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

Sergejs Kozlovičs

the Transformation-Driven Architecture and its Graphical Presentation Engines

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, 2012

IEGULDĪJUMS TAVĀ NĀKOTNĒ

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

Supervisor:

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

Reviewers:

1) *Prof., Dr.Sc.Comp. Guntis Arnicāns*
   University of Latvia
2) *Prof., Dr.Sc.Ing. Oksana Ņikiforova*
   Riga Technical University
3) *Dr.Sc.Comp. Uģis Sarkans*
   The European Bioinformatics Institute, Cambridge, United Kingdom

The thesis will be defended at the public session of the Doctoral Committee of Computer Science, University of Latvia, at 15:00 on May 29, 2013, 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.

Abstract

In this thesis, a model-driven software architecture for interactive systems (systems consisting of multiple interoperating components) is proposed. This architecture, called the Transformation-Driven Architecture (TDA), advances the ideas of the Model-Driven Architecture (MDA). Unlike MDA, which uses models and model transformations at software development time, TDA uses them at runtime.

The following important TDA modules are also described in the thesis: TDA Kernel (provides the communication mechanism, the undo/redo mechanism, and the multi-repository mechanism), Environment Engine (a replaceable engine for using TDA in different environments), Dialog Engine (for specifying and displaying dialog windows), and Error Engine (for defining and visualizing error messages).

TDA proved its viability, when used as a foundation for building domain-specific tools.

Keywords: Model-Driven Architecture, Transformation-Driven Architecture, interactive systems, model-driven software development, domain-specific tools.

Acknowledgements

I gratefully acknowledge the help of my colleagues at Research Laboratory of System Modeling and Software Technologies of the Institute of Mathematics and Computer Science, University of Latvia (IMCS-UL):

- for personal conversations on model repositories (Sergejs Rikačovs, Kārlis Čerāns, Elīna Kalniņa, Mārtiņš Opmanis);
- for personal conversations on the use of quadratic optimization for Dialog Engine (Kārlis Freivalds);
- for personal conversations on the history of TDA (Audris Kalniņš, Kārlis Podnieks);
- for personal conversations on tool-building platforms (Audris Kalniņš, Elīna Kalniņa, Agris Šostaks);
- for personal conversations on transformation and mapping languages (Audris Kalniņš, Elīna Kalniņa, Agris Šostaks);
- for personal conversations on TDA use cases (Lelde Lāce, Mārtiņš Zviedris, Renārs Liepiņš, Artūrs Sproģis);
- for personal conversations on numerous TDA-related topics (all the colleagues);
- for efforts in organizing the development and maintenance of TDA and TDA-based tools (Pēteris Ručevskis).

I thank Andris Zariņš for developing Graph Diagram Engine and integrating it into TDA. I also thank him for developing the main window and the head engine for the tool-building platform GrTP and its TDA-based successor GRAF. This inspired me to introduce Environment Engine in TDA as well as the central TDA component called TDA Kernel.

I express my sincere gratitute to Prof. Kārlis Podnieks for organizing and supervising the work of the whole laboratory. His laborous work was essential for launching TDA. I also appreciate his encouragement for developing the next major version of TDA.

I gratefully acknowledge the help of Prof. Jānis Bārzdiņš, the supervisor of this thesis. His great experience along with practical and emotional support helped me to go on with the thesis and to finish it.

I thank Prof. Guntis Arnicāns, Prof. Jānis Bičevskis, and Assoc. Prof. Ģirts Karnītis (Faculty of Computing, University of Latvia) for their interest in TDA and for their comments, suggestions and objections that helped to improve TDA.

I and Prof. Bārzdiņš really appreciate the help of Prof. Juris Aivars (Faculty of Biology, University of Latvia), who gave us an insight into the structure of the human brain.

I appreciate the knowledge and experience I received during more than 10 years of studying at Faculty of Computing[1], University of Latvia. I also appreciate the professional work of its dean Juris Borzovs and other staff members.

I am grateful to my mother and sister for their support and patience during my work on this thesis.

I am grateful to all my friends, who supported me emotionally and played basketball with me (that helped my brain to work much better).

[1]formerly known as Department of Computer Science, Faculty of Physics and Mathematics

Contents

Introduction **8**

I The Transformation-Driven Architecture **21**

1 Theoretical Background **22**

1.1 Models and Abstraction . . . 22

Models, Metamodels, and Meta-Metamodels . . . 23

Technical Spaces . . . 24

Linguistic vs. Ontological Metamodelling . . . 28

1.2 Model Transformations . . . 33

Different Types of Model Transformations . . . 33

Transformation Languages . . . 34

Other Languages . . . 36

Semantic Reasoners as Model Transformations . . . 37

1.3 The World of "Model-Driven" . . . 38

Model-Driven Architecture (MDA) and Architecture-Driven Modernization (ADM) . . . 39

Model-Driven Engineering (MDE) . . . 39

Domain-Specific Modelling . . . 40

2 Motivation **41**

2.1 The MDA Idea . . . 41

2.2 MDA: Pros and Cons . . . 42

2.3 Why to Advance MDA Ideas? . . . 43

3 The Transformation-Driven Architecture **45**

3.1 The Outline View on TDA . . . 45

Engines . . . 45

Interface Metamodels . . . 46
Model Transformations . . . 47
Other Metamodels . . . 48
3.2 The Technical View on TDA . . . 48
3.3 The Communication Mechanism in TDA . . . 50
Communication Between Two Engines, or Between an Engine and a Transformation . . . 50
Communication Between Two Transformations . . . 52
An Example of Communication . . . 52

4 Creating Domain-Specific Tools 54
4.1 Domain-Specific Languages and Tools . . . 54
4.2 Existing Model-Driven Tool-Building Platforms . . . 56
4.3 Extensibility and Customization Obstacles . . . 60
4.4 TDA as a Foundation for DSL Tools . . . 61
4.5 The History of TDA . . . 63

II TDA Kernel And Engines 65

5 The Kernel Of TDA 66
5.1 The Fundamentals of RAAPI . . . 66
Organizing Meta-Levels into Quasi-Meta-Levels . . . 66
The Essentials of RAAPI . . . 69
5.2 Adapters . . . 71
Adapters for Repositories . . . 71
Adapters for Engines . . . 72
Adapters for Transformations . . . 73
5.3 The Basic Services of TDA Kernel . . . 73
5.4 Related Work . . . 75

6 The Multi-Repository Mechanism 76
6.1 Multiple Repositories as a Single Repository . . . 77
6.2 Packages as Mount Points . . . 77
6.3 Proxy References . . . 78
6.4 Manipulating the Packages . . . 79

