from which the current collection was derived. These two methods together with the manipulation methods provide a very readable way to traverse tree-like object structures. To see these methods in action, let us consider a somewhat contrived example: we want

to find all persons in Figure 5, then increment the age of their children by one year and the age of their children's pets by two years, then we want to go back to the children and find a child with the name "Bill", rename him to "Bob", and delete his pets. To perform these actions, in the given order, we can write:

```
allObjects()
  :find(".Person")
     :find("/child")
        :each(increment_age, 1)
        :find("/pet")
           :each(increment_age, 2)
           :back()
        :find("[@name = Bill]")
           :setFeatures({name = "Bob"})
           :find("/pet")
              :delete()
```

Note that *allObjects()* returns an object collection, so we can use the *find* method on it. We use indentation to make the traversal more readable, i.e. after each *find* we increase the indentation to signify that we have a new object collection, and after each *back* call we decrease the indentation to signal that we have returned to the previous collection. Also, note that the result of methods *each* and *setFeatures* is the same collection they were called on (this style of methods is inspired by the so-called *fluent interface* approach to API design).

Although all of the previous examples used the shorthand selector notation in the *find* method, it is by no means the standard situation. In real life tasks, we would use custom selector combinators or predefined patterns because in any complex task we would have built a domain specific selector language on top of the primitives.

4.4 RELATED WORK

Transformation languages are optimized for matching of patterns in the source model and creating of the corresponding patterns in the

Table 8: Object Collection Methods

Object Collection Method	Description
`coll:find(selector)`	returns a new object collection that is the result of applying query *selector* to *coll*
`coll:back()`	returns the collection from which the *coll* was derived
`coll:setFeatures(featureTable)`	*featureTable* is a table where each key corresponds to a feature name and each value corresponds to the new value of the feature; this method sets these values for each object in *coll* and returns the same collection *coll*
`coll:deleteLinks(featureTable)`	*featureTable* is a table where each key corresponds to a link name and value corresponds to the objects to whom the link must be deleted; the method deletes those links and returns the same collection *coll*
`coll:delete()`	deletes all objects that are in *coll* from repository, and returns an empty collection
`coll:each(fn, args)`	for each object in *coll* a function *fn* is called; first argument is the current object and the rest arguments are *args*; returns the same collection *coll*

target model. Because navigation is not the most significant problem in such tasks, transformation languages support either only one-step navigations through link names [50], or navigation expressions that have been inspired by OCL [5], like in the languages ATL [48] and QVT [3]. However, none of these languages treats navigation expressions as first-class values, and thus it is impossible to build or change navigation expressions at runtime or pass them as arguments to other functions. This makes them less usable in situations where the task at hand requires a construct that the language designers did not anticipate. For example, if *lQuery* did not have the *closure* combinator as a built-in primitive, it would be possible to add it as a user-defined function, and use it just as if it were a language primitive. This ability allows a programmer also to define a new higher-level selector language that will be tailored for his domain and thus abstract away the specific details of the metamodel structure. This approach has two advantages: firstly, the code becomes more readable because the selectors are tailored for the problem, and secondly, if the structure of the metamodel changes, we only need to update our domain-specific selectors but all the logic may remain the same because it is built on top of custom selectors.

EMF Model Query [2] is a model query library that is a part of the *Eclipse Modeling Framework* [1]. It treats selectors as objects and can build them at runtime. However, the resulting queries are in the style of SQL, i.e. *select-from-where*, where *from* and *where* clauses accept structures that are similar to *lQuery* selectors. However, we think that XPath-like navigation paths, where navigation and filtering can be intermixed, are more readable.

There are two main limitations to the *lQuery* approach in comparison to other transformation languages. First, the limited support of graph pattern matching, which is highly supported in the mainstream transformation languages MOF QVT [3], Tefkat [57], Viatra [34], GReAT [21], ATL [48], AGG [69], Fujaba [60], UMLX [74], MOLA [50]. The current implementation is not very convenient for specifying graph patterns, i.e. the user needs to introduce explicit variables. Additionally, the user needs to write the graph matching in steps.

However, according to the specific needs of the GRAF Tool Building framework, the necessary context object is always provided; thus this limitation is not a serious obstacle.

The second limitation is the performance penalty because of the interpreted nature of the language. However, it turns out that this is also not a serious limitation, because our goal was to design a language specifically for use cases in graphical tool building, thus our primary performance objective was that the transformation execution time should not be noticeable for the user, when the transformations were handling real-time user actions. This performance goal has been achieved, as was demonstrated by the number of tools developed using *lQuery*, where the transformations are executing without a noticeable delay for the user.

4.5 conclusions

The main result of this chapter is a model transformation language (*lQuery*) that is specifically designed for writing incremental model update transformations. Additionally, the implementation of lQuery is a demonstration how a transformation language for model interpretation can be bootstrapped in any high-level general purpose programming language that supports lambda expressions. The practical usage of *lQuery* has demonstrated that it is easier to use than the transformation language family *Lo* [26] that was used in IMCS UL for tool building prior to the development of *lQuery*.

Part III

TOWARDS ONTOLOGY BASED TOOL BUILDING

5 WHY ONTOLOGY BASED TOOL BUILDING – A MOTIVATING EXAMPLE

In the previous part of the thesis, an MDE-based graphical tool building framework was described that allows to define large parts of the DSL specification in a declarative form. However, there are some widely used components that a tool developer still needs to program manually. Specifically, components that involve complex validity checking, contextual style calculations, and unobtrusive user notification that some parts of the diagram are unfinished. In this part of the thesis, we will explore how these components can be defined using ontologies (in particular, the ontology language OWL [7]). We will also analyze the additional services and extensions that are necessary for the ontologies that will be used as a base metamodeling layer in future versions of the tool building framework.

We will start by looking at an example. Let us suppose that we want to define a simple flowchart editor. The flowchart notation consists of the following types: *flowchart* diagram type, *flow* edge type, and four node symbol types — *start*, *end*, *action*, and *decision*. Even such a simple language as this one must include a number of validity constraints. For example, each *flowchart* diagram must contain exactly one *start* symbol, and exactly one *end* symbol. The *start* symbol must have exactly one outgoing *flow* line, and cannot have any incoming *flow* lines.

It turns out that such validity constraints can be naturally described by using an ontology language. Figure 7 shows how the *flowchart* diagram constraints are expressed in a natural language and by using an OWL class expression. Note that the conditions correspond to the completed state of the diagram. However, most of the conditions are violated at one point or another during the diagram construction process. For example (Figure 8), suppose, we have just started an editing

session and have created an empty *flowchart* diagram. Right away, the conditions requiring that each diagram must contain exactly one *start* symbol, and exactly one *end* symbol are violated. It would be cumbersome for the user if the tool always will notify him of such violations. However, there are some violations that the user should not be allowed to make. For example, if the user tries to draw an outgoing *flow* from a *end* symbol, he should not be allowed to do that. Moreover, he should be notified about it with an explanation of the validity constraint that is violated.

Name	Style	Conditions	Conditions written in OWL
flowchart		Every **flowchart** has exactly one **start** element Every **flowchart** has exactly one **end** element	**FlowChart** *SubClassOf:* **hasElement** *exactly* 1 **Start** **FlowChart** *SubClassOf:* **hasElement** *exactly* 1 **End**
start		Every **start** element does *not* have an incoming **flow** Every **start** element has at least one outgoing **flow**	**Start** *SubClassOf:* *not* **hasIncoming Flow** **Start** *SubClassOf:* **hasOutgoing** *min* 1 **Flow**
action	Action	Every **action** element has exactly one incoming **flow** Every **action** element has exactly one outgoing **flow**	**Action** *SubClassOf:* **hasIncoming** *exactly* 1 **Flow** **Action** *SubClassOf:* **hasOutgoing** *exactly* 1 **Flow**
decision	Decision	Every **decision** element has exactly one incoming **flow** Every **decision** element has at least two outgoing **flows**	**Decision** *SubClassOf:* **hasIncoming** *exactly* 1 **Flow** **Decision** *SubClassOf:* **hasOutgoing** *min* 2 **Flow**
end		Every **end** element has at least one incoming **flow** Every **end** element does *not* have an outgoing **flow**	**End** *SubClassOf:* **hasIncomming** *min* 1 **Flow** **End** *SubClassOf:* *not* **hasOutgoing Flow**
flow		Every **flow** element has a beginning Every **flow** element has an end	**Flow** *SubClassOf:* **hasBeginning** (**Start** *or* **Action** *or* **Decision**) **Flow** *SubClassOf:* **hasEnd** (**Action** *or* **Decision** *or* **End**)
			Element *DisjointUnionOf:* **Node**, **Edge** **Node** *DisjointUnionOf:* **Start**, **End**, **Action**, **Decision**

Figure 7: Flowchart validity constraints written in a natural language and in OWL

It turns out that there is a simple logic determining when to show a violation and when to omit it. A violation can be omitted when it is a consequence of something missing in the diagram, because then the user is most likely in the process of creating the diagram, and will add the missing element eventually. However, if the violation is a consequence of something extra, then it is clearly a mistake, and the system should notify the user about it.

Fortunately, exactly such a behavior is put in the ontology language OWL. OWL is based on the so-called "open world assumption". It assumes that the system has only partial information about the world, and therefore it does not make the conclusion that everything that is not explicitly known is false. As was shown in the previous paragraph, that is exactly the desired behavior in the process of DSL

diagram creation. The tool knows a part of the diagram; the complete diagram exists in the user's mind. Therefore, ontologies are a good candidate for the base metamodeling layer of DSL tool building frameworks.

Moreover, OWL can also be used to explicitly assert that everything that is not known should be assumed to be false. There exists an algorithm that performs this "closure" operation and provides the minimal set of axioms that produce a violation [64]. In this way, the user can check whether the diagram is valid, by declaring to the tool that the diagram is complete (nothing more will be added to the diagram). As an additional service in such cases, the constraints that are not valid, can be reported to the user in natural language sentences [49].

Although the explicit "closure" service is very useful and informative for the user, it still requires an explicit step from the user. He must inform the tool, that he wants to perform validation. It would be desirable to show to the user what diagram elements are incomplete and need further additions. For example, in the *flowchart* diagrams *start* symbols without outgoing *flows* could be displayed with a yellow background, thus drawing the users attention to the fact that something is missing. Such a service cannot be implemented universally because each DSL may require a different way to draw users attention that does not clash with the base notation of the DSL. However, there must be a mechanism how the developer of the tool could specify this kind of behavior.

It would be nice if ontologies could also be used for the specification of the customs styling proposed in the previous paragraph. It could be done by using subclasses with equality expressions, that match the desired elements. Then OWL could be used for styling of the elements by adding style property value constraints. For example, in the *flowchart* diagram metamodel, we can create a subclass of *start* elements, which would contain only those *start* elements that do not have outgoing *flows*. Then we can add a constraint, that these elements must have a yellow background color.

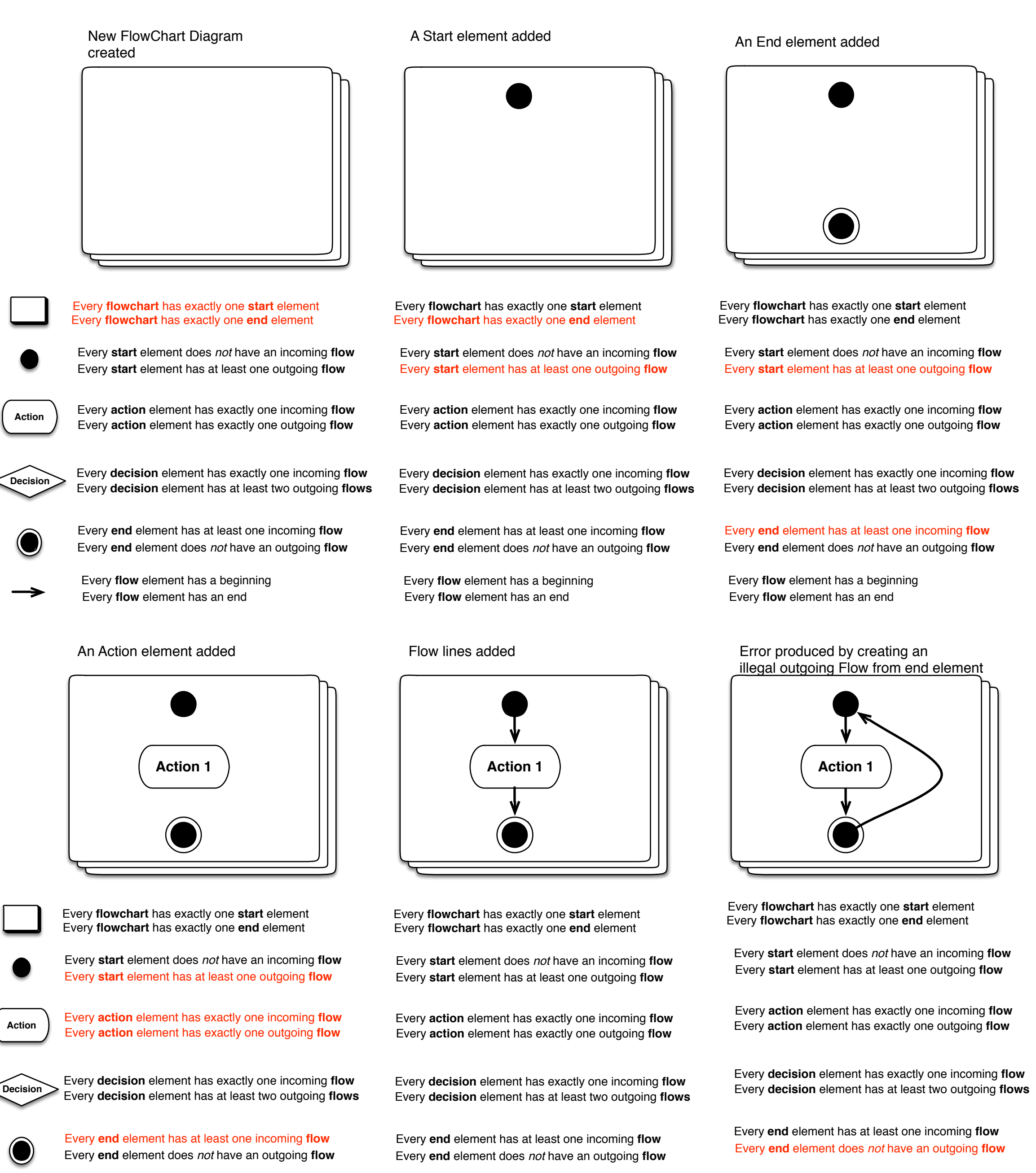


Figure 8: Validity constraint examples during the editing process

However, because of the open world assumption and the monotonic reasoning requirements of ontologies, there are some classes of elements that cannot be described by using OWL expressions. Namely, the elements that are missing something cannot be defined using OWL. For example, there is no way to specify the *start* elements, which **currently** do not have outgoing *flows*. If we would explicitly *close the world*, by asserting that everything that is not known is explicitly asserted as non-existing, but then we would usually run into contradictions with other assertions. For example, in the *flowchart* diagrams, we assert that each *start* element must have at least one outgoing *flow*. Thus, we cannot create a subclass of *start* elements that have no outgoing *flows* because it would contradict the superclass assertion. Consequently, OWL reasoners cannot be used for this task, and we need some other way to specify such classes.

In the following chapters, we will look at how to solve this and other problems to make ontologies a viable option for use a base modeling layer for a graphical tool building framework. First we will solve the problem of combining ontologies with the existing model-based technologies, such as transformation languages, and how to represent them graphically. Then we will explore how to extend ontologies for non-monotonic reasoning tasks. Finally, we will provide a proposal how an ontology-based graphical tool building framework would look.

---

6 UML-INSPIRED METAMODEL AND NOTATION FOR THE OWL ONTOLOGY LANGUAGE

Initially, OWL was defined as an extension of RDF graphs. Therefore, the canonical form for representing of OWL ontologies is a set of *subject-predicate-object* triples. This format is uniform, and this makes representations simple to be parsed and stored by computers, but it is unusable for humans because humans tend to think in terms of higher-level abstractions like classes, instances and relations. However, the actual ontology visualization tools such as IsaViz [61], GrOWL [55], visualize ontologies by showing every RDF triple as two nodes with an edge between them. Thus, the information gets cluttered and spread over a large area, making the structure hard to perceive.

For a graphical form to be useful, it has to group the related concepts together, this approach is used in UML class diagrams. Many concepts of OWL are similar to those of UML class diagrams. There have been attempts to define a UML profile for OWL [31] that would make it possible to use the existing UML tools to create and visualize ontologies. However, OWL has more features than UML class diagrams, e.g. *class expressions*, and *anonymous classes*, which are commonly used, but have unintuitive graphical representations in the UML profile for OWL. Therefore, even though UML profile is better than RDF graphs, it is still hardly comprehensible. Another option is to use Protégé OWL editor [11] that enables to load and save ontologies, edit classes, properties and define class hierarchies. Protégé also provides a detailed view of each concept in the ontology. However, its main shortcoming is the lack of a view that shows the overall structure of the ontology. In the following, we will propose a way allowing to solve this problem.

The next section explains the proposed domain-specific notation for OWL ontologies. After that, we demonstrate a metamodel for OWL that is a layer above the UML class diagram metamodel, which will allow us to merge OWL with the transformation language *lQuery*. Finally, we will describe a graphical editor for the proposed UML-inspired notation.

6.1 OWL AS A DESCRIPTION LAYER ABOVE CLASS DIAGRAMS

Despite the semantic differences between the UML and OWL modeling approaches, UML class diagrams can be used to represent the core features of OWL ontologies – the OWL classes (represented as UML classes), OWL object properties (typically represented as associations in the UML diagram) and OWL datatype properties (typically represented as attributes in the UML class diagrams). Therefore, we will organize our explanation in two steps. The first step is the core part of our notation that is a proper subset of UML class diagram notation [16, 17] (this section); and the second step will cover the extension part that contains OWL [7] specific features (Section 6.2). The explanation is based on the formal metamodel shown in Figure 10. In the context of the current chapter, we will call it the UMLOWLCore metamodel. We use an equivalent encoding (Figure 4) of the core features of the UML class diagram metamodel presented in Chapter 4 for the lQuery language. This decision was made to enable the interoperation of OWL with *lQuery* that will be presented in Chapter 7.

The UMLOWLCore (the bright yellow boxes in Figure 10) includes only those UML class diagram features, which have direct one-to-one equivalents in OWL. For example, n-ary associations are not included in UMLOWLCore because their reduction to OWL requires the introduction of multiple intermediate classes and properties [6, Chapter 16]. Figure 9 shows an example of "mini-university" ontology in our proposed UML notation, as well as its textual form of OWL Functional Syntax [9] notation. This example uses only UMLOWLCore.

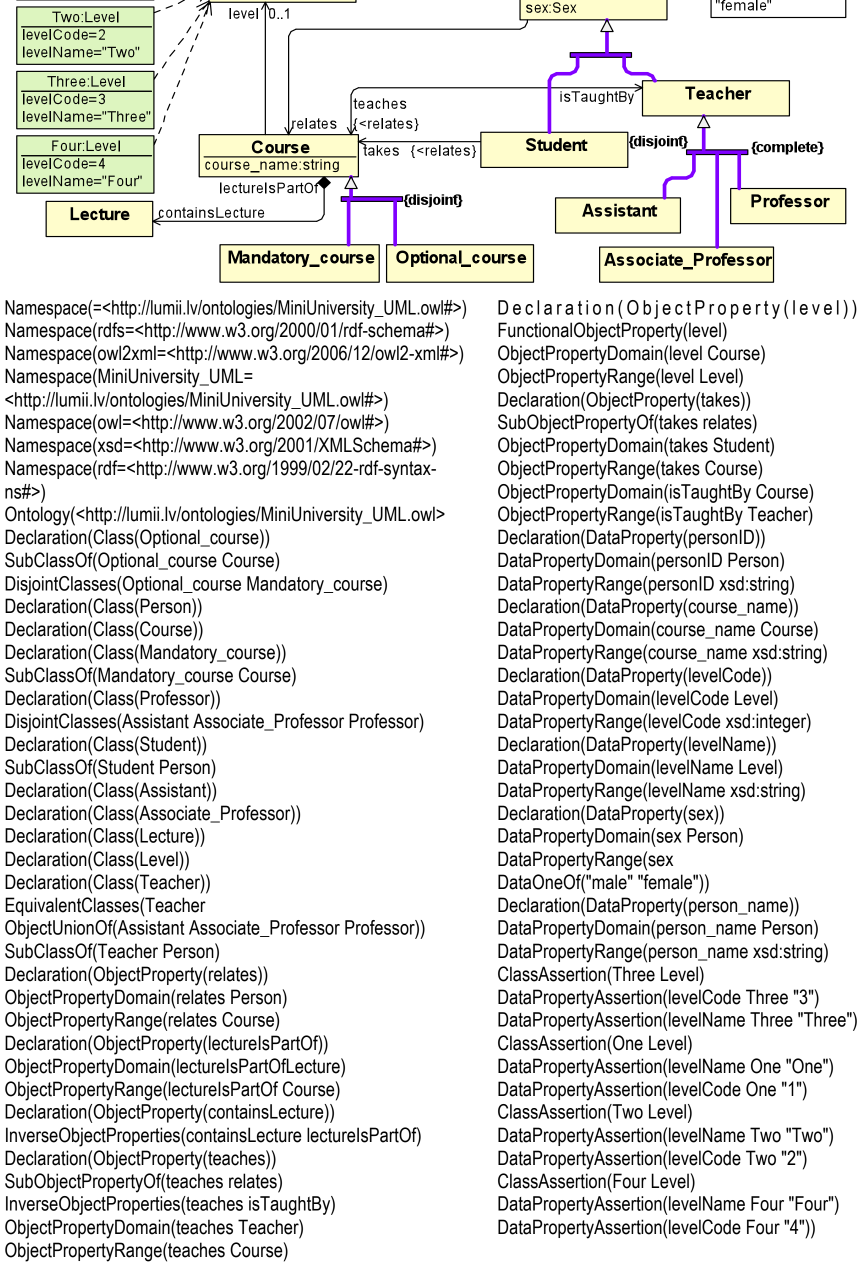
Here we define the details of mappings between the core OWL structures and UML class diagrams. UML classes denote OWL classes while UML properties denote OWL object properties and OWL datatype properties. Typically, the UML properties that are represented as associations denote OWL object properties and the UML properties that are depicted in attribute notation denote OWL datatype properties. However, other combinations of UML properties used for denoting OWL properties are also allowed.

The UML *Generalization* is used to denote the *subclassOf* relation in OWL. We note that it is possible to use *complete* and *disjoint* tags with UML generalizations, and these have a well defined semantics in OWL. For a UML generalization set comprising *subClassOf(B,A)*, *subClassOf(C,A)* and *subClassOf(D,A)* relations, a *disjoint* tag would add the OWL axiom *disjointClasses(B,C,D)* and a *complete* tag would add the OWL axiom *subClassOf(A,unionOf(B,C,D))*.

We allow the use of aggregation in the OWL ontology diagram representation (e.g. *containsLecture* and *lectureIsPartOf* properties in Figure 9). Currently the aggregation symbol is treated as a regular OWL property and is supported in the diagram editor for the sake of structuring and readability of the graphical model only; however, in the future it would be preferable to assign to the aggregation symbol the formal OWL semantics.

OWL individuals are included in the specification of UML class diagrams; their concrete property values are denoted by UML slots and their corresponding value specifications.

Some changes to the semantics of the UML notation are unavoidable because OWL relies on the *open-world* assumption whereas UML relies on the *closed-world* assumption. To satisfy the needs of OWL by using the UML notation, we have changed the default values of some UML constructions. First, in UML, the default cardinality for class attributes is 1 and the default cardinality for association domain and range is "*". We have changed them to "*" in both cases. Secondly, the scope of a class attribute in UML is the corresponding class but in our notation the scope is changed to the entire ontology. Thus, if the same attribute names occurs in multiple classes, then it is inter-




```
Namespace(=<http://lumii.lv/ontologies/MiniUniversity_UML.owl#>)
Namespace(rdfs=<http://www.w3.org/2000/01/rdf-schema#>)
Namespace(owl2xml=<http://www.w3.org/2006/12/owl2-xml#>)
Namespace(MiniUniversity_UML=
<http://lumii.lv/ontologies/MiniUniversity_UML.owl#>)
Namespace(owl=<http://www.w3.org/2002/07/owl#>)
Namespace(xsd=<http://www.w3.org/2001/XMLSchema#>)
Namespace(rdf=<http://www.w3.org/1999/02/22-rdf-syntax-
ns#>)
Ontology(<http://lumii.lv/ontologies/MiniUniversity_UML.owl>
Declaration(Class(Optional_course))
SubClassOf(Optional_course Course)
DisjointClasses(Optional_course Mandatory_course)
Declaration(Class(Person))
Declaration(Class(Course))
Declaration(Class(Mandatory_course))
SubClassOf(Mandatory_course Course)
Declaration(Class(Professor))
DisjointClasses(Assistant Associate_Professor Professor)
Declaration(Class(Student))
SubClassOf(Student Person)
Declaration(Class(Assistant))
Declaration(Class(Associate_Professor))
Declaration(Class(Lecture))
Declaration(Class(Level))
Declaration(Class(Teacher))
EquivalentClasses(Teacher
ObjectUnionOf(Assistant Associate_Professor Professor))
SubClassOf(Teacher Person)
Declaration(ObjectProperty(relates))
ObjectPropertyDomain(relates Person)
ObjectPropertyRange(relates Course)
Declaration(ObjectProperty(lectureIsPartOf))
ObjectPropertyDomain(lectureIsPartOfLecture)
ObjectPropertyRange(lectureIsPartOf Course)
Declaration(ObjectProperty(containsLecture))
InverseObjectProperties(containsLecture lectureIsPartOf)
Declaration(ObjectProperty(teaches))
SubObjectPropertyOf(teaches relates)
InverseObjectProperties(teaches isTaughtBy)
ObjectPropertyDomain(teaches Teacher)
ObjectPropertyRange(teaches Course)
Declaration(ObjectProperty(level))
FunctionalObjectProperty(level)
ObjectPropertyDomain(level Course)
ObjectPropertyRange(level Level)
Declaration(ObjectProperty(takes))
SubObjectPropertyOf(takes relates)
ObjectPropertyDomain(takes Student)
ObjectPropertyRange(takes Course)
ObjectPropertyDomain(isTaughtBy Course)
ObjectPropertyRange(isTaughtBy Teacher)
Declaration(DataProperty(personID))
DataPropertyDomain(personID Person)
DataPropertyRange(personID xsd:string)
Declaration(DataProperty(course_name))
DataPropertyDomain(course_name Course)
DataPropertyRange(course_name xsd:string)
Declaration(DataProperty(levelCode))
DataPropertyDomain(levelCode Level)
DataPropertyRange(levelCode xsd:integer)
Declaration(DataProperty(levelName))
DataPropertyDomain(levelName Level)
DataPropertyRange(levelName xsd:string)
Declaration(DataProperty(sex))
DataPropertyDomain(sex Person)
DataPropertyRange(sex
DataOneOf("male" "female"))
Declaration(DataProperty(person_name))
DataPropertyDomain(person_name Person)
DataPropertyRange(person_name xsd:string)
ClassAssertion(Three Level)
DataPropertyAssertion(levelCode Three "3")
DataPropertyAssertion(levelName Three "Three")
ClassAssertion(One Level)
DataPropertyAssertion(levelName One "One")
DataPropertyAssertion(levelCode One "1")
ClassAssertion(Two Level)
DataPropertyAssertion(levelName Two "Two")
DataPropertyAssertion(levelCode Two "2")
ClassAssertion(Four Level)
DataPropertyAssertion(levelName Four "Four")
DataPropertyAssertion(levelCode Four "4"))
```

Figure 9: "mini-university" ontology (UML notation and OWL Functional Syntax)

preted as the same OWL property; its domain is the intersection of the corresponding classes, and its range is the intersection of its data types.

6.2 EXTENSION OF THE CORE METAMODEL

The UMLOWLCore metamodel is sufficient only for denoting a part of OWL constructs. Figure 10 shows UMLOWLCoreExtended metamodel that is an extension of UML metamodel, and that is used as a basis for our OWL notation. The UMLOWLCoreExtended metamodel extends the UMLOWLCore metamodel with constructs that enable a convenient combination of graphical and textual rendering facilities for almost all OWL 2.0 constructs.

Figure 11 shows an illustration of the use of our UML-based OWL notation on the "mini-university" ontology example.

In what follows, we describe the details of our proposed OWL notation for rendering and editing of OWL ontologies. Note that, as it is common in UML class diagrams, in some cases we allow alternative graphical and/or textual notations for the same OWL construct. Alternative notations enable the user to tune the look of the diagram to his taste, as well as to choose the rendering option that is most suitable to the size and structure of the particular ontology.

6.2.1 *Equivalent and Disjoint Classes and Properties*

The simplest extensions to the UML metamodel are equivalent classes and disjoint classes which are introduced in the extended UML by *equal* and *disjoint* relations from UML *Class* to UML *Class* and OWL *ClassExpression* class. The equivalent and disjoint properties are introduced by *equal* and *disjoint* relations between the *Properties* superclass of classes *DataProperty* and *ObjectProperty*.