6.5 Using Multiple Repositories in DSL Tool-Building . . . 81
6.6 Related Work . . . 82

7 The Undo/Redo Mechanism **83**
7.1 Basics Notions . . . 84
7.2 Undo Metamodel: the First Approximation . . . 85
7.3 Non-linear Undo: Multiple Undo History Streams and Dependencies Between Them . . . 87
7.4 External Actions and States . . . 89
7.5 Adding Support for Multiple Redo Branches . . . 92
7.6 Related Work . . . 94

8 Environment Engine **96**

9 Dialog Engine **103**
9.1 Dialog Engine Metamodel . . . 104
The Core of Dialog Engine Metamodel . . . 104
Tree Metamodel . . . 112
Table Metamodel . . . 113
9.2 Applying Quadratic Optimization . . . 115
The QMDC and the Extended QMDC Problems . . . 115
The Application of EQMDC . . . 116
9.3 Related Work . . . 122

10 Error Engine **125**
10.1 Error Handling and the Quasi-Ontological Meta-Meta Level . . . 126
10.2 Error Meta-Metamodel for Describing Errors . . . 128
ErrorClass . . . 129
WaitingErrorClass . . . 131
NonWaitingErrorClass . . . 131
AccumulatingErrorClass . . . 132
10.3 Handling Errors . . . 133
10.4 Related Work . . . 135

Discussion **137**

Conclusion **142**

Bibliography **145**

A Repository Access Application Programming Interface (RAAPI) Documentation **167**

A.1 IRepository Interface Reference 167

Detailed Description 168

A.2 RAAPI Interface Reference 168

Detailed Description 170

Member Function Documentation 170

A.3 IRepositoryManagement Interface Reference 192

Detailed Description 192

Member Function Documentation 193

* * *

Note on the usage of the pronouns "we" and "I" in this thesis. Depending on the context, the pronoun "we" will mean "humans", "computer scientists", or "the author and the reader". In no case "we" will mean "I and colleagues". To emphasize the personal contribution and to avoid awkward sentences, the pronoun "I" will be used. Arguments for using "I" can be found in "Handbook of Technical Writing" [16, pp. 405–406].

Note on the language. Mainly, I use British English in my prose. For certain words I make the choice in favour of their forms traditionally used in Computer Science (like "program" and "dialog"). Whenever I quote other authors or refer to widely-accepted terms, I use original syntax (usually, in American English). This explains why sometimes the word "modelling" is written as "modeling".

* * *

Introduction

Earlier we had written programs in machine code,
then in assembler.
Afterwards translators appeared,
then tools.
And now we have tools for creating tools.
What will be the next?
—prof. J. Borzovs at a seminar for doctoral students of Computer Science,
University of Latvia, December 2008

— Dad, what computers are for?
— Computers, son, are intended to solve problems
that did not exist before invention of computers.
(humour)

From tally sticks and abacus, the history of computing leads us to modern programmable computers intended to aid the humanity in solving numerous tasks. However, in order the computer could solve the task, we have to overcome the gap between the idea in the human mind and its implementation in machine code. That is why programmers are required.

To automate the programmers' job, assembly languages were invented in the late 1940-ties. Instead of writing the machine code directly, a programmer could express the idea at a bit higher level of abstraction, and the assembler could automatically generate the machine code. This increased the productivity of programmers, but not very much. In the late 1950-ties so called third generation programming languages (3GLs) appeared. According to SPR 2006 [17, as cited by [18]], 3GLs increased the productivity of programmers by 450%. Other types of languages were introduced later, which include

special-purpose languages (e.g., SQL, MATLAB, etc.) as well as functional (e.g., Haskell, ML, F#) and declarative languages (e.g., Prolog, Lisp). Besides languages, there are a number of software libraries, which factor out common recurring tasks.

Still, all these attempts to improve the productivity of the programmers cannot reach the horizon, since the horizon moves away. The increasing power of computers as well as their proliferation create new problems to be solved by computers, the problems that did not exist earlier. The demand for software increases more drastically than the productivity of programmers (compare: 100 times versus 2 times in the period 1965–1985 [19, as cited by [20]]). Furthermore, programmers today need to solve far more complex tasks than the tasks some fifty years ago. V. Parondzhanov even uses the term "intellectual terrorism" to describe the pressure the information age puts on programmers [20].

Besides the horizon moving away, there seems to be a kind of "force of friction" associated with software. Frederick P. Brooks in his famous "No silver bullet" paper (1987) [21] pointed that besides accidental difficulties in software development (artificial barriers, which can eventually be overcome) there are also essential difficulties "inherent in the nature of software" . He claimed:

> "Not only are there no silver bullets now in view, the very nature of software makes it unlikely that there will be any–no inventions that will do for software productivity, reliability, and simplicity what electronics, transistors, and large-scale integration did for computer hardware. We cannot expect ever to see twofold gains every two years[2]."

To raise the productivity of programmers and to help them to deal with an ever-growing complexity of the software systems being built we should search for concepts that are close to our mind, the concepts that can be used to raise the level of abstraction. It is noteworthy that all the techniques used by programmers mentioned in the second paragraph are based on the abstraction principle (the principle of raising the level of abstraction).

Two inter-related concepts that are close to our mind are models and transformations. Models (in a broad sense) have been used by humans all the time. Probably, models are at the root of our cognition [22, pp. 105–106]. Like models, transformations (in a broad sense) are also all around us. A caterpillar transforms into a butterfly [23]. A mustard

[2] Here Brooks alludes to Moore's law, an observation that the number of transistors in computers doubles every two years, — *a note by S. Kozlovičs.*

grain, which is a very small seed, while growing, transforms into a big tree [24]. A fertilized ovum transforms to an embryo, and then to a newborn infant [25]. Elīna Kalniņa noticed that "the occurrence and the way of transformation is predefined somewhere in nature, most probably in DNA" [26].

Hereinafter I focus on models and transformations that can be processed by computers. That means that these models and transformations have to be formally defined.

In 2001, Object Management Group (OMG) launched a widely-recognizable initiative called the Model-Driven Architecture, MDA [27, 28, 29]. MDA is a software development approach that uses models and model transformations. MDA is a starting point for and a subset of the research area called Model-Driven Engineering, MDE.[3] In MDA and MDE, models are not just technical sketches — they are first-class citizens, which shift the traditional object-oriented paradigm with the main principle "Everything is an object" to a model-driven paradigm with the main principle "Everything is a model" [30].

Although there are certain use cases, where the principles of MDA have been successfully applied, MDA faces certain objections now (see Chapter 2). Even in the MDE area a kind of a crisis is being observed. Jean Bézivin, a famous professor and researcher in the MDE area, thinks that "if we measure success in terms of adoption by the industry and the start of large modeling projects, MDE has reached a standstill" (from notes by Jean-Jacques Dubray [31]). During my work on this thesis, Bézivin travelled with the talk entitled "Why did MDE miss the boat?", where he pointed to various difficulties that MDE is facing now [32, 31]. Nevertheless, he thinks this is not the end of MDE, but only a "temporary failure".