The OWL class equivalence is modeled by the *equal* relation between *Class*, and it can be visually represented in the diagram in two ways – either as a connector with «*equivalent*» stereotype linking two

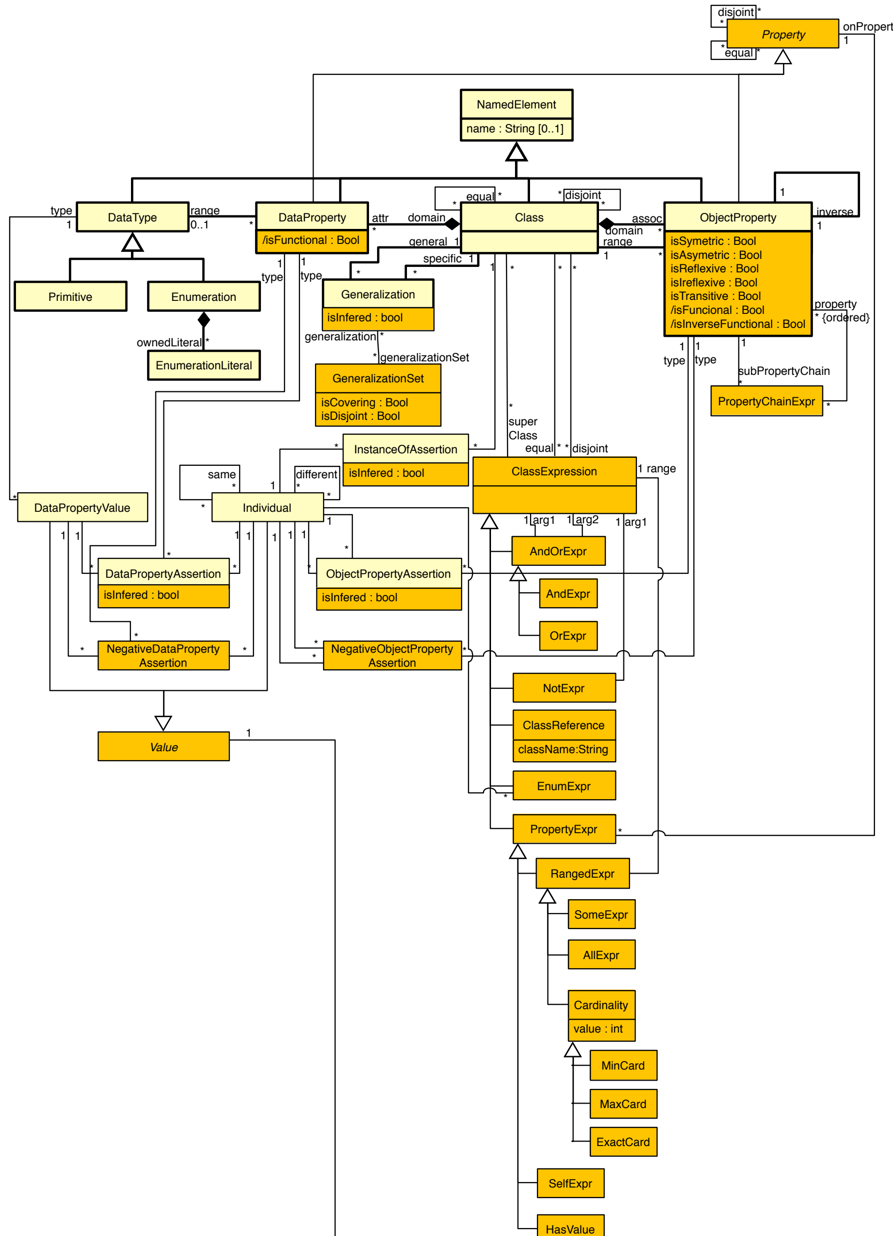


Figure 10: UMLOWLCoreExtended metamodel; classes in bright yellow are UMLOWLCore metamodel (equivalent to metamodel shown in Figure 4)

classes, or as a note symbol with *«equivalent»* stereotype connected to the equivalent classes. The OWL class disjointness is modeled by the *disjoint* relation, and it can be visually represented in the diagram either as a connector with *«disjoint»* stereotype linking two classes, or as a note symbol with *«disjoint»* stereotype connected to the disjoint classes (see Figure 11 where the disjointness of *Person*, *Level* and *Course* classes is asserted). There are also other options for denoting the class equivalence and disjointness using the notion of *class expression* that will be explained later.

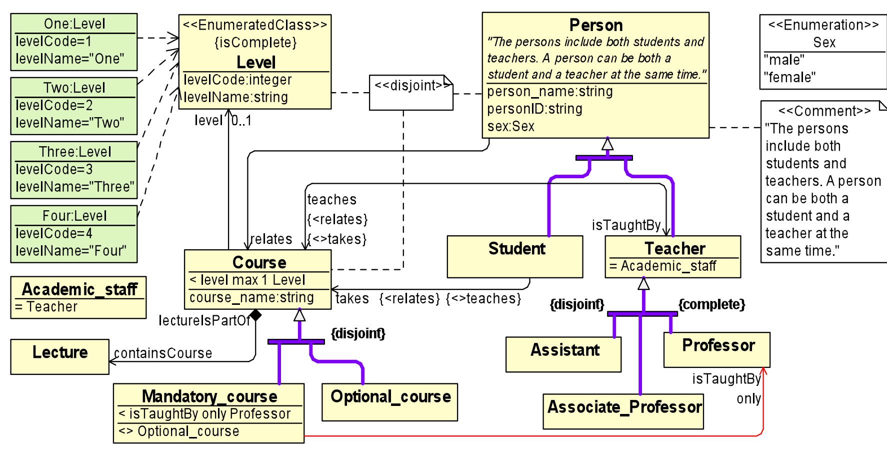


Figure 11: "mini-university" ontology (UMLOWLCoreExtended notation)

The OWL property equivalence is modeled by the *equal* relation between *Property* class, and it is represented by an equivalent property compartment in the property visualization. For example, to assert that the property *p1* is equivalent to the property *p2*, we add a *{= p2}* compartment to the *p1* visualization (we may add a *{= p1}* compartment to the *p2* visualization, as well). A similar notation, using a <> symbol instead of = represents OWL property disjointness being modeled by *disjoint* relation between *Property* class. For instance, in Figure 11 the properties *teaches* and *takes* linking the classes *Teacher* and *Course* are disjoint.

6.2.2 *Class Expressions*

The primary source of OWL's expressive power is its ability to form *class expressions* by means of boolean expressions (*and*, *or*, *not*) out of the declared classes (referenced by the class name) and property-based constraints. We have extended the basic UML metamodel with the class *ClassExpression* and introduced the *ClassReference* class whose instances allow considering UML classes as *class expressions*.

In the OWLGrEd notation, *class expressions* are usually shown by using the OWL Manchester syntax [10], while the traditional UML class representation is used for *named classes*. We include in the metamodel direct means of stating that a class is *a subclass of*, *equivalent to*, or *disjoint with* a class expression (the *superClass*, *equal* and *disjoint* relations from *Class* to *ClassExpression*), and we provide designated textual compartments in the class symbols where to show the class expressions that are related to the class via these relations. For instance, in Figure 11 the class *Mandatory_course* is a subclass of the expression *isTaughtBy only Professor* by using the compartment prefixed with '<' symbol . In this case there is a *ClassReference* instance pointing to the class *Professor*, and it is *range* to an *AllExpr* object that has *isTaughtBy* as its *onProperty* value. The compartments for *equal* (an equivalent class expression) and *disjoint* (a disjoint class expression) are prefixed with '=' and '<>' respectively.

There is also an alternative form of denoting the fact that a class *c* is a subclass of a class expression *'some values from'* or *'all values from'*, namely, a red line leading from *c* to the class that corresponds to the *classReference* that is *range* of the expression; the constraint line is labeled by the name of the expression's *onProperty* property and the word *'only'* or *'some'* (*'only'* corresponds to an *'all values from'* and *'some'* to *'some values from'* constraint). In Figure 11 the red line from *Mandatory_course* to *Professor* shows an alternative form of denoting the *subClassOf(Mandatory_course, isTaughtBy only Professor)* constraint.

The OWL cardinality constraints *subClassOf(c, p card n cc)* (*c* is a class, *cc* is a class expression, *card* ∈ *{min, max, exactly}*, *n* is a nonnegative integer, and *p* is a property) can be denoted either by using the

Manchester syntax (i.e. by adding a $<$ *p card n cc* compartment to the class *c*); or by the UML style cardinality notation either on the line corresponding to the property *p*, if the domain of *p* is *c* and the range of *p* is the class that is referenced by *cc*, or on a red constraint line that originates from *c* and goes to *cc*.

6.2.3 *Anonymous Classes*

There may be a need to assert *superclass*, *equivalent class* or *disjoint class* relations not only between two named classes, or between a class and a *class expression*, but also between two *class expressions*. It may also be necessary to set *class expression* as the domain or range of a property. In the proposed metamodel, these situations are modeled by introducing anonymous (non-named) classes that are defined to be equivalent (via the *equal* relation) to the respective *class expressions*. In the graphical notation, such anonymous classes are depicted like classes, with the only difference that these classes have no name (the name compartment is empty).

Note that a similar notation for anonymous classes is present in UML/OWL profile [6, 52], however, in our approach anonymous classes are introduced only when they are a domain or range of some property. When they are a superclass, disjoint class, or equivalent class of a named class, they can be shown in a textual form in a compartment of the named class. This allows us to achieve more compact and readable diagrams in most cases.

6.2.4 *Enumerated Classes*

In our extended UML metamodel, the enumeration expression is represented with the *EnumExpr* class. The enumeration expressions correspond to OWL *ObjectOneOf* construction. In the graphical notation, OWL Manchester notation can be used to represent the enumeration expressions (e.g., a class can have a compartment *={A,B,C}*, where *A*, *B* and *C* are instance names).

Our enumerated class construct is similar to the one in [52], and it provides an alternative notation for the OWL Manchester notation of the enumeration expressions. The enumerated class in our extended UML metamodel is a (named) class *c* that is equivalent (via *equal* link) to some enumeration expression *e* (for instance, *e* may be the expression *{A,B,C}*). Notationally we add a stereotype *«Enumerated-Class»* together with a tagged value *isComplete* to *c*. In this case, the enumeration expression *e* does not need to appear explicitly in the OWL ontology presentation. However, the class *c* is assumed to be equivalent to the enumeration of all its instances being present in the diagram (and that are denoted either by *instanceOf* links to *c*, or by explicit specification of *c* as instance's type in the compartment of the instance's rectangle). The enumeration expression *e*, in this case, is implicitly represented in the diagram, and it will be restored explicitly when exporting the diagram to the OWL notation. For example, in Figure 11 the class *Level* is defined to be an enumeration class and it is defined to be equivalent to the enumeration expression *{One, Two, Three, Four}*.

6.2.5 *Further Metamodel Extensions*

The UMLOWLCoreExtended metamodel provides means for introducing *symmetric*, *asymmetric*, *reflexive*, *irreflexive* and *transitive* characterizations for *object properties* in OWLGrEd notation. Graphically, these characteristics are represented in the textual compartments next to the line representing the property. In the metamodel, these characteristics are available as attributes of classes *DataProperty* and *Object-Property*.

We also note the possibilities to add the *same* and *different* specifications to OWL individuals in the UMLOWLCore metamodel. At the graphical notation level these options are available both in binary specification form by offering lines with stereotypes *«sameAs»* and *«different»*, and in *n-ary* specification form by offering *note* boxes with the same stereotypes and connecting them to the corresponding instances.

As for instance level negative property assertions, we introduce classes *NegativeDataProeprtyAssertion* and *NegativeObjectPropertyAssertion* into the UMLOWLCoreExtended metamodel. In graphical notation, these specifications are similar to the ordinary ("positive") data property assertions, with = replaced by <> (e.g. $x <> 5$ instead of $x = 5$).

To model the data in OWL ontologies, we extend the spectrum of available primitive data types, as well as we classify the literal specifications by their corresponding primitive data types.

We provide in the UMLOWLCoreExtended metamodel also the means for introducing annotations and attaching those to classes, properties and class instance specifications. The classes annotations can be depicted visually either in specifically designated textual compartments or by respective stereotyped note symbols that are connected to the class symbols that are being annotated.

The UMLOWLCoreExtended metamodel also covers advanced OWL 2 features such as *keyProperty*, *PropertyChainExpr*, and *SelfExpr*.

6.3 THE EDITOR FOR THE PROPOSED NOTATION – OWLGRED

To make the notation usable in practice, we have built a graphical OWL editor (OWLGrEd) which contains many additional services to ease ontology development and exploration, e.g. different layout algorithms for automatic ontology visualization, search facilities, zooming, graphical refactoring and interoperability with Protégé. The editor is built by using the GRAF Tool Building framework described in Chapter 3. Figure 12 shows the "African Wildlife" ontology [22, pp 133–136] as visualized in OWLGrEd.

Graphical refactoring is one of the most important services that allows modifying the graphical notation without changing semantics as long as the same concept can be expressed through different constructs. This feature enables the user to choose the most compact graphical format depending on the context and taste. One example that illustrates the need for graphical refactoring is related to mutually disjoint subclasses: if a class has subclasses that are mutually dis-

joint, then it is preferable to group the subclass relations visually with a "fork" symbol that possesses the *disjoint* label. This is a much more compact representation than the alternative notation where each subclass line is by itself, and there are explicit *disjoint* labeled edges between all subclasses. By using the visual refactoring, the graphical reorganization can be done with one click.

Automatic layout and search facilities are crucial when ontologies become large (more than 100s of classes), and their management becomes more difficult. A good automatic layout is significant for understanding large ontologies. Also, searching for the specific element in large ontologies may become irritating without an appropriate service. Therefore several alternative automatic layout modes and searching mechanism allowing finding the necessary element by the value of one of its text fields (e.g. searching a class by its name) is supported in our editor.

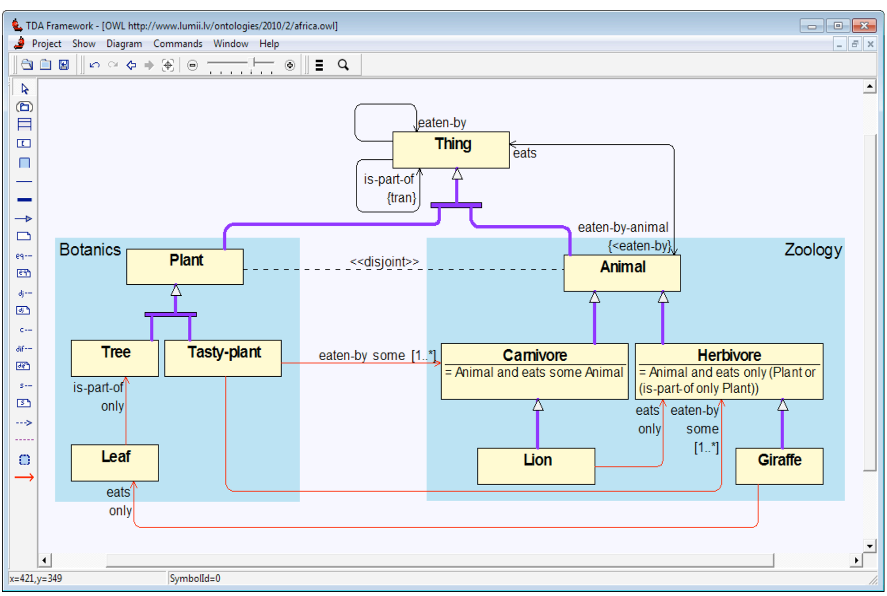


Figure 12: An African Wildlife ontology in OWLGrEd editor

A more advanced service is full interoperability with Protégé [11], a tool that is widely used by ontology developers. The interoperability is implemented via a custom Protégé plug-in that allows to send and receive (via TCP/IP socket) an active ontology between our editor and Protégé. The interoperability service allows ontology developers to use Protégé without changing their habits and only afterwards to

visualize ontologies in OWLGrEd by using various automatic layout algorithms (the user can specify the way ontologies will be visualized by selecting notation options in *Preferences*). In the graphical editor, ontology developers can create new ontologies from scratch or edit graphically (in a WYSIWYG[1] way) the ontologies imported from Protégé. All graphically developed ontologies can be exported to Protégé from where they can be stored to various formats or processed with OWL reasoners.

The graphical ontology editor OWLGrEd has been made available to the public online at http://owlgred.lumii.lv and currently is being widely used worldwide. Currently (from Nov 1st, 2013 till May 31st, 2015), it has been downloaded 2156 times from outside Latvia. The uptake demonstrates the usability of the proposed notation. Figure 13 shows the cities (outside Latvia) from where OWLGrEd has been downloaded. Each circle represents a city, and the size of the circle represent the download count from that city. The largest circle corresponds to 28 downloads, the smallest to 1 download. The statistics have been collected from the OWLGrEd web page by using the Google Analytics service.

6.4 RELATED WORK

The proposed OWL visual notation is based on the UML [16] class diagram notation. In our opinion, the most important feature for achieving a readable graphical OWL notation is the maximum of compactness. The proposed notation achieve it by exploiting the native power of UML and by using its notation as far as possible. The UML class diagram notation is extended with the Manchester-like syntax [10] for the missing OWL features, thus making the notation compact and comprehensible. Furthermore, many software engineers are already familiar with the UML notation and use it to model data; we expect

1 Acronym from *what you see is what you get* – denoting the representation of text on a screen in a form exactly corresponding to its appearance on a printout. Src.: New Oxford American English Dictionary.

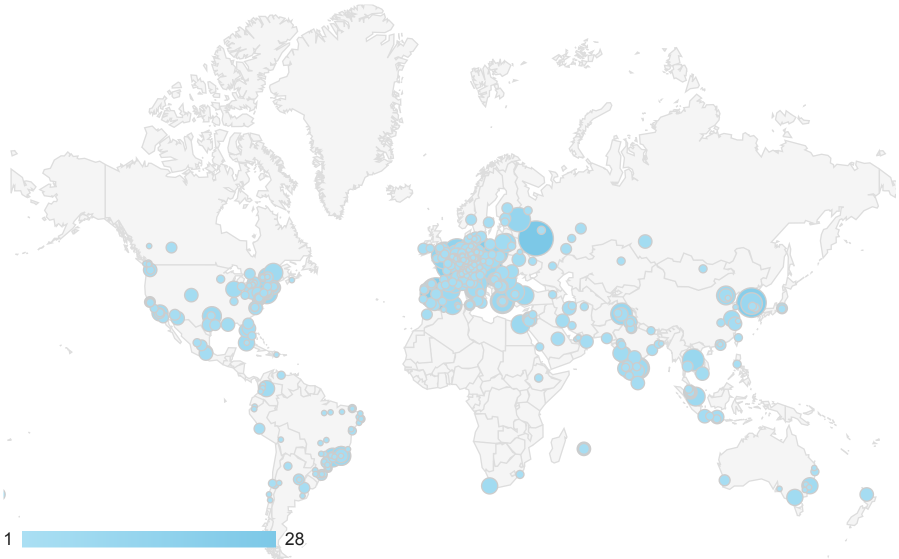


Figure 13: OWLGrEd downloads by cities outside Latvia from Nov 1st, 2013 till May 31st, 2015

that this familiarity would enable them to adopt easily the new formalism we propose.

The application of UML class diagram notation to OWL is not an entirely new idea; it has been implemented in the *TopBraid Composer* [15]. However, that implementation is based on a simplified UML class diagram model, it lacks graphical editing facilities, and the available graphical services are limited. Some other solutions have been proposed for the graphical UML-style representation of OWL ontologies; the most notable is ODM (see [6, Chapter 14]) that defines a UML profile for OWL. The main advantage of ODM approach is the possibility to use existing UML tools for ontology modeling. Meanwhile, the price for this compatibility is a more verbose notation that does not facilitate comprehensibility.

6.5 CONCLUSIONS

In this chapter, we created an OWL metamodel that is a constraint layer above UML class diagram metamodel. This will enable transformation languages to work simultaneously with OWL constraints. We also developed a graphical notation for OWL ontologies that is an

extension of UML class diagram notation. The notation allows people to use the additional expressive power provided by OWL without learning an entirely new notation. Also, the notation and its editor OWLGrEd appears to be useful by itself, as demonstrated by the worldwide usage.

---

7 EXTENDING OWL ONTOLOGIES WITH QUERY-BASED CONSTRUCTOR CLASSES

In this chapter, we discuss how to merge the world of transformation languages with the world of Semantic Web ontologies and reasoners to create a more powerful knowledge representation formalism. As was explained in the previous section, OWL ontologies were designed to capture the domain expert's knowledge in a direct and intuitive way. OWL was intended for the applications in Semantic Web where the information is distributed, and it is a norm that only a small part of the whole knowledge is available at a time. This is because the OWL formalism is based on the so-called "open-world assumption", meaning that, if something is not known to be true, then this does not imply that it is false. For example, assume we only know that Bob is a person. From this, we cannot derive that Bob is a student nor that he is not one. Usually, our knowledge is incomplete, and this can be correctly represented in OWL. However, because of the expressiveness of ontology languages and the available support tools (e.g. Semantic Web reasoners), more and more people outside of the semantic web community wish to use OWL for their applications.

The open world assumption, although suitable in many contexts, poses some problems for the use of ontologies in other domains. The main reason – there are intuitive concepts that cannot be described with OWL ontologies. One such intuitive concept is classes that deal with some form of the "closed-world reasoning", e.g. objects for whom it may be not known whether they possess a value of some property or not. Due to the open-world assumption, OWL can only define either classes of objects for whom it can be proven that they must possess some property value, or classes of objects for whom it can be proven that they cannot have a property value. It is also impossible to define classes that involve conditions on aggrega-

tions, e.g. objects for whom the sum of some attribute values must be equal to some fixed value.

The prevailing attitude in the semantic web community is that such cases should be handled outside of the ontology either by adding the additional information in a preprocessing step or by deducing it later, in the application. Such attitude is right for the logical purity of the language, but it is problematic for a practical adaptation because the end-users want to treat the knowledge base as a black box [59]. They want to specify everything in the ontology and let the knowledge base decide, what to calculate by using a reasoner and what to derive by using other means.

In this chapter, we will present an extension of OWL for the specification of classes that cannot be described by using only OWL constructs. The extension is based on the lQuery language that was presented in the 4th chapter. We also show how it can be integrated with a reasoner in practical applications. We will start with an example ontology, which will illustrate some shortcomings of the OWL language. After this, we will extend the example to show how the proposed extension improves the situation.

7.1 MOTIVATING EXAMPLE – A UNIVERSITY ONTOLOGY

The example is an excerpt from an ontology of a university information system. The ontology is intended for data storage, data validation, and query answering. It is shown in Figure 14 (the OWLGrEd notation from the previous chapter is used). The primitive classes are *Teacher*, *Professor*, *Student*, *AcademicProgram*, *Course*, and *Grade*. These classes are sufficient for the purpose of data storage. However, the end-users may want to make queries not only about all students, but also about students that possess some specific features, e.g. students that take some course or students that have graduated. Usually, a feature set describes some intuitive concept that can be named, e.g. students that take some course correspond to a concept *ActiveStudent*. Many intuitive concepts can be defined in OWL as *derived classes* by using class expressions. For example, the class *ActiveStudent* (from

Figure 14) can be defined by an OWL expression "*takes some Course*". *Derived classes* can be used later not only for answering queries, but also for defining of further intuitive concepts.

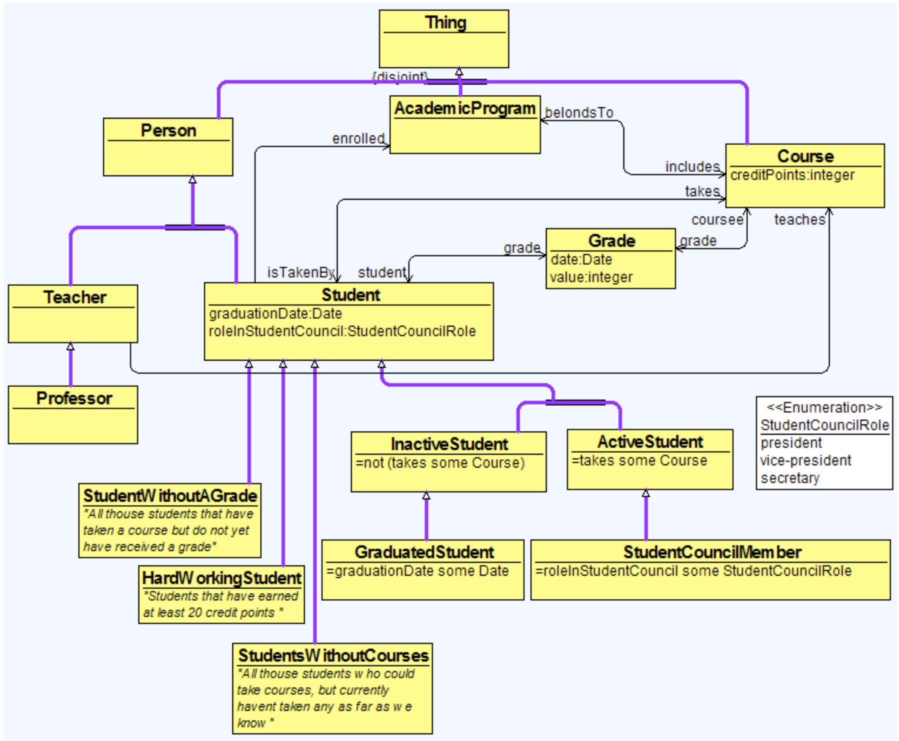


Figure 14: Simplified University Ontology.

However, some intuitive and useful concepts cannot be specified by using OWL. For example, consider instances of students who have not yet registered for courses (imagine that we want to send them a reminder that they will be exmatriculated if they do not register). It may seem that it would be sufficient to define two classes, namely, *ActiveStudents* – those who take some course, and *InactiveStudents* – those who currently do not take any courses. Let us look at a fragment of the university ontology shown in Figure 15 where these two classes are defined. We must be careful: what exactly is meant by these definitions from the perspective of the open world assumption? Let us look at the individuals from the class *Student* in Figure 15. The individual *s2* has a link *takes* to the course instance *c1*, therefore it can be inferred that it is also an instance of the derived class *ActiveStudent*. However, what about the individual *s1*? There are no outgoing links from it. From the perspective of the open-world as-

sumption, this means that we do not know about any outgoing links, but there actually may be some. Hence, we cannot infer neither that the individual *s1* is an *ActiveStudent* nor that he is an *InactiveStudent*. Consequently, it is better to read expressions like "*takes some Course*" and "*not takes some Course*" as "all individuals for whom it can be proved that they take some course" and "all individuals for whom it can be proved that they **cannot** take a course". When read in such a way and by assuming that we have only partial information in our knowledge base, it becomes quite intuitive why the individual *s1* can be classified neither as *ActiveStudent* nor as *InactiveStudent*.

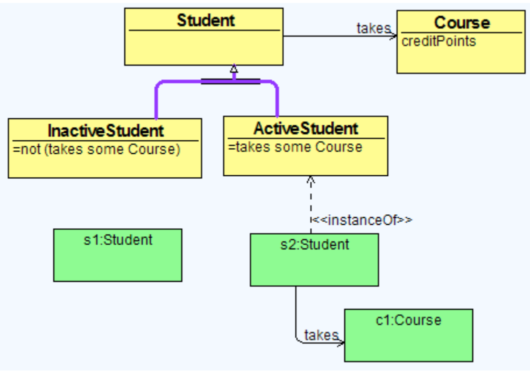


Figure 15: Fragment of the University Ontology with instances.

In Figure 15 the individual *s2* can be classified as an *ActiveStudent* because it has a link *takes* to the course *c1*. However, the individual *s1* cannot be classified as either an *ActiveStudent* or as an *InactiveStudent*. Because there is no enough information whether the individual *s1* takes some course or not.

Actually, no OWL expression can describe only those individuals for whom it is not known whether they have a link or not [41]. Therefore, the only option is to add a primitive class with a natural language annotation explaining what is meant by this class. The class *StudentWithoutCourses* in Figure 14 demonstrates this approach.

Our goal is to present a language allowing to describe classes like the *StudentWithoutCourse*. We will call such classes *Introspective-Classes*.

7.2 lquery selectors in ontologies

As we saw in the previous section, OWL can only refer to individuals about whom it can be proved either that they possess some feature or that they **cannot** possess this feature at all. However, there is no way to refer to individuals about whom there is no information whether they possess some feature or not. Now we will see that lQuery expressions can be used for referring to such individuals.

lQuery expressions are always evaluated in the context of some object collection (e.g. individuals of the class *Thing*), and each object that matches the selector is returned in the resulting collection. Thus, each selector expression defines a new class of objects. The two main types of selector expressions are filters and navigators. Filters are used to obtain a subset of the initial collection based on some condition. Navigators are used to obtaining a new collection of objects from the starting collection. Examples of filter selectors: filtering by a class membership (i.e. there is an explicit *instanceOf* assertion in the ontology), or filtering by *data-property* value. Examples of navigation selectors: the collection of objects that are reachable from the current collection by a given *object-property*, or the collection of values of some *data-property*.

7.2.1 *Integration with Ontology*

To use lQuery expressions in ontology engineering, we need some way for the ontology designers to specify them in the graphical notation. We will extend the OWLGrEd notation from the previous chapter with the syntax for lQuery expressions. In the OWLGrEd notation, classes are represented by boxes, and there is a field (starting with "=") under the class name for equivalent class expressions in the OWL Manchester Syntax. lQuery expressions will be used similarly. Therefore, we will use a similar notation. To distinguish lQuery expressions from OWL expressions, we will enclose lQuery expressions in «» marks (see Figure 17).

The lQuery language also needs to be integrated with the OWL metamodel from the previous chapter. The metamodel is extended with a class *lQueryExpression*, that represents an lQuery selector expression. There is a link *lQueryEquals* from the intuitive class *Class* to the class *lQueryExpression*. This link means that the intuitive class contains all the instances that are returned by the connected lQuery selector, when it is executed on the entire repository. The extended metamodel is shown in Figure 16.