If there indeed is a failure within MDE, then I would agree with Bézivin that this failure is *temporary*. We need new ideas leading to new research directions, which could eventually help to realize the benefits of models and to put them into mainstream. The Transformation-Driven Architecture, TDA, is such a new idea. Based on the MDA foundation, TDA advances some MDA ideas and takes a new direction in MDE. In Chapter 2 I will show that in MDA it is difficult to describe the dynamics of interactive systems (systems, consisting of multiple interoperating components). The main purpose of TDA is to extend MDA ideas and provide a software architecture that is suitable for describing such systems. An important application of TDA is development of domain-specific tools, which form a considerable subset of interactive systems. TDA brings certain additional

[3] There are also other abbreviations related to MDE (MDSD, MDD, DSM, etc.). Section 1.3 (from Chapter 1) clarifies them.

benefits to interactive systems, e.g., automatic undo/redo mechanism and the ability to use different technologies for different components of a system. Unlike MDA, which refers to the software development process, TDA is a real software architecture. Besides, in TDA models and transformations are used at runtime, although TDA can replace MDA at software development time as well. TDA is able to eliminate also many of the MDE difficulties mentioned by Bézivin in his talk. I believe that TDA is an important step that can help MDE to reach the missed boat.

* * *

The introduction is now being continued by formal sections.

The Aim and Tasks of the Research

The main aim of the thesis is:

- to propose a software architecture that would simplify development of complex interactive systems. By using formal models and their transformations at runtime, the proposed architecture can raise the level of abstraction, at which the business logic of a system is described. This would allow the productivity of developers to increase.

Other goals of the thesis:

- To identify problems that are commonly found in interactive systems and related to internal communication and graphical user interface; to offer solutions that would fit into the overall architecture.
- To motivate the use of TDA; to describe existing TDA usages as well as the potential of TDA.

Tasks for reaching the goals:

- To study model-driven and related technologies.
- To describe the Transformation-Driven Architecture and its parts (models, transformations, and system components).
- To formalize the communication mechanism of TDA.

- To formalize TDA abstraction layers, which allow TDA to use different types of models, transformations, and components (e.g., components written in different programming languages), when building an interactive system.
- To describe and formalize important graphical components that ensure the communication between the user and TDA.
- To offer solutions for the most essential TDA-level mechanisms: the undo/redo mechanism and the multi-repository mechanism.
- To explore the state-of-the-art in the area of domain-specific tool building (domain-specific tools form an important subset of interactive systems); to study the principles and technologies behind tool-building platforms.
- To demonstrate TDA usability for building domain-specific tools. To compare TDA with other existing tool-building platforms and to explore TDA advantages.

The Clause and Research Directions

The main clause of the thesis is:

> It is possible to define and formalize a software architecture for interactive systems, where components are described by models, while system dynamics and business logic are described by model transformations. When describing business logic, we can abstract away from internal implementation details of components. This raises the level of abstraction, at which the system is being described, and facilitates the reuse of third-party components. As a result, the productivity of software developers increases.

Research directions:

- What are the main parts of such an architecture?
- What abstraction layers are needed to be able to work with different model repositories, transformation languages, and different kinds of components?
- Which common problems can be solved in a universal way taking into a consideration capabilities and restrictions of the overall architecture?

Research Methods Used

One of the main research methods used is *logical deduction*. It helped to choose the most suitable solution from various alternatives.

When searching for solutions, *the priority was given to solutions that were easier to use* (as contrasted to solutions that were easier to implement). For instance, the implementation of Dialog Engine uses a complex algorithm for quadratic optimization. However, once implemented, this solution lets the developer not to specify coordinates of dialog components, since these coordinates can be computed automatically.

When multiple alternatives were possible, the priority was given to the solutions that fit into the overall architecture. For instance, the undo/redo mechanism can be implemented for a transformation language. However, this approach wouldn't work for components that are not transformations. A better solution is to implement the undo/redo mechanism as a TDA-level mechanism.

When working on TDA specification, I used generally accepted *metamodelling and model formalization techniques*.

Several tools that have been developed using TDA as a foundation can be thought of as a *proof-by-demonstration* of TDA practical applicability. Those tools have been used to verify TDA principles *experimentally*. The received feedback helped to improve TDA specification as well as to make TDA more suitable for practical needs.

The thesis also contains a *comparative analysis* of existing TDA-related solutions.

The Main Results of the Thesis

The most significant result of the thesis is:

- the specification of the Transformation-Driven Architecture, TDA. It formally defines TDA components and their interoperability. Units that implement auxiliary functionality (graphical presentations and certain services) are called **engines**. Although when implementing engines different programming languages and specific technologies can be used, all engines are described in a unified platform-independent way by means of **interface metamodels**. The business logic is described by model transformations, which can be written in different transformation languages. TDA models and metamodels are stored in a **model repository**. **TDA Kernel** is the central TDA component. Its specification defines certain abstraction layers (uni-

versal interfaces) that allow TDA Kernel to operate with components of different types (e.g., engines written in different programming languages, or various model repositories).

Other important results are:

- A universal (but adjustable) TDA-based solution for undo/redo.
- The TDA multi-repository mechanism, which allows TDA to work with multiple model repositories simultaneously. Some repositories may be virtual; they can be used, for example, to implement views on other repositories.
- The concept of Environment Engine, which is a universal means for adapting TDA to different platforms and for plugging-in TDA to other programs. Environment Engine has been described by means of Environment Engine Metamodel.
- The concept of Dialog Engine, which is a universal model-based means for defining user interface dialog windows and displaying them on the screen. Dialog Engine has been described by means of Dialog Metamodel.
- An algorithm for computing coordinates of dialog window components. The algorithm transforms a Dialog Metamodel instance to the quadratic optimization problem instance.
- The concept of Error Engine, which is a universal model-based means for visualizing and grouping error messages. Error Engine has been described by means of a meta-metamodel[4] and a base metamodel.
- A demonstration of TDA practical applicability for building domain-specific tools.

Validation of the Results

Applying the results in practice

TDA ideas were approbated in several experimental and semi-industrial domain-specific tools developed at IMCS-UL. For example, by means of a TDA-based tool-building platform GRAF [7, 4] the following tools were built: ProMod and BiLingva (business process management tools), GradeTwo (a UML tool), ViziQuer (a graphical semantic data query

[4] Error Engine uses the third ontological meta-level, see Section 1.1 .

tool), and OwlGrEd (a graphical OWL ontology editor) [3, 33, 34, 35]. ProMod and BiLingva are being used in production environment in Latvia. The OWLGrEd tool is a free tool, which is being used worldwide.

My publications on the topic of the thesis

I have 14 refereed publications related to the topic of the thesis; 10 of them are published in international editions/conference proceedings, while the other four are published in local editions (Scientific Papers, University of Latvia). Among all of the publications, four are published in editions with recognized citation index (SCOPUS, ACM). The main principles of TDA are described in a journal paper published in two languages.

The following table describes my personal contribution to each of the publications (publications are ordered by year).

Table 1: Author's publications on the topic of the thesis.

Authors	**Publication**	**My contribution**	**Description of the contribution**
J. Barzdins, S. Kozlovics, E. Rencis	"The Transformation-Driven Architecture," in Proceedings of DSM'08 Workshop of OOPSLA 2008, Nashville, Tennessee, USA, 2008, pp. 60–63. [1]	70%	• Concretization of the initial TDA idea • A description of the initial TDA communication mechanism • A description of the initial idea for the undo/redo mechanism
J. Barzdins, K. Cerans, S. Kozlovics, E. Rencis, A. Zarins	"A Graph Diagram Engine for the Transformation-Driven Architecture," in Proceedings of MDDAUI 2009 Workshop of International Conference on Intelligent User Interfaces 2009, Sanibel Island, Florida, USA, 2009, pp. 29–32. [2]	20%	• A description of the communication between TDA and Graph Diagram Engine • Participation in the development of metamodels
J. Barzdins, K. Cerans, A. Kalnins, M. Grasmanis, S. Kozlovics, L. Lace, R. Liepins, E. Rencis, A. Sprogis, A. Zarins	"Domain specific languages for business process management: a case study," in Proceedings of DSM'09 Workshop of OOPSLA 2009, Florida, USA, 2009, pp. 34–40. [3]	10%	• Adapting TDA for industrial applications • *Word* Engine development

Authors	Publication	My contri-bution	Description of the contribution
J. Barzdins, K. Cerans, S. Kozlovics, L. Lace, R. Liepins, E. Rencis, A. Sprogis, A. Zarins	"MDE-based Graphical Tool Building Framework," in Scientific Papers, University of Latvia, vol. 756, 2010, pp. 121–138. [4]	10%	• Using TDA in graphical tool building
J. Barzdins, K. Cerans, S. Kozlovics, E. Rencis, A. Zarins	"A Graph Diagram Engine for the Transformation-Driven Architecture," in Scientific Papers, University of Latvia, vol. 756, 2010, pp. 139–149. [5]	15%	• A description of the communication between TDA and Graph Diagram Engine • Participation in the development of metamodels
S. Kozlovics	"A Dialog Engine Metamodel for the Transformation-Driven Architecture," in Scientific Papers, University of Latvia, vol. 756, 2010, pp. 151–170. [6]	100%	• Developing Dialog Engine Metamodel and describing its semantics • Dialog Engine Metamodel extensions • Comparing Dialog Engine Metamodel and other ways of specifying dialogs
A. Sprogis, R. Liepins, J. Barzdins, K. Cerans, S. Kozlovics, L. Lace, E. Rencis, A. Zarins	"GRAF: a graphical tool building framework," in Proceedings of the Tools and Consultancy Track of ECMFA 2010, S. Gerard, Ed. CEA LIST, 2010. [7]	10% (poster: 90%)	• Using TDA in graphical tool building
S. Kozlovics, E. Rencis, S. Rikacovs, K. Cerans	"Universal UNDO mechanism for the Transformation-Driven Architecture," in Proceeding of the Ninth International Baltic Conference, DB&IS 2010. Riga, Latvia: University of Latvia Press, 2010, pp. 325–340. [8]	70%	• Participation in the development of base ideas of the undo/redo mechanism • The extensions of the undo/redo mechanism

Authors	Publication	My contribution	Description of the contribution
S. Kozlovics, E. Rencis, S. Rikacovs, K. Cerans	"A kernel-level UNDO/REDO mechanism for the Transformation-Driven Architecture," in Proceedings of the 2011 conference on Databases and Information Systems VI: Selected Papers from the Ninth International Baltic Conference, DB&IS 2010, ser. Frontiers in Artificial Intelligence and Applications, vol. 224. Amsterdam, The Netherlands: IOS Press, 2011, pp. 80–93. [9] [ACM]	80%	Aforementioned + • introducing TDA Kernel; its relation to the undo/redo mechanism
S. Kozlovics	"A universal model-based solution for describing and handling errors," in Perspectives in Business Informatics Research, ser. Lecture Notes in Business Information Processing, vol. 90. Springer Berlin Heidelberg, 2011, pp. 190–203. [10] [SCOPUS]	100%	• Error Engine Meta-Metamodel and its semantics • Automated error handling
E. Rencis, J. Barzdins, S. Kozlovics	"Towards open graphical tool-building framework," in Scientific Journal of Riga Technical University (Special issue for the 10th International Conference on Perspectives in Business Informatics Research), ser. Computer Science: Applied Computer Systems, vol. 46, no. 5. Riga, Latvia: RTU Press, 2011, pp. 80–87. [11]	30%	• Participation in the development of ideas of the tool-building approach based on the metamodel specialization concept
S. Kozlovics	"Calculating The Layout For Dialog Windows Specified As Models," in Scientific Papers, University of Latvia, vol. 787, 2012, pp. 106–124. [12]	100%	• The quadratic optimization based dialog window layout algorithm
S. Kozlovics	"The orchestra of multiple model repositories," in SOFSEM 2013: Theory and Practice of Computer Science, ser. Lecture Notes in Computer Science, vol. 7741. Springer Berlin Heidelberg, 2013, pp. 503–514. [13] [SCOPUS]	100%	• The main idea and a description of the multi-repository mechanism

Authors	Publication	My contribution	Description of the contribution
С. Козлович, Я. Барздиньш (S. Kozlovics, J. Barzdins)	"Управляемая трансформациями архитектура для интерактивных систем," Автоматика и вычислительная техника, т. 47, №1, 2013. С. 39–52. [14] The English translation: "The Transformation-Driven Architecture for interactive systems," Automatic Control and Computer Sciences, vol. 47, no. 1/2013, Allerton Press, Inc., 2013, pp. 28–37. [15] [SCOPUS]	90%	• A description of the Transformation-Driven Architecture in its new variant • Using Transformation-Driven Architecture for developing interactive systems

Table 2 links the chapters of the thesis with the corresponding publications (the first two chapters are introductory chapters, thus, they do not appear in the table).

Table 2: Mapping the main results to the corresponding publications.