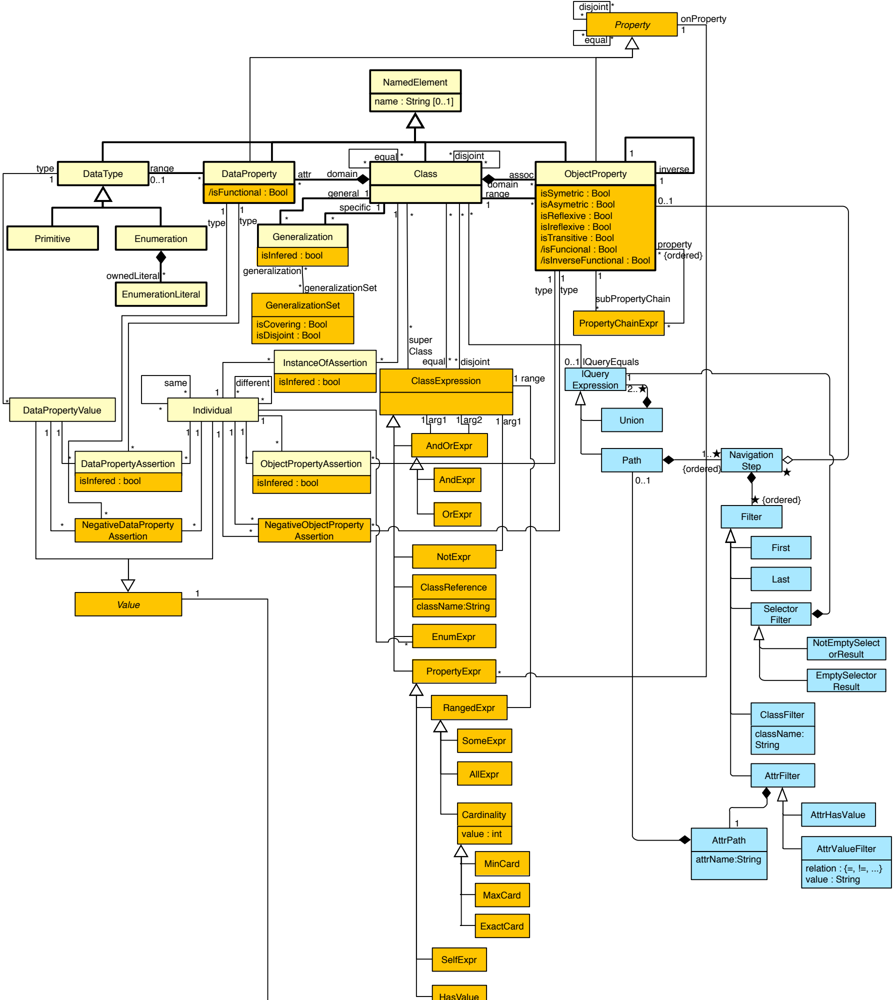


Figure 16: OWL metamodel extension with lQuery constructor classes

The lQuery expressions are serialized in ontology files by using OWL annotation properties [8, Annotation Property]. Annotation properties allow to attach arbitrary information to any OWL entity or assertion. The lQuery expressions are always added to a named

class. We introduce an OWL annotation property *lQueryEquals* whose domain is an OWL *Class*, and the range is *lQueryExpression*. When the ontology is exported from the OWLGrEd ontology notation to an OWL file, the lQuery expressions will be exported as annotations. Figure 18 shows the example from Figure 17 serialized in the Manchester Syntax.

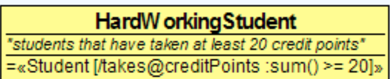


Figure 17: Demonstration of the OWLGrEd syntax extension for the lQuery expressions

```
Datatype: lQueryExpression
AnnotationProperty: lQueryEquals

----------
Class: PassedStudent
    Annotations:
        lQueryEquals "Student:has(/grade/course@creditPoints:sum() >= 20)"^^lQueryExpression
        rdfs:comment "Students that have earned at least 20 credit points"
```

Figure 18: lQuery expressions as annotation properties in Manchester syntax

Datatype: lQueryExpression
AnnotationProperty: lQueryEquals

Class: PassedStudent
    Annotations:
        lQueryEquals "Student:has(/grade/course@creditPoints:sum() >= 20)"^^lQueryExpression
        rdfs:comment "Students that have earned at least 20 credit points"

7.2.2 *Some Examples from the University Ontology*

Now we will consider some examples from the University Ontology that we were not able to specify by using OWL class expressions. First, let us return to the example of students without known courses from the previous section. Figure 19 shows the same ontology fragment but with an additional class *StudentWith-NoKnownCourses*. The class defines the instances that we could not obtain with OWL (the class belongs to the category of *IntrospectiveClasses*). Let us recall that the OWL definition of the class *InactiveStudents* – "not (takes some Course)" – does not describe the instances we want, because the class will contain only instances for which it can be proved that they cannot possess a link *takes*. However, now we are in a situation where there is no information from which we could prove that the individual *s1* could possess a link or not. Therefore, it is not classified as either *InactiveStudent* or *ActiveStudent*. In contrast, the lQuery selectors work

only with the information that is present in the local knowledge base and assume that everything that is not known is false.

Let us consider how this example works step by step. The class *StudentWith-NoKnownCourses* is defined by the lQuery expression "*Student:not(/takes)*". The expression is a selector chain that consists of two primitive selectors – selector by class name ("*Student*"), and a negative filter selector. The first primitive selector returns a collection of instances that have been classified as a *Student*. In Figure 19 that collection contains the individuals *s1* and *s2*. The next selector ("*:not(/takes)*") is evaluated in the context of this collection. The selector leaves only those instances from the context collection for which the sub-selector ("*/takes*") returns an empty collection. In our example, it leaves only the instance *s1*. Finally, the query result is materialized as an *instanceOf* assertion from each instance in the result collection to the class *StudentWithNoKnownCourses* (shown as a red dotted line in Figure 19).

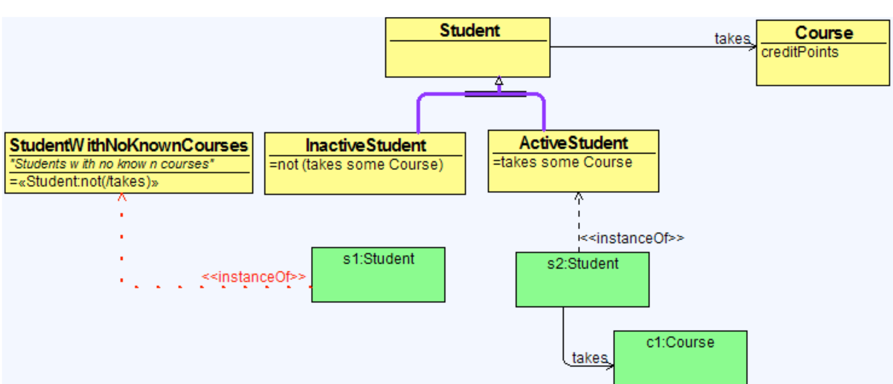


Figure 19: Demonstration of the lQuery semantics in contrast to OWL semantics

Another feature, missing in OWL, is aggregation operations. Consequently, many intuitive classes cannot be defined by using OWL. For example, in the university ontology such a class is "students that have taken at least 20 credit points" (*HardWorkingStudents*). In lQuery this statement can be written as "*Student [/takes@creditPoints :sum() >= 20]*". Let us consider how this expression is evaluated on an example from Figure 20. The lQuery selector consists of two parts – the selector by class name and the filter by condition whose sub-selector is a

selector chain. The result of the "*Student*" selector is a collection with instances *s1* and *s2*. The next selector ("*[/takes@creditPoints :sum() >= 20]*") will return those objects from the context collection for whom the condition evaluates to true. In this case the condition is set to the sum of data property values, i.e. the sum must be greater or equal to 20. The path "*/takes@creditPoints*" means that for each student we will find all the courses that the student *takes* and obtain values of the corresponding data-property *creditPoints*. The result is a collection of integers that is passed to the operator ":sum()". The result is compared with the value 20 and, if it is greater or equal, then the student is added to the result collection. In the current example, the collection will contain only the individual *s2*.

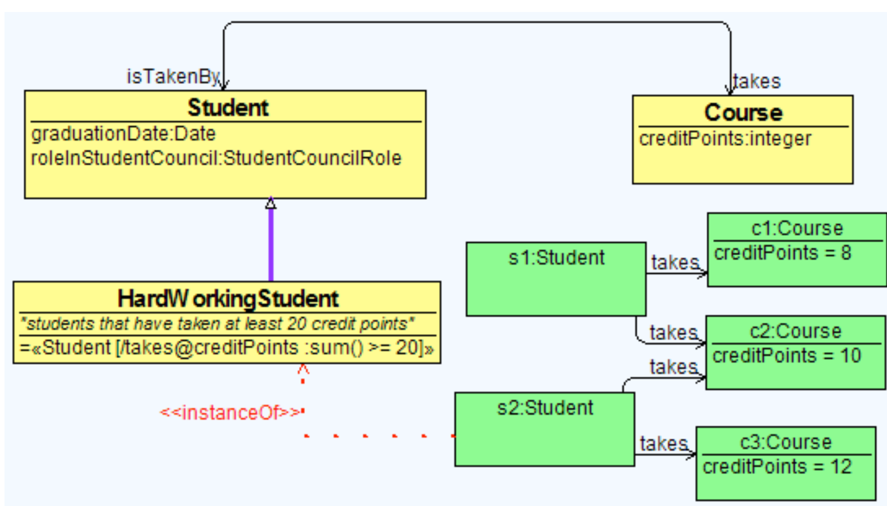


Figure 20: Demonstration of the lQuery aggregation expressions.

We could also define a more complex class, like "students that have taken at least 20 credit points and have no grade less than 4". It can be written by using an lQuery as follows – "*Student [/takes@creditPoints :sum() >= 20] :not([/grade@value < 4])*".

7.2.3 *Advanced Example from the University Ontology*

Now, let us consider a more advanced example from the University Ontology. We started the previous subsection with the problem of defining a class with only those students about whom there is no in-

formation whether they take some course or not. We discovered that it was impossible to define such a class in OWL (Figure 15). Then we demonstrated how we can use lQuery to define this class (Figure 18). The main reason we succeeded was the use of the closed-world semantics in lQuery. From that example, it may seem that we would always want to use the closed-world semantics. However, in some cases the situation may be more complicated.

Let us suppose that we want to define the class of students to whom we want to send reminders to register for some course. In this example, the ontology (shown in Figure 21) will have two additional classes, namely, *GraduatedStudent* (those that have graduation date) and *StudentCouncilMember* (those that have some position in a student council). For both of these classes, there is a corresponding OWL definition of what it means for an individual to belong to that class. In addition, it is known that a *GraduateStudent* cannot take any courses because it is a subclass of *Inactive-Student* and *StudentCouncilMember* must take at least one course because it is a subclass of *ActiveStudent*.

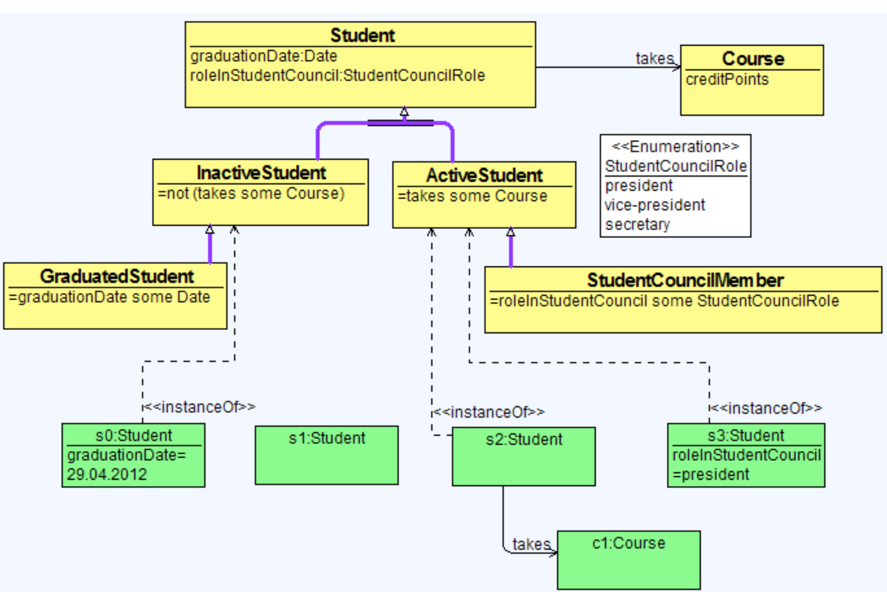


Figure 21: Extended example from Figure 15.

Let us consider what this means in terms of instances (remember that we are thinking in terms of the open-world assumption). Now, we have two additional students – *s0* and *s3* and for each of them

we know an extra feature. For the student *s0* we know his graduation date and for the student *s3* we know his position in a student council. Thanks to the additional information and the corresponding class definitions it can be inferred that *s0* is an *InactiveStudent* and *s3* is an *ActiveStudent* (even though we do not know which course he takes, we know that he is taking some course because otherwise he could not be a *StudentCouncilMember*). If we return to the original example – "students whom we need to send reminder to register to some course" – we can see that the closed-world assumption is not by itself sufficient to obtain what we need because then we would obtain all the individuals without a *takes* link, i.e. *s0*, *s1* and *s3*. But we actually want only the individual *s1* because there is no need to send reminders to students who have graduated (*s0*) or to students about whom we know indirectly that they are taking some course (*s3*).

As we see, the closed-world assumption is not sufficient, and to define the class that we want, we need the possibilities to refer to the results of the reasoning process. Therefore, we need to write the lQuery expression as follows: take all students, then exclude students about whom it is known that they are an *InactiveStudent* or an *ActiveStudent*. (This can be done using lQuery expression "*Student:not(InactiveStudent):not(ActiveStudent)*"). In the example of Figure 21 this expression will return a collection with only one student – *s1* – which is exactly what we wanted.

However, in some cases, the *instanceOf* assertions that allowed lQuery to obtain the correct result, are not present in the ontology directly. Typically they are calculated by the reasoner. It is not required that the inferred *instanceOf* assertions must be explicitly present in the ontology all the time. Therefore it raises a question, how should the reasoner and the lQuery interoperate? We will answer this question in the next section.

7.3 integration with a reasoner

There are two kinds of derived classes in our extended ontology language, namely, classes defined by OWL class expressions and classes

defined by lQuery selector expressions. Now we will define the *IntegrationAlgorithm* that will allow to use both types of expressions to classify instances.

The *IntegrationAlgorithm* will work as follows. It will start by classifying the ontology by using an OWL reasoner. Because, as we saw, for lQuery expressions to work correctly they need all the inferred *instanceOf* assertions to be explicitly asserted in the ontology. After the reasoner has classified the ontology, i.e. added the inferred *instanceOf* assertions, the *IntegrationAlgorithm* will perform the lQuery classification step. This step first finds all the classes that are defined by the lQuery selector expression. Then, for each class, it evaluates the selector expression and obtains a collection containing the corresponding individuals. If the collection found is a strict superset of the currently asserted class individuals, then it adds *instanceOf* assertions for each newly found individual to the ontology. However, if the class contains an explicitly asserted individual that is not in the collection, then it report a contradiction.

After the lQuery step has finished, new *instanceOf* assertions will be in the ontology that could be used by the reasoner to derive some additional information. Therefore we would want to run the reasoner one more time. After this, of course, lQuery could again find some additional information, and so on. Let us show that this algorithm always terminates with either a contradiction or an ontology where no new information can be deduced.

First, note that running the reasoner can yield 3 types of results: the reasoner finds a contradiction, the reasoner finds new *instanceOf* assertions or the reasoner finds nothing new, i.e. every hidden assertion has been materialized, and there is no contradiction. Because the OWL reasoner specification is such that no new instances can be created, and no existing instances can be deleted. The lQuery step can have the same three types of results. So both, the lQuery, and the reasoner, can only add new *instanceOf* assertions to the ontology. Therefore running them one after another in a loop will end in either a contradiction or with an ontology state where running one or the other will result in the same ontology state.

By using this algorithm, we can reuse classes derived in lQuery in OWL expressions and vice versa.