Chapter	Main results and the corresponding publications
3	The concept of the Transformation-Driven Architecture; the TDA communication mechanism [1, 14, 15, 2, 5]
4	TDA use cases [3, 7, 4]
5	TDA Kernel and its abstraction layers [14, 15, 9, 13, 11]
6	The TDA Undo/Redo Mechanism [8, 9]
7	The TDA Multi-Repository Mechanism [13]
8	Environment Engine and its metamodel [14, 15]
9	Dialog Engine, its metamodel, and an algorithm for laying out dialog components by means of quadratic optimization [6, 12]
10	Error Engine and its metamodel [10]

Presentations at scientific conferences

The results of the thesis have been presented at the following international conferences[5]:

- S. Kozlovics, J. Barzdins and E. Rencis:
  "The Transformation-Driven Architecture"
  The 8th OOPSLA Workshop on Domain-Specific Modeling, Nashville, TN, USA, October 19-20, 2008
- S. Kozlovics*, E. Rencis, S. Rikacovs, K. Cerans:
  "Universal UNDO Mechanism for the Transformation-Driven Architecture"
  Ninth Conference on Databases and Information Systems, 2010, Riga, Latvia, July 5-7, 2010
- S. Kozlovics*:
  "A Universal Model-Based Solution for Describing and Handling Errors"
  10th International Conference on Perspectives in Business Informatics Research, Riga, Latvia, October 6-8, 2011

[5]my presentations are marked with the "*" symbol

- J. Barzdins, K. Cerans, M. Grasmanis, A. Kalnins, S. Kozlovics, L. Lace, R. Liepins, E. Rencis, A. Sprogis and A. Zarins:
"Domain Specific Languages for Business Process Management: a Case Study"
The 9th OOPSLA Workshop on Domain-Specific Modeling, 25-26 October, 2009

- J. Barzdins, K. Cerans, S. Kozlovics, E. Rencis, A. Zarins:
"A Graph Diagram Engine for the Transformation-Driven Architecture"
Fourth International Workshop on Model Driven Development of Advanced User Interfaces (MDDAUI 2009), Sanibel Island, Florida, USA, February 8th, 2009

- E. Rencis, J. Barzdins and S. Kozlovics:
"Towards Open Graphical Tool-Building Framework"
10th International Conference on Perspectives in Business Informatics Research, Riga, Latvia, October 6-8, 2011

- A. Sprogis, R. Liepins, J. Barzdins, K. Cerans, S. Kozlovics, L. Lace, E. Rencis and A. Zarins:
"GRAF: a Graphical Tool Building Framework"
6th European Conference on Modelling Foundations and Applications (ECMFA 2010), Paris, France, June 15-18, 2010

- S. Kozlovics*:
"The Orchestra of Multiple Model Repositories"
39th International Conference on Current Trends in Theory and Practice of Computer Science, Špindlerův Mlýn, Czech Republic, January 26–31, 2013

I was the main author of the poster "GRAF: a graphical tool-building framework", which was presented at the ECMFA 2010 conference in Paris, France.

Besides, I reported on the results of the thesis at annual conferences of University of Latvia in 2008, 2009, 2011, and 2013.

The Structure of the Thesis

The thesis is divided into two parts:

- Part I presents the Transformation-Driven Architecture, TDA. TDA is the core of this thesis.

- Part II describes certain TDA components in detail — a central component called TDA Kernel, which implements the core functionality of TDA, and three auxiliary components (called *engines*) that implement essential graphical presentations and services.

Part I starts with Chapter 1, which sets theoretical background for later chapters of the thesis. In Chapter 2, MDA ideas are used to set up the foundation for TDA. Problematic

issues to be solved by TDA are also identified in this chapter. Chapter 3 presents the overall view on TDA. Chapter 4 describes domain-specific tool building as an important use case for TDA, where significant results have been achieved.

Part II starts with Chapters 5–7, which present TDA Kernel in detail. These chapters describe how TDA Kernel:

- ensures communication between TDA components, which can be written using different technologies and different programming languages (Chapter 5);
- is able to work with different model repositories simultaneously (Chapter 6);
- implements the undo/redo mechanism (Chapter 7).

The next three chapters (Chapters 8–10) in Part II present three essential TDA engines authored by me. They are:

- *Environment Engine*, which manages the main application window and takes away environment-specific aspects from TDA (Chapter 8);
- *Dialog Engine* capable of displaying graphical dialog windows generated at runtime (Chapter 9);
- *Error Engine*, which is a universal engine for displaying errors to the user (Chapter 10).

The Discussion chapter (on page 137) underlines the significance of TDA, explains its limitations, and presents an interesting relationship between TDA and the architecture of the human brain. The Conclusion summarizes the results.

Part I

The Transformation-Driven Architecture

Chapter 1

Theoretical Background

This chapter sets the background for the thesis: it introduces fundamental definitions used throughout the monograph and identifies the research areas related to the main topic.

1.1 Models and Abstraction

In his article in the "Software and Systems Modeling" journal Jochen Ludewig pointed that although we, humans, use models all the time, it is difficult to define what a model is. "Endless discussions have proven that there is no consistent common understanding of models", — he writes [36]. For the purpose of this thesis models must have one essential property, namely, the ability to be automatically processed by a computer. Thus, I will use the following definition of a model found in the book by A. Kleppe et al. [29]. The definition consists of the two parts:

A model is a description of (part of) a system written in a well-defined language.

A well-defined language is a language with well-defined form (syntax) and meaning (semantics), which is suitable for automated interpretation by a computer.

For other possible definitions the reader can refer to the thesis by Elīna Kalniņa, where she has done extensive work on investigating various sources, while seeking for a model definition [26].

Another important notion related to models is the notion of abstraction. Although it is a general term, in Computer Science it has a quite concrete sense. A Dictionary of Computing by Oxford University Press gives the following definition:

Abstraction [is] the principle of ignoring those aspects of a subject that are not relevant to the current purpose in order to concentrate solely on those that are. The

application of this principle is essential in the development and understanding of all forms of computer system [37].

Thus, abstraction removes irrelevant details. The number of details is related to the notion of abstraction level. The following definitions can be found in the book by A. Kleppe et al. [29]:

Abstraction Level [is] the inverse of the (relative) amount of details that are in a model.

High Abstraction Level [means] a (relative) low amount of details.

Low Abstraction Level [means] a (relative) high amount of details.

In this thesis, "higher level of abstraction" will usually mean "closer to human mind", while "lower level of abstraction" will mean "closer to computer hardware".

Although mainly I will use the definition of abstraction from above, there is another (general) meaning of that term, which is also used in Computer Science, and which is particularly important, when we speak about models. The Encyclopædia Britannica describes it as follows:

Abstraction [is] the cognitive process of isolating, or "abstracting," a common feature or relationship observed in a number of things, or the product of such a process [38].

Abstraction in this sense appears, for instance, in object-oriented programming (OOP): a class defines common features for its objects and a superclass factors out common features of its subclasses.

A model and abstraction are the two main notions used in this thesis.

Models, Metamodels, and Meta-Metamodels

In the definition of a model from above a well-defined language is a language for describing models. That language can be considered a model by its own. Thus, it is a model for describing other models, or a **metamodel**. Saying that a metamodel MM describes a model M is the same as saying that M **conforms to** MM.

Example A. The Java grammar can be considered a metamodel for describing models in the form of Java programs. Every syntactically correct Java program conforms to the Java grammar.

A metamodel is also a model, thus, it is also written in some well-defined language. We can think of that language as of a metamodel for describing metamodels, or a **meta-metamodel**.

> *Example B.* Extended Backus-Naur Form (EBNF), a notation for context-free grammars, can be considered a meta-metamodel for grammars [39]. Grammars themselves can be considered metamodels (as in Example A) that conform to EBNF.

Although we can continue to add other meta-s, in many cases a meta-metamodel is able to describe itself. This observation is called the **three-level conjecture**, where the three meta-levels are the model level (M1), the metamodel level (M2), and the meta-metamodel level (M3); the fourth level is not needed, since it equals to M3. These *levels* are sometimes called *layers* [40].

> *Example C.* A Java program lies at Level M1, the Java grammar lies at M2, and EBNF lies at M3. EBNF can be described in EBNF.

Technical Spaces

In 2002 I. Kurtev, J. Bézivin and M. Aksit have made an observation that the three-level conjecture can be applied to numerous technologies. The concept of **technical space** was introduced to identify such technologies [41, 40]. In a technical space, usually a fixed meta-metamodel at Level M3 provides a basis for defining metamodels at Level M2, which, in their turn, define models at Level M1. This chain can be extended by saying that models at Level M1 define objects at Level M0, which are real-world objects or runtime objects of a system being modelled. For instance, UML (Unified Modeling Language) Infrastructure is based on such Four-level Metamodel Hierarchy [42, Sections 7.9–7.12]. At the same time, the border between M1 and M0 is where modelling either finishes (when M1 elements describe real objects), or changes its form (when, for example, M1 elements describe data in a database representing real world objects). In accordance with J. Bézivin and I. Kurtev, I will consider Level M0 lying outside the concept of technical space [40].

Having three meta-levels, a technical space must also define the “conforms to” relation between them. This relation is defined differently in different technical spaces, but usually there are tools to check this relation. For instance, in the EBNF technical space (from

Example B above), the “conforms to” relation between M1 and M2 is checked by language parsers.

Technical spaces are not just three meta-levels and the “conforms to” relation. J. Bézivin and I. Kurtev say:

> “A technical space is a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities. Apparently, there are human related components in this definition since most technologies have emerged in a given community that has knowledge, performs research, and even may have dedicated conferences. In addition, a technology allows creation and manipulation of artifacts.” [40]

The motivation to investigate different technical spaces (TS’s) and to support them is driven by the following considerations:

- One TS can be more suitable and more convenient for the given purpose than another. That resembles how one programming language can be more suitable for a particular purpose than another.
- A person can be more familiar with (i.e., have skills and knowledge in) one TS than with another. If the efforts to study a new TS are big enough, it may be reasonable to stay in a more familiar TS.
- A capability not available in a desired TS can be borrowed from another TS, which implements that capability. This encourages “more cooperation than competition among alternative technologies” [40].

Technical spaces need to store models somewhere. The term **model repository** (or simply: **repository**) will denote a store, where models can be saved. It can be a specific data store, a simple text file, or an XML-file[1]. Different types of repositories (even within one TS) have to be accessed differently. In some cases a repository is accessed by means of a specific parser, while in other cases — via certain API (Application Programming Interface).

Table 1.1 summarizes my research on technical spaces and their repositories.

[1]XML = Extensible Markup Language; XML-files are text files formatted according to the XML standard [43]

Table 1.1: Technical spaces (TS's) and their repositories.

MOF TS	*Characteristics:*	Models are graphs with attributed nodes and labelled edges. Inspired by CDIF[2] [44] and IRDS[3] [45], the MOF[4] standard [46] by OMG is central in this TS. MOF consists of the two main variants: Essential MOF (EMOF) and Complete MOF (CMOF). OMG is working also on Semantic MOF (SMOF), which will contain features not available in MOF TS, but available in the RDF/OWL TS (see below) [47]. The *de facto* standard in this TS, however, is ECore from EMF[5] [48, 49][6]. ECore implements MOF concepts in Java.. There is also the KM3[7] language for defining MOF-like metamodels. I combine all these standards into a single TS called MOF TS.
	Meta-levels:	model --→ metamodel --→ meta-metamodel (MOF, ECore, KM3)
	Repositories:	EMF/Ecore [48, 49]; Enhanced Model Repository [50]; NetBeans MDR[8] [51]; MOF 2 for Java [52]; MetaMart Metadata Repository [53]; CDO[9] [54]; JR[10] [55]
XML TS	*Characteristics:*	Models are trees with attributed nodes.
	Meta-levels:	XML-file [43]--→XML schema [56, 57] --→ XML meta-schema (XSD.xsd)[11]
	Repositories:	XML files (there are numerous libraries for parsing/saving XML files)
Microsoft DSL Tools TS	*Characteristics:*	Similar to MOF, but classes have to be arranged into a tree by means of compositions. Relationships may act as classes, and it is possible to define inheritances between relationships.
	Meta-levels:	model --→domain model (metamodel) --→ implicit meta-metamodel, which can be reified [40]

[2]CASE Data Interchange Format
[3]Information Resource Dictionary System
[4]Meta-Object Facility
[5]Eclipse Modeling Framework
[6]At the Transformation-Tool Contest (TTC) event, a satellite to TOOLS conference, solutions had to accept ECore models as input and produce ECore models as output.
[7]Kernel Meta Meta Model
[8]Meta Data Repository
[9]Connected Data Objects
[10]New Repository ("Jaunais Repozitorijs" in Latvian)
[11]XSD stands for XML Schema Definition