7.4 RELATED WORK

The related work can be divided into two categories: a) papers proposing query and rule languages for semantic web technologies, and b) papers proposing OWL extensions with some closed world capabilities. Let us first look at the query languages that could be used in place of lQuery.

The most widely used query language for RDF data stores is SPARQL [13]. Its main purpose is retrieval and manipulation of RDF triples [12]. Because RDF is one of the standard serialization formats for OWL, and almost all semantic web data is stored in RDF databases, SPARQL can also be used for writing queries to OWL data. The main problem with this approach is that we need to encode the RDF serialization of OWL expressions in the query, thus making them "verbose, difficult to write, and difficult to understand" [66]. Consequently, SPARQL is unsuitable for usage instead of lQuery because our goal was to build an intuitive language in which the queries could be expressed by using the ontology terms, and not their underlying serialization in some other language.

Another language that can be used alongside OWL for defining new classes is SWRL [14]. SWRL extends OWL with a new type of assertion – a rule that consists of an antecedent part and a consequent part. Both parts consist of a conjunction of atoms. Informally, the SWRL rule can be read as: if all atoms in the antecedent part are true, then the consequent part must also be true. SWRL has the full expressive power of OWL-DL, combined with (binary) function-free Horn logic, consequently, it is undecidable [44], and full implementation of it is impossible. Additionally, because SWRL uses the same open-world assumption on which OWL is based, it has the same problems for our purpose, i.e. it cannot describe classes of objects for which something is not known.

A similar language from a different domain, which, in fact, largely inspired the design of the lQuery, is the OCL [5] – a constraint language for UML. The main difference is that the semantics of OCL is tailored for constraint checking and not for classification. Therefore, it always works in the context of a single instance and not in the context of all individuals.

There have been attempts to extend OWL with operators to describe objects with unknown features (epistemic operators) [41]. These elements have largely been based on work on non-monotonic reasoning. Currently, only a partial success has been achieved in this direction. The main advantage of introducing epistemic operators directly into OWL (and writing a reasoner that understands them) is the possibility of proving that the ontology is consistent. In our proposed system the reasoner and the extension are only partially integrated – but an inconsistency can happen only on the instance level; i.e. a contradiction can be found only when a contradictory instance is added to the repository. The reasoner that understands OWL and epistemic operators could find a contradiction earlier, already from class definitions.

7.5 CONCLUSIONS

In this chapter, we discussed the benefits and limitations of OWL for the task of knowledge capture and retrieval. The main emphasis was put on exploring how suitable OWL is for the definition of derived classes that are intuitive for the end-users. The main advantage of OWL is the availability of reasoners that can classify individuals given only partial information about them contained in class definitions. However, the reasoner can classify individuals only when it can prove either that an object possesses some feature or that it cannot possess this feature. Consequently, it is impossible to define the class of precisely those individuals about whom some information is missing, e.g. students about whom we do not know what courses they have taken. Such *IntrospectiveClasses* are very natural and useful in practical applications.

We proposed an OWL extension that retains all the benefits of the pure OWL but solves the problem of introspective class definitions and retrieval of their instances. The extension consists of two parts, namely, a selector language lQuery and an algorithm for integrating it with the existing reasoners. The proposed extension allows to classify instances by using either OWL semantics or lQuery semantics. One drawback of this solution is that it is no longer possible for reasoners to prove that the extended ontology is consistent by looking only at the class level.

The primary advantage of the proposed extension is the ability to define more derived classes at the ontology level and use them for the classification of individuals. These additional classes make it easy for end-users to write ad-hoc queries because they can select from a larger set of predefined intuitive classes instead of specifying them in a low-level query language. It is possible because the proposed algorithm can materialize both OWL inferences and lQuery inferences by storing them in a data store. Thus higher-level query languages, such as ViziQuer [24, 25], Facet Graphs [43], etc., can take the advantage of the additional expressivity following from our extension without changing anything in their implementation.

---

8 ONTOLOGY-BASED TOOL BUILDING FRAMEWORK: ARCHITECTURE PROPOSAL

In the previous two chapters, we presented solutions to two of the main problems for moving toward ontology-based tool building framework. Now we will present a way how to use them to upgrade the architecture of the MDA based tool building framework from Chapter 3 to achieve the benefits of the ontology-based system outlined in Chapter 5.

The motivation for moving towards an ontology-based tool building framework, as explained in Chapter 5, was to reduce the need for programming of tool-specific transformations. The reduction can be achieved by moving the tool-specific behavior description from transformations to declarative model annotations. The declarative annotations contain lQuery based selectors that can define classes. The defined classes then can be used by OWL reasoners to achieve the desired tool behavior (constraint checking, element styling). For this purpose in Chapter 6, we have designed a UML-based OWL metamodel and in Chapter 7 we have designed the OWL orthogonal extension metamodel and algorithm for the integration of reasoners with lQuery.

Let us start by recapping the components of the MDA based tool building framework as presented in Chapter 3. The framework consisted of 4 parts: Model Repository for storage of model instances; View Engines for user interaction; Transformation for interaction logic; and TDA Core through which these components communicate. The TDA Core handles communication as *Commands* and *Events* which were stored in the model repository. The access to the model repository was organized via a universal interface defined by TDA, called RAAPI [54]. The transformations were divided into two sets: a) the Universal Transformations handling common tasks, like ele-

ment creation/deletion and copy/paste; and b) tool specific transformation handling custom behaviors, like constraint checking, dynamic style selection, and interaction with outside systems. The overall architecture of the MDA based tool building framework is shown in Figure 22.

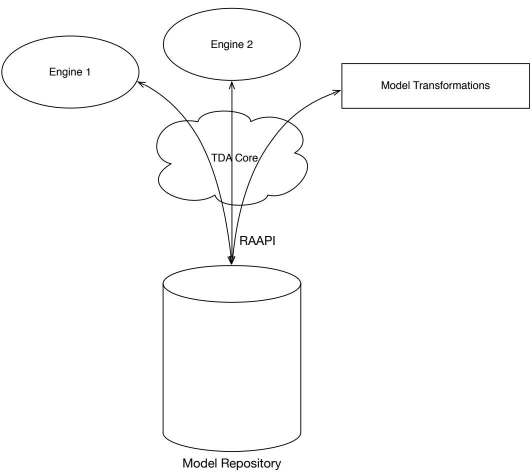


Figure 22: Overall architecture of the MDA based tool building framework from Chapter 3.

Our goal is to reuse as much of the original MDA-based architecture as possible, to avoid rewriting of the existing View Engines and the Universal Transformations. All components of the tool building framework communicate with one another through the TDA Core using *Commands* and *Events* stored in the model repository, hence, we can replace the model repository with a new one that supports the extended OWL and lQuery semantics. As long as the new repository is supporting the same outside interface (API) as the existing one, the TDA Core will work with it, and we do not need to change the current View Engines and Universal Transformations.

The question then becomes how to implement the new repository? We can reuse the existing model repository (used in the MDA-based tool building platform) as a core for the new repository. The current repository is preloaded with the OWL Orthogonal extension metamodel discussed in Chapter 7. Then we need to rewrite the API functions to work according to the new semantics of extended OWL and lQuery semantics. The *read* part of the API functions can work in the

same way as described in Chapter 4, where lQuery was presented. This is possible because the core of the OWL Orthogonal extension metamodel (Figure 16) is the same as lQuery repository metamodel (Figure 4). The implementation of the *write* part of the API is more complicated because the changes can invalidate some of the inferred relations and also result in new inferred relations as was explained in Chapter 7. Thus, in each *write* operation we need to cancel the previously inferred results, apply the *write* operation according to lQuery semantics from Chapter 4 and execute the integration algorithm of Section 7.3. After that, when the next *read* operation is performed, it will have all the inferred information already explicitly present in the repository, as regular objects, attributes, and links. The new overall architecture is shown in Figure 23.

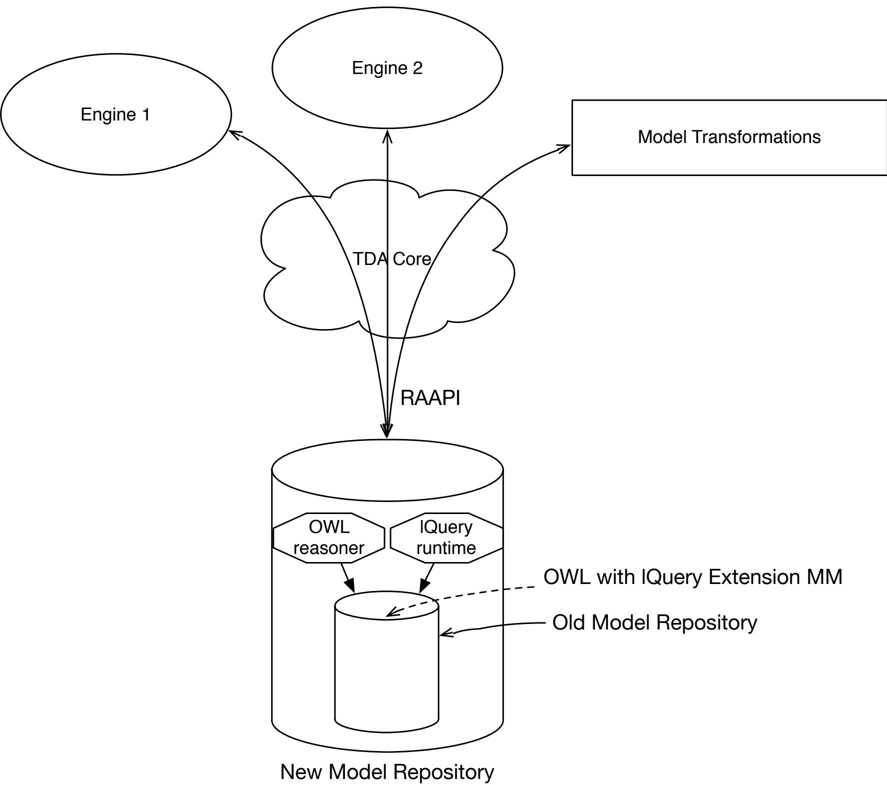


Figure 23: New overall architecture of OWL-based tool building framework.

With the new architecture, the domain-specific tool designer can create subclasses in the tool definition metamodel (presented in Chapter 3) by using the full expressive power of OWL and lQuery selectors. These subclasses can be defined by using the OWLGrEd graphical no-

tation and the editor presented in Chapter 6. This is done with standard configuration mechanism that uses the Type instances (see Section 3.2). Meanwhile the existing View Engines and Universal Transformations can work without changes.

To better understand how the proposed new architecture will work in practice, let us consider a small fragment of an *activity* diagram editor.

8.1 A RUNTIME EXAMPLE IN THE PROPOSED ARCHITECTURE

In Chapter 5 we used the *activity* diagram editor as a motivating example for an ontology-based tool building framework. Now we will use a small fragment of this program to illustrate how it functions in the proposed ontology-based architecture. The editor will consist of an *activity* diagram and a *start* element. We will define the following behaviour: 1) the *activity* diagram must contain no more than one *start* element; 2) if the diagram does not contain a *start* element, it should have a red background color; 3) if the diagram contains exactly one *start* element, it should have a white background color.

First let us look how the tool definition metamodel of Chapter 3 is represented in the new architecture. Remember that in the new architecture we have a new repository, which contains inside the old one, and in which the OWL Orthogonal Extension Metamodel (OOEM) of Chapter 7 is loaded. Consequently, the tool definition metamodel is an instance of OOEM. However, from the outside it continues to look as before because everything is working through the same API (called RAAPI). Figure 24 illustrates the situation.

Next let us look at how the *activity* diagram fragment is added, first at the logical level in OWLGrEd notation, and then in the internal representation. The *acitivity* diagram specific classes are defined as subclasses of the tool definition metamodel of Chapter 3. First, a subclass (*ActivityDiagram*) of the *Diagram* class; and a subclass (*Start*) of the *Element* class is created. Two additional subclasses of the *Activity-Diagram* class also are needed – one for each style situation. The first subclass is *EmptyActivityDiagram*, it contains two expressions: a) an

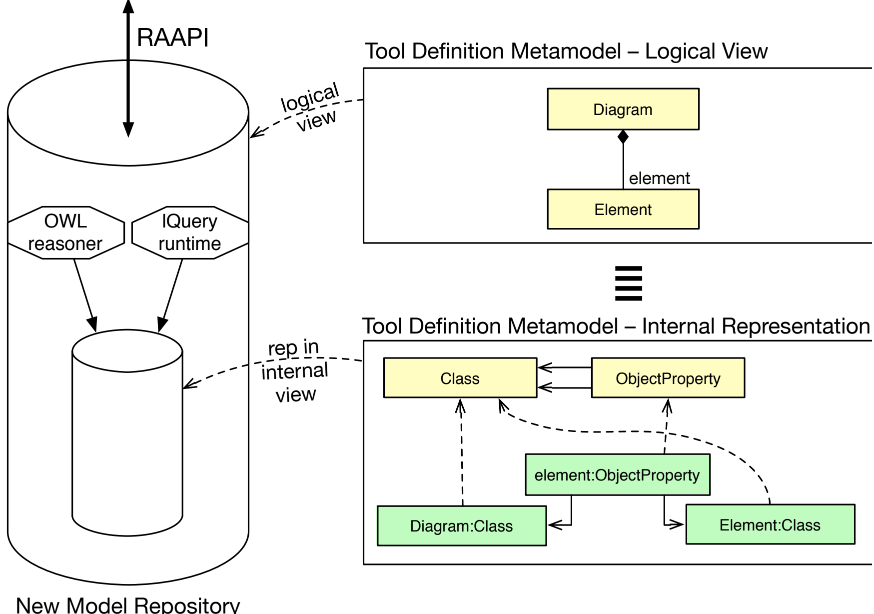


Figure 24: Fragment of the Tool Definition metamodel in the New Repository – Logical View and Internal Representation

lQuery expression that defines the class so that it contains only those *activity* diagrams that have no *start* elements; and b) an OWL expression that specifies the red background color for instances of this class. The second subclass is *NonEmptyActivityDgr*. It defines the *activity* diagrams with at least one *start* element by using similar expressions. Also, a validity constraint is created that says that all *activity* diagrams must have at most one *start* element, this is done by adding an OWL *subclassOf* assertion to the *ActivityDiagram* class with OWL class expression *element max 1 Start*. The updated model is shown in Figure 25. The upper part of the image shows the logical view using OWLGrEd notation and the lower part shows the internal representation as an instance of the OWL Orthogonal metamodel.