	Repositories:	*In-Memory Store* (models are serialized as customizable XMLs) [58, p. 89]
Grammarware TS	*Characteristics:*	A model is a string, which can be parsed into an abstract syntax tree.
	Meta-levels:	text/string --→ grammar --→ EBNF (or similar meta-grammar)
	Repositories:	usually text files (tools such as *bison* or *javacc* can be used to generate parsers)
GOPPRR (MetaEdit+) TS	*Characteristics:*	Similar to MOF. N-ary relationships between concepts are possible. Relationships and their ends (roles) may have properties associated with them.
	Meta-levels:	model --→ metamodel --→ GOPPRR[12]
	Repositories:	a proprietary GOPPRR repository [59, 60]
RDF/OWL TS	*Characteristics:*	This TS consists of knowledge representation systems, where all data are encoded in triples (subject, predicate, object). These triples form a graph (subjects and objects are nodes, while predicates are edges). Elements are identified by URIs (uniform resource identifiers).
	Meta-levels:	RDF[13]/OWL[14] individuals[61, 62, 63]--→ RDF vocabulary/OWL ontology--→ RDFS[15]/some OWL variant[16]
	Repositories:	Sesame [65]; Virtuoso [66]; OWLIM [67]; OWL API [68]; Apache Jena [69]; JR [55]; AllegroGraph [70]
Relational Database TS	*Characteristics:*	A classical way to encode entities and relationships by means of tables. No support for generalizations (although they can be simulated).
	Meta-levels:	database rows --→ database schema (ER-model) --→ system tables for storing database schemas

[12] Graph-Object-Property-Port-Role-Relationship

[13] Resource Description Framework

[14] Web Ontology Language (the first two initial letters are swapped)

[15] RDF Schema, a language, which extends RDF and permits describing taxonomies of classes as RDF vocabularies

[16] There are the following OWL variants: OWL Lite, OWL DL, OWL Full, OWL 2 (direct semantics and RDF-based semantics), OWL 2 EL, OWL 2 QL, OWL 2 RL (EL/QL/RL can be combined). They differ by expressive power, decidability, and computational complexity for decidable variants [64]. Besides, pure RDF and OWL Full permit having multiple meta-levels and mixing them.

	Repositories:	numerous database management systems from SQLite to ORACLE; the LINQ[17] technology [71]; ORM[18] technologies [72, 73]
Typed Attributed Graphs (TAG) TS	*Characteristics:*	Typed graphs, where nodes and edges may have attributes.
	Meta-levels:	graph ⇢ graph schema ⇢ typed attributed graph definition
	Repositories:	JGraLab [74] and others
Note. The grouping of different technologies into technical spaces is not strict. Some of TS's are similar (e.g., I combined OMG/MDA TS and EMF TS into MOF TS; I also combined RDF and all OWL variations into RDF/OWL TS). Furthermore, in some cases one TS can be used from another (or, they can be bridged) [75, 76, 77, 78, 47, 79]. This makes the border between TS's even more vague.		

Linguistic vs. Ontological Metamodelling

While the "conforms to" relation holds between a model and its metamodel, there is another relation called "**instance of**", which holds between model elements (objects) and the corresponding metamodel elements (types, or classes)[19]. The "conforms to" and "instance of" relations go between two adjacent meta-levels.

With a help of the "instance of" relation it is quite easy to demonstrate, how the need for more than three meta-levels (offered by technical spaces) arises. Douglas Hofstadter in his famous book "Gödel, Escher, Bach" (1979) mentions the following example [80]:

(Level $n+5$) a publication

(Level $n+4$) a newspaper

(Level $n+3$) The San Francisco Chronicle

(Level $n+2$) the May 18 edition of the The San Francisco Chronicle

(Level $n+1$) my copy of the May 18 edition of the The San Francisco Chronicle

[17] Language-Integrated Query

[18] object-relational mapping

[19] Sometimes "instance of" is used instead of "conforms to", when speaking about a model and its metamodel.

(Level n) my copy of the May 18 edition of the The San Francisco Chronicle as it was when I first picked it up (as contrasted with my copy as it was a few days later: in my fireplace, burning)

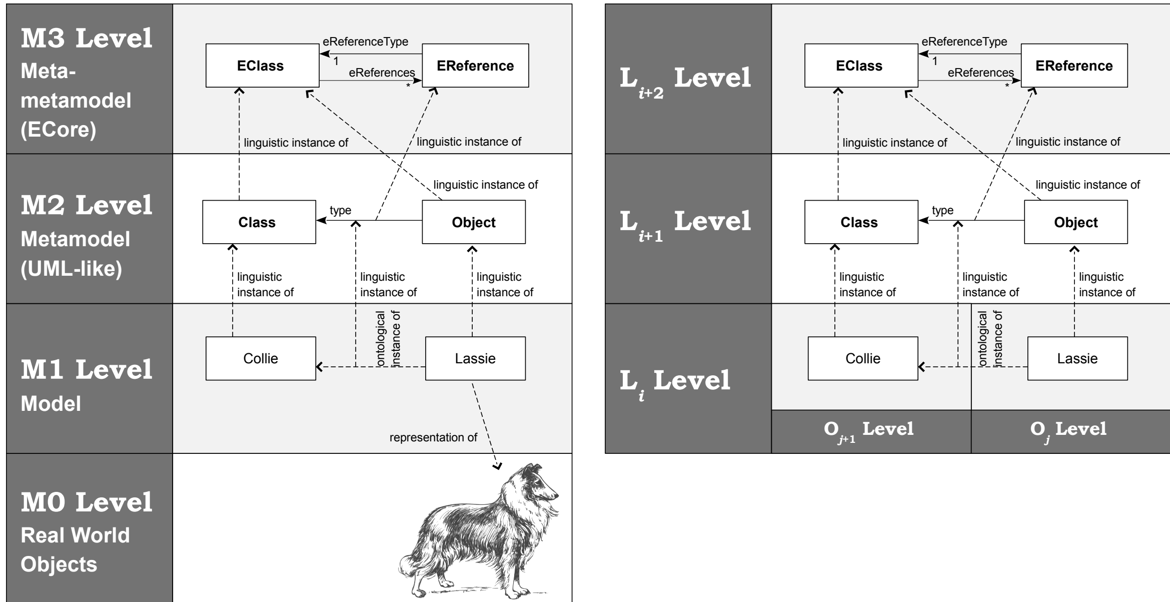
In this example, an element at Level $n+i$ is instance of the element at Level $n+i+1$, $i \in [0,1,2,3,4]$. The question arises how such multiple meta-levels can be put within the three-level conjecture?

Atkinson and Kühne [81, 82, 83] noticed that actually there are two variants of the "instance of" relation: "linguistic instance of" and "ontological instance of". Thus, meta-levels can also be linguistic and ontological. To see the difference between these two types of meta-levels, refer to Figure 1.1. Figure 1.1(a) is an example of UML Four-level Metamodel Hierarchy [42]. Although Level M0 (a level of real world objects) lies outside MOF TS, it is depicted for informative purposes. ECore is a possible meta-metamodel for Level M3 (only a small part of it is depicted in Figure 1.1(a)). Level M2 contains a UML-like metamodel described in ECore. For this example it is essential that the M2 Level metamodel is able to describe both classes and objects, see classes *Class* and *Object*. Level M1 contains a sample model, which conforms to the UML-like metamodel from Level M2.