Now let us look what happens at a runtime when the user creates a diagram. First, just like in the MDA-based tool building framework of Chapter 3, the Graph Diagram Engine creates an *event*, corresponding to the new diagram creation request. The Universal Transformation interprets the *event*. As part of the interpretation, the transformation uses the *create_instance* function from the model repository API with a class name parameter *ActivityDigram*. In the previous

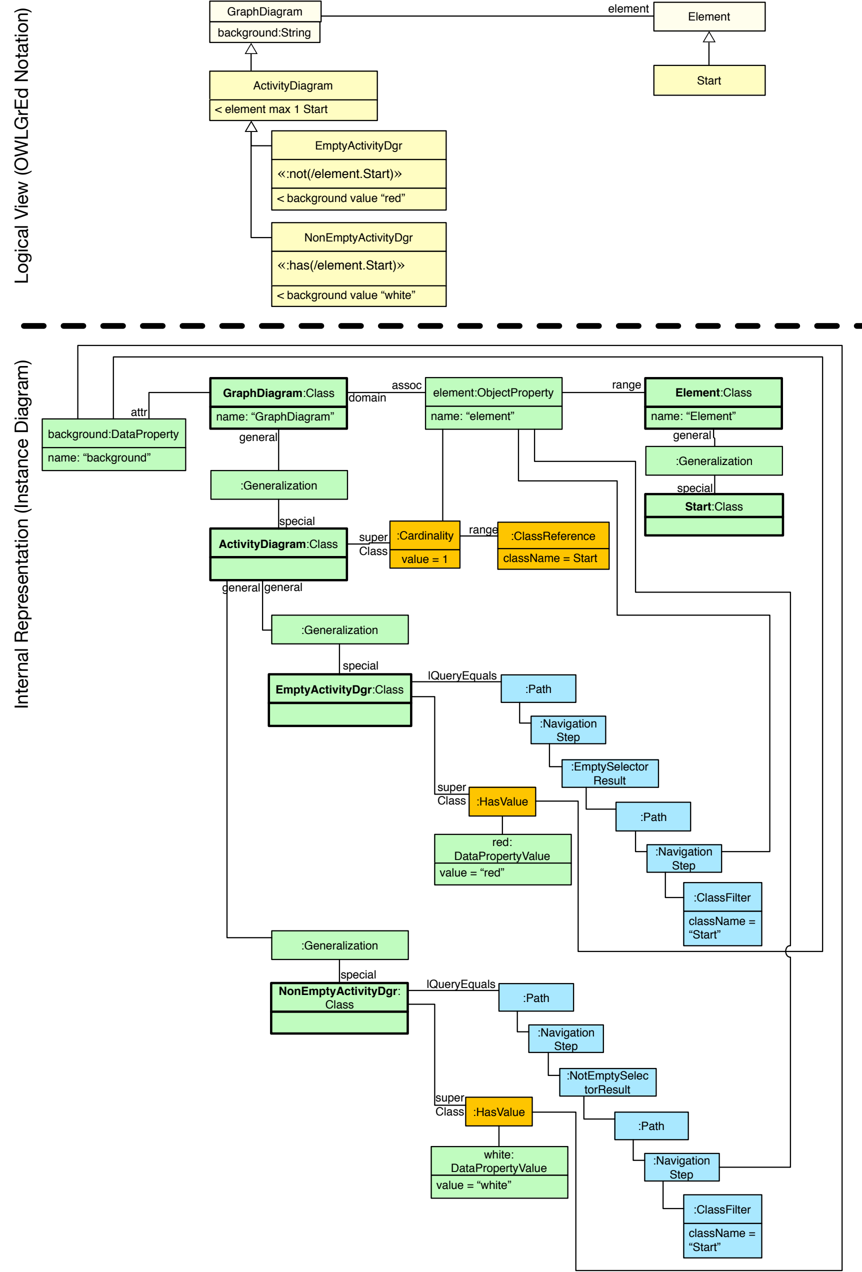


Figure 25: Fragment of the tool definition metamodel with activity diagram specific subclasses (Logical View and Internal Representation)

architecture, the instance would simply be created, but in the new one, there are additional steps. Specifically, first the repository deletes all previous inference results. This time, there are no previous inferences to undo because we assume that we are starting from scratch. Then, the new *ActivityDigram* instance is created, and the inference engines are invoked. First, the lQuery selector engine goes through the classes defined by lQuery selectors, in this case, they are *EmptyActivityDgr* and *NonEmptyActivityDgr*. The newly created *ActivityDigram* instance matches the selector from the *EmptyActivityDgr* class («:not(/element.Start)»). Thus, an *instanceOf* link is added (and marked as *inferred*) from the newly created *ActivityDigram* instance to the class *EmptyActivityDiagram*. Then the inference algorithm continues and invokes the OWL reasoner inference engine. It applies the rule according to which all *EmptyActivityDiagram* instances must have attribute "background" value "red". Thus, an attribute assertion "background = red" is added to the repository, and marked as inferred. When nothing more can be inferred, the inference algorithm finishes, and the Universal Transformation continue its work. Finally, the Graph Diagram Engine is notified that all changes have been done and displays the newly created diagram to the user. The repository state (logical and internal encoding) after the addition of the diagram is shown in Figure 26.

Notice that in the Logical view (the upper part of the image) the newly created instance is both an *ActivityDiagram* and an *EmptyAcitvityDgr* and its background attribute value is "red". However, in the Internal view (the lower part of the image) we can see that some of the assertions are inferred, and some of them are not inferred (axioms). That distinction allows us to apply and retract inference results while the rest of the framework can operate as if everything is explicitly asserted.

Now let us see what happens one step further when the user wants to add a *start* element to the diagram. The Universal Transformation receives the corresponding *event* (add a box with type *Start* to the active diagram). The Universal Transformation must issue two repository API calls to accomplish this. First, it must create a *Start* class

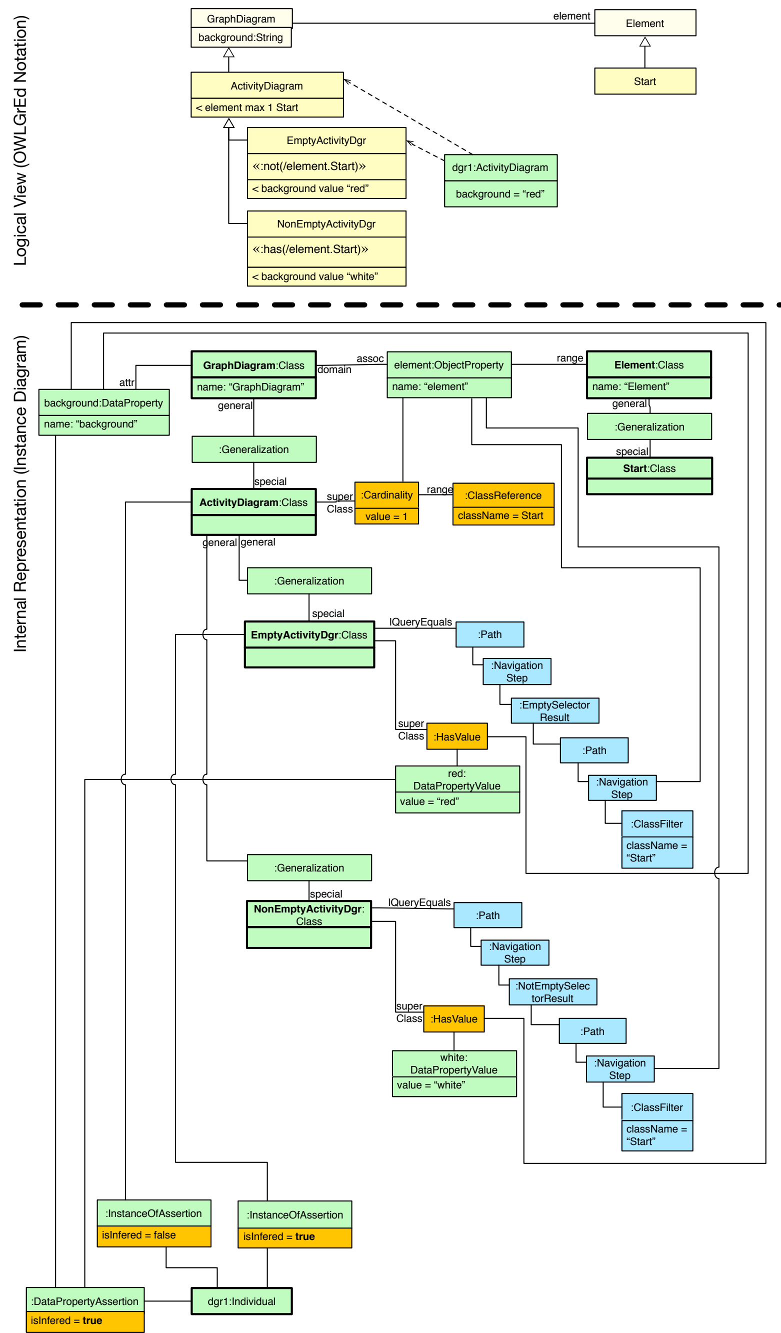


Figure 26: The state of the tool definition metamodel with one empty activity diagram (Logical View and Internal Representation)

instance, secondly it must create a link *element* between the *ActivityDiagram* instance from the previous paragraph and the newly created *Start* instance. As previously said, each API call to the new repository first retracts all inferences, then makes the change and finally reruns and materializes new inferences in the repository. So, before adding a new *Start* element, the algorithm retracts the inferred *EmptyActivityDgr instanceOf* relation and the red background color data property assertion from the instance of *ActivityDiagram*. Then the *Start* class instance is created, and the inference is rerun. In this case, because the start element is not yet connected to anything, the results are the same, i.e. the activity diagram is still empty, and, therefore, its background color is still red. Now comes the second step, the addition of the *element* link from the *ActivityDiagram* instance to the just created *Start* instance. Again the new repository first retracts all inferences, then adds the *element* object property assertion and finally reruns the inference algorithm. This time, the results are different – the activity diagram is no longer classified as an *EmptyActivityDgr*, but instead it is now a *NonEmptyActivityDigram* (because it matches the lQuery selector "*«:has(/element.Start)»*"). Consequently, the diagram instance has a new inferred background color value – "white". Afterwards the Graph Diagram Engine is notified of the changes and shows the new repository state (shown in Figure 27) to the user. The user sees that the *start* element has been added, and also that the diagram background is now white, meaning that all constraints are satisfied.

Finally, let us go one step further and see what happens when the user tries to commit an error by adding a second *start* element to the diagram. In that case, the inference engine encounters a contradiction because the *AcitivityDiagram* class includes an OWL constraint that it can contain at most one *Start* element. Thus, the repository becomes inconsistent, and no further inferences can be made. To avoid the contradiction, the repository state is set back to the snapshot before the user event. Moreover, the user is provided with the minimum set of axioms that produced the contradiction (supplied by the OWL reasoner). Hence, the user can conclude what was the problem and act correspondingly.

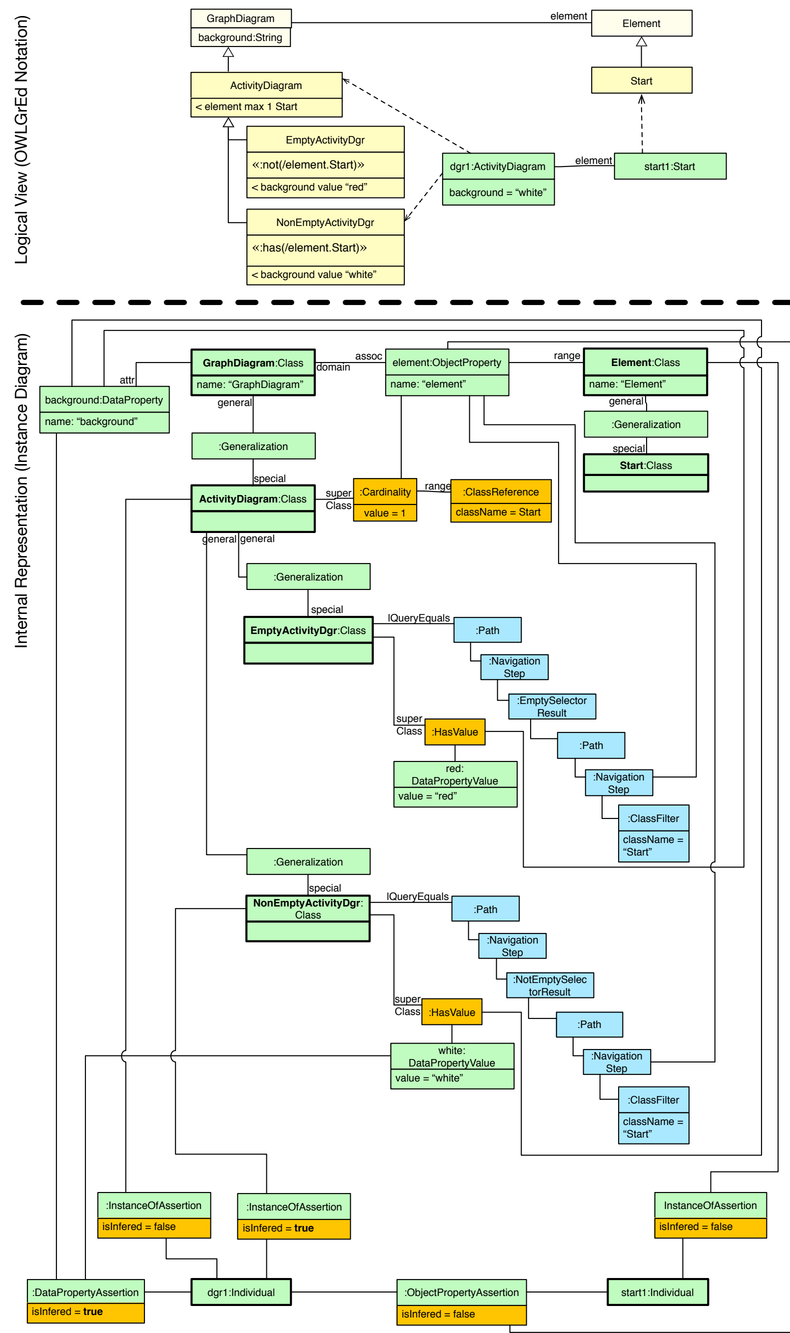


Figure 27: The state of the tool definition metamodel with one empty activity diagram (Logical View and Internal Representation)

8.2 conclusions

This concludes our proposal of how an ontology-based tool building framework could work. Using such a framework domain-specific tools can be built just by declaratively describing their behavior by using an extended OWL and lQuery modeling language. This will make building of domain-specific tools faster and changing them – much easier. At the same time this approach can be susceptible to side effects that are hard to understand, therefore it requires from the tool definer a deep understanding of the nuances of *open-world* and *closed-world* reasoning.

CONCLUSIONS

In this Thesis, we have considered two approaches to defining DSML Tools. The first approach is an extension of the model-based method, the second is a new ontology-based method for tool definition. The research was done in the context of the tool building platform GRAF. The author of the Thesis has participated in the development and continues to participate in its further evolution, moving it towards a fully ontology-based platform.

The four main results of the thesis are:

- the model transformation language lQuery;
- a new metamodel and graphical notation for the ontology language OWL;
- an editor for the graphical notation; and
- an orthogonal extension of the ontology language OWL with expressions from transformation languages.

The transformation language lQuery is a new type of transformation languages, which is specifically designed for transformation tasks that are common in graphical tool building contexts. The language has been used for the development of the tool-building framework itself, as well as for the development of several DSML Tools by using the framework.

The ontology-based approach for tool building presented the ways how the ontologies and reasoning software can be used to easy the task of tool building. The author presented the principles for a tool-building framework based on OWL ontologies and reasoning software.

For the development of an ontology-based tool-building framework, the author developed a metamodel and graphical notation for OWL. The metamodel and notation were designed as a layer above

UML class diagrams. The design allows the future versions of the tool-building platform to use ontologies as its core metamodeling language. Also, the developed ontology notation is already widely used worldwide.