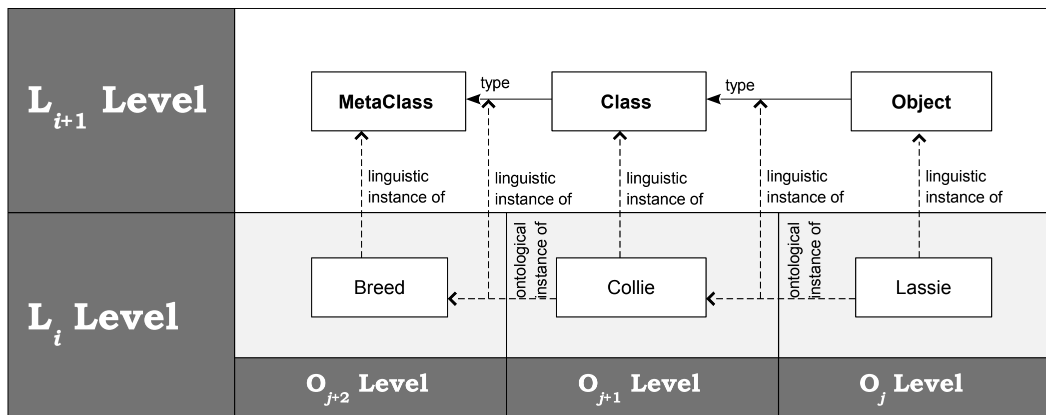
All the three levels (M1-M3) are **linguistic meta-levels**, since the meta-metamodel from Level M3 can be considered a language for specifying metamodels at Level M2, and each metamodel from M2 can be considered a language for specifying models at M1. However, if we look at Level M1 more narrowly, we can see that it can be split into two levels, where UML-style objects occupy one level, and UML-style classes occupy another level (Figure 1.1(b)). The borderline is defined by the "type" relation from Level M2. These new levels are called **ontological meta-levels**. They are denoted O_j and O_{j+1} in Figure 1.1(b), while linguistic meta-levels (M1-M3) have been renamed to L_i, L_{i+1}, and L_{i+2}. The pivot indices i and j have been introduced here to be independent on any particular absolute numbering[20]. For the purposes of this thesis, only relative arrangement of (both linguistic and ontological) meta-levels will matter.

Figure 1.1(c) adds one more ontological meta-level O_{j+1} by introducing the *MetaClass* concept at L_{i+1}. However, in order to express the six-meta-level example by D. Hofstadter, we must add at least three more meta-classes at L_{i+1} (in order the total number of ontological meta-levels become six). Figure 1.1(d) shows that instead of introducing meta-

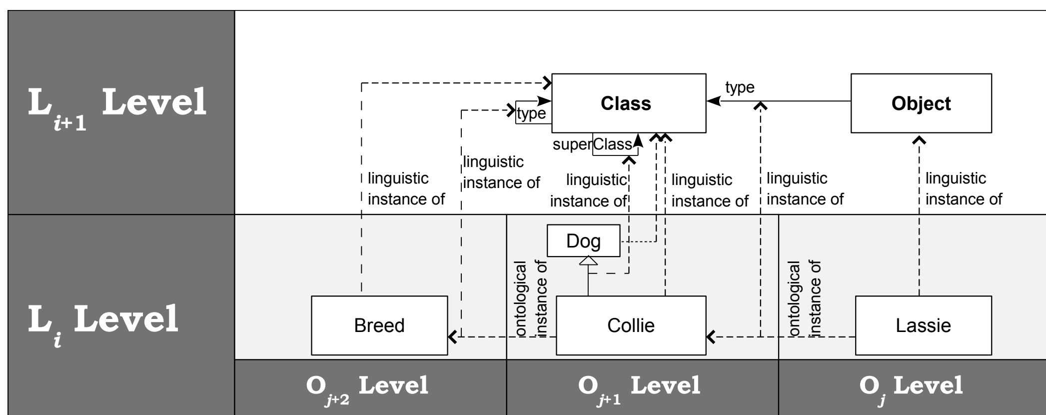
[20] Different authors may use different absolute numbering. For instance, Gašević et al. use L_1 and L_2 for L_i and L_{i+1} [84], while Atkinson and Kühne use L_0 and L_1 [81].



(a) (b)



(c)



(d)

Figure 1.1: Linguistic and ontological meta-levels. (a) UML Four-level Metamodel Hierarchy. (b) Ontological meta-levels O_j andO_{j+1} realized. (c) An ontological meta-level O_{j+2} added by introducing *MetaClass* at L_{i+1}. (d) An ontological meta-level O_{j+2} without introducing *MetaClass*.

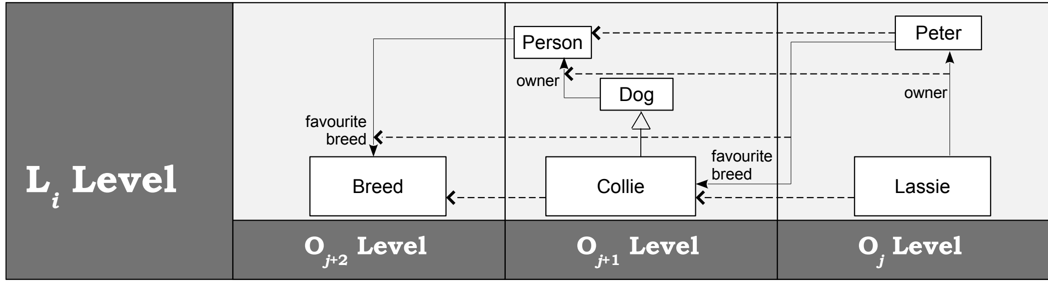


Figure 1.2: Mixing ontological meta-levels. The “favourite breed” relation between *Person* and *Breed* as well as the link between *Peter* and *Collie* cross two adjacent meta-levels.

class(es), multiple ontological levels can be described by means of the “type” relation from *Class* to *Class* at L_{i+1}. This relation can be used as many times as needed, thus, creating a chain of ontological meta-levels as in the example by D. Hofstadter (the “lowest” meta-level can be linked to the chain by the “type” relation between *Object* and *Class*).

In Figure 1.1(d), *Class* can also be made a subclass of *Object* at L_{i+1} (Atkinson and Kühne [85] introduce the term “clabjects” [from class+object] for such classes). In this case, the “type” relation between *Object* and *Class* is not needed any more, since its role is played by the “type” relation from *Class* to *Class*.

In some cases ontological meta-levels can be mixed. In Figure 1.2, a class *Person* has been added. A person can be linked to a dog by means of the “owner” relation defined at Level O_{j+1}. A person can also be linked to a breed by means of the “favourite breed” relation, which crosses levels O_{j+1} and O_{j+2}. Having a particular person *Peter* at O_j, we can see that the “owner” link lies in the same O_j level, while the “favourite breed” link crosses