The OWL orthogonal extension with transformation language expressions presents a way how non-monotonic class expressions can be added to OWL. This development enables a wider class of problems that occur in tool building tasks to be described in a declarative way, thus easing the development of new DSML Tools.

BIBLIOGRAPHY

[1] EMF: Eclipse Modeling Framework, . URL http://www.eclipse.org/emf/. (Cited on pages 18 and 60.)

[2] EMF Model Query Developer Guide, . URL http://help.eclipse.org/helios/index.jsp?nav=/22. (Cited on page 60.)

[3] Meta Object Facility (MOF) 2.0 Query/View/Transformation, V1.1, . URL http://www.omg.org/spec/QVT/1.1. (Cited on page 60.)

[4] Meta Object Facility (Mof™) Core 2.0, . URL http://www.omg.org/spec/MOF/2.0/. (Cited on pages 15 and 31.)

[5] OMG Object Constraint Language, Version 2.3.1. URL http://www.omg.org/spec/OCL/2.3.1. (Cited on pages 53, 60, and 96.)

[6] Ontology Definition Metamodel. URL http://www.omg.org/spec/ODM/1.0. (Cited on pages 69, 76, and 81.)

[7] OWL 2 Web Ontology Language, . URL http://www.w3.org/TR/owl2-overview/. (Cited on pages 16, 63, and 69.)

[8] OWL 2 Web Ontology Language Structural Specification, . URL http://www.w3.org/TR/owl2-syntax/#Annotation_Properties. (Cited on page 88.)

[9] OWL Functional Syntax, . URL http://www.w3.org/TR/owl2-syntax/. (Cited on page 69.)

[10] OWL 2 Manchester Syntax, . URL http://www.w3.org/TR/owl2-manchester-syntax/. (Cited on pages 75 and 80.)

[11] Protégé 4. URL http://protege.stanford.edu/. (Cited on pages 68 and 79.)

[12] RDF Primer. URL http://www.w3.org/TR/rdf-primer/. (Cited on pages 16 and 95.)

[13] SPARQL Query Language for RDF. URL http://www.w3.org/TR/rdf-sparql-query/. (Cited on page 95.)

[14] SWRL: A Semantic Web Rule Language Combining OWL and RuleML. URL http://www.w3.org/Submission/SWRL/. (Cited on page 95.)

[15] TopBraid Composer. URL http://www.topquadrant.com/products/TB_Composer.html. (Cited on page 81.)

[16] Unified Modeling Language: Infrastructure, version 2.1, . URL http://www.omg.org/docs/ptc/06-04-03.pdf. (Cited on pages 69 and 80.)

[17] Unified Modeling Language: Superstructure, version 2.1, . URL http://www.omg.org/docs/ptc/06-04-02.pdf. (Cited on page 69.)

[18] XML Path Language (XPath) Version 1.0. URL http://www.w3.org/TR/xpath/. (Cited on page 43.)

[19] Graphical Modeling Framework (GMF, Eclipse Modeling subproject), 2009. URL http://www.eclipse.org/gmf/. (Cited on page 33.)

[20] MetaEdit+ Workbench User's Guide, Version 4.5, 2009. URL http://www.metacase.com/support/45/manuals/mwb/Mw.html. (Cited on pages 18 and 32.)

[21] A. Agrawal, G. Karsai, and F. Shi. Graph transformations on domain-specific models. *Journal on Software and Systems Modeling*, 37:1–43, 2003. URL http://w3.isis.vanderbilt.edu/publications/archive/agrawal_a_11_0_2003_graph_tran.pdf. (Cited on page 60.)

[22] G. Antoniou and F. Van Harmelen. *A semantic web primer*. the MIT Press, 2004. (Cited on pages 16 and 78.)

[23] F. Baader, I. Horrocks, and U. Sattler. Description logics as ontology languages for the semantic web. *Mechanizing Mathemat-*

ical Reasoning, 2005. URL http://link.springer.com/chapter/10.1007/978-3-540-32254-2_14. (Cited on page 16.)

[24] G. Barzdins, R. Rikacovs, and M. Zviedris. Graphical Query Language as SPARQL Frontend. In *Local Proceedings of 13th East-European Conference (ADBIS 2009)*, pages 93–107. (Cited on page 97.)

[25] G. Barzdins, E. Liepins, M. Veilande, and M. Zviedris. Ontology Enabled Graphical Database Query Tool for End-Users. *Databases and Information Systems V*, pages 105–116, 2009. (Cited on page 97.)

[26] J. Bārzdiņš, A. Kalnins, E. Rencis, and S. Rikacovs. Model transformation languages and their implementation by bootstrapping method. *Pillars of computer science*, pages 130–145, 2008. (Cited on pages 15 and 61.)

[27] J. Bārzdiņš, E. Rencis, and S. Kozlovičs. The Transformation-Driven Architecture. In *Proc. of 8th OOPSLA Workshop on Domain-Specific Modeling. Nashville, USA*, pages 60–63, 2008. (Cited on pages 21 and 26.)

[28] J. Bārzdiņš, K. Čerāns, A. Kalniņš, M. Grasmanis, S. Kozlovičš, L. Lace, R. Liepiņš, E. Rencis, A. Sprogis, and A. Zariņš. Domain specific languages for business process management: a case study. In *The 9th OOPSLA Workshop on Domain-Specific Modelling, Orlando, USA, October*, pages 25–26, 2009. (Cited on page 6.)

[29] J. Bārzdiņš, K. Čerāns, S. Kozlovičs, E. Rencis, and A. Zariņš. A Graph Diagram Engine for the Transformation-Driven Architecture. *Proc. of 4th MDDAUI. Florida, USA*, pages 29–32, 2009. (Cited on pages 23 and 26.)

[30] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific american*, 284(5):28–37, 2001. (Cited on page 16.)

[31] S. Brockmans, R. Volz, A. Eberhart, and P. Löffler. Visual modeling of OWL DL ontologies using UML. In *The Semantic Web–ISWC 2004*, pages 198–213. Springer, 2004. (Cited on page 68.)

[32] P. c K ikusts and P. Ru v c evskis. Layout algorithms of graph-like diagrams for GRADE windows graphic editors. In *Graph Drawing*, pages 361–364. Springer, 1996. (Cited on page 26.)

[33] S. Cook, G. Jones, S. Kent, and A. C. Wills. *Domain-Specific Development with Visual Studio DSL Tools*. Pearson Education, May 2007. ISBN 9780132701556. (Cited on pages 14 and 18.)

[34] G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró. VIATRA-visual automated transformations for formal verification and validation of UML models. In *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on*, pages 267–270. IEEE, 2002. (Cited on page 60.)

[35] C. J. Date and H. Darwen. *A Guide to the SQL Standard*. Addison-Wesley Reading, 1987. URL `http://www.bortzmeyer.org/sql-standard.pdf`. (Cited on page 14.)

[36] D. Flanagan. *JavaScript*. O'Reilly, 1998. (Cited on page 39.)

[37] D. Flanagan and Y. Matsumoto. *The ruby programming language*. O'Reilly Media, Inc., 2008. URL `http://swblog.net/attachment/dn371.pdf`. (Cited on page 39.)

[38] K. Freivalds and P. Kikusts. Optimum layout adjustment supporting ordering constraints in graph-like diagram drawing. In *PROCEEDINGS-LATVIAN ACADEMY OF SCIENCES SECTION B*, pages 43–51, 2001. (Cited on page 26.)

[39] J. Friedl. *Mastering regular expressions*. O'Reilly Media, Inc., 2006. (Cited on page 14.)

[40] M. Gong, L. Scott, Y. Xiao, and R. Offen. A rapid development model for meta-CASE tool design. *Conceptual Modeling — ER '97*, 1331(Chapter 37):464–477, 1997. doi: 10.1007/3-540-63699-4{_}37. URL `http://www.springerlink.com/index/10.1007/3-540-63699-4_37`. (Cited on page 3.)

[41] S. Grimm and B. Motik. Closed world reasoning in the semantic web through epistemic operators. *by Bernardo Cuenca Grau,*

Ian Horrocks, Bijan Parsia, and Peter Patel-Schneider. Nov..(Cit. on p.), 2005. (Cited on pages 86 and 96.)

[42] R. C. Gronback. *Eclipse Modeling Project*. A Domain-Specific Language (DSL) Toolkit. Pearson Education, Mar. 2009. ISBN 9780321635198. (Cited on page 14.)

[43] P. Heim, T. Ertl, and J. Ziegler. Facet graphs: Complex semantic querying made easy. *The Semantic Web: Research and Applications*, pages 288–302, 2010. (Cited on page 97.)

[44] P. Hitzler and B. Parsia. Ontologies and rules. *Handbook on Ontologies*, pages 111–132, 2009. (Cited on page 95.)

[45] G. Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, 1992. (Cited on page 39.)

[46] R. Ierusalimschy, L. H. De Figueiredo, and W. C. Filho. Lua-an extensible extension language. *Softw., Pract. Exper.*, 26(6):635–652, 1996. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.72.3859&rep=rep1&type=pdf. (Cited on page 39.)

[47] R. Ierusalimschy, L. H. De Figueiredo, and W. Celes. Passing a language through the eye of a needle. *Communications of the ACM*, 54(7):38–43, 2011. (Cited on page 39.)

[48] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez. ATL: a QVT-like transformation language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 719–720. ACM, 2006. (Cited on pages 15 and 60.)

[49] K. Kaljurand and N. E. Fuchs. Verbalizing OWL in Attempto Controlled English. In *OWLED*, pages 5–20, 2007. (Cited on page 65.)

[50] A. Kalnins, J. Bārzdiņš, and E. Celms. Model transformation language MOLA. In *Model Driven Architecture*, pages 62–76. Springer, 2005. (Cited on pages 15 and 60.)

[51] S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Pr, 2008. ISBN 0470036664. URL http://www.worldcat.org/isbn/0470036664. (Cited on page 32.)

[52] E. Kendall, R. Bell, R. Burkhart, M. Dutra, and E. Wallace. Towards a Graphical Notation for OWL 2. In *OWLED 2009, OWL: Experiences and Directions. Sixth International Workshop, Chantilly, Virginia, USA 23-24 October 2009*, pages 1–8, 2009. (Cited on pages 76 and 77.)

[53] A. G. Kleppe, J. B. Warmer, and W. Bast. *MDA Explained*. The Model Driven Architecture : Practice and Promise. Addison-Wesley Professional, 2003. ISBN 9780321194428. (Cited on pages 14 and 15.)

[54] S. Kozlovičš. *The Transformation-Driven Architecture and its Graphical Presentation Engines*. University of Latvia, 2013. (Cited on pages 21 and 98.)

[55] S. Krivov, R. Williams, and F. Villa. Growl: A tool for visualization and editing of owl ontologies. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2):54–57, 2007. (Cited on page 68.)

[56] J. Lara and H. Vangheluwe. AToM 3: A Tool for Multi-formalism and Meta-modelling. *Fundamental approaches to software engineering*, pages 174–188, 2002. (Cited on page 15.)

[57] M. Lawley and J. Steel. Practical Declarative Model Transformation with Tefkat. *Satellite Events at the MoDELS 2005 Conference*, 3844(Chapter 15):139–150, 2006. doi: 10.1007/11663430{_}15. URL http://www.springerlink.com/index/10.1007/11663430_15. (Cited on page 60.)

[58] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *Computing Surveys (CSUR)*, 37 (4):316–344, Dec. 2005. doi: 10.1145/1118890.1118892. URL http:

//portal.acm.org/citation.cfm?doid=1118890.1118892. (Cited on pages 3 and 14.)

[59] G. Ng. Open vs Closed world, Rules vs Queries: Use cases from Industry. *OWL: Experiences and Directions (OWLED 2005), Calway, Ireland (November 2005)*, 2005. (Cited on page 84.)

[60] U. Nickel, J. Niere, and A. Zundorf. The FUJABA environment. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 742–745. ACM, 2000. ISBN 1-58113-206-9. doi: 10.1109/ICSE.2000.870485. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=870485. (Cited on pages 15 and 60.)

[61] E. Pietriga. Isaviz: a visual environment for browsing and authoring rdf models. In *Eleventh International World Wide Web Conference Developers Day*, 2002. (Cited on page 68.)

[62] D. C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39(2):25–31, Feb. 2006. doi: 10.1109/MC.2006.58. URL http://www.computer.org/csdl/mags/co/2006/02/r2025.html. (Cited on pages 3 and 16.)

[63] R. Shearer, B. Motik, and I. Horrocks. HermiT: A Highly-Efficient OWL Reasoner. In *Proc. of the 5th Int. Workshop on OWL: Experiences and Directions*, pages 91–105, Karlsruhe, Germany, 2008. (Cited on page 16.)

[64] E. Sirin and J. Tao. Towards Integrity Constraints in OWL. In *OWLED*, pages 36–51, 2009. (Cited on page 65.)

[65] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical owl-dl reasoner. *Web Semantics: science, services and agents on the World Wide Web*, 5(2):51–53, 2007. (Cited on page 16.)

[66] E. Sirin, B. Bulka, and M. Smith. Terp: Syntax for OWL-friendly SPARQL queries. *Proc. of OWLED*, 2010. (Cited on page 95.)

[67] A. Sprogis. *Domēnspecifisku Rīku Konfigurācijas Valoda un Tās Realizācija*. PhD thesis, University of Latvia, 2013. (Cited on pages 6 and 21.)

[68] A. Sprogis, R. Liepiņš, J. Bārzdiņš, K. Čerāns, S. Kozlovičs, L. Lace, E. Rencis, and A. Zariņš. GRAF: a Graphical Tool Building Framework. In *ECMFA 2010 Tools and Consultancy track*, pages 18–21, 2010. (Cited on page 3.)

[69] G. Taentzer. AGG: A tool environment for algebraic graph transformation. *Applications of Graph Transformations with Industrial Relevance*, pages 333–341, 2000. (Cited on page 60.)

[70] J. P. Tolvanen and M. Rossi. MetaEdit+: defining and using domain-specific modeling languages and code generators. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 92–93. ACM, 2003. URL http://dl.acm.org/citation.cfm?id=949365. (Cited on page 14.)

[71] D. Tsarkov and I. Horrocks. FaCT++ description logic reasoner: System description. In *Automated reasoning*, pages 292–297. Springer, 2006. URL http://owl.man.ac.uk/factplusplus/. (Cited on page 16.)

[72] G. VanRossum and F. L. Drake. *The Python Language Reference*. Python Software Foundation, 2010. URL http://cra.incraft.ru/py-intro/docs/docs-pdf/reference.pdf. (Cited on page 39.)

[73] S. A. White. Introduction to BPMN. *IBM Cooperation*, 2(0):0, 2004. (Cited on page 14.)

[74] E. D. Willink. UMLX: A graphical transformation language for MDA. In *Workshop on Model Driven Architecture: Foundations and Applications*, pages 13–24, 2003. URL http://eclipse.org/gmt/umlx/doc/UmlxFormalization/UmlxGraphicalLanguage.pdf. (Cited on page 60.)

[75] J. W. Yoder, F. Balaguer, and R. Johnson. Architecture and design of adaptive object-models. *ACM Sigplan Notices*, 36(12):50–60, 2001. (Cited on page 28.)

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*".

Final Version as of July 3, 2015 (version 1.0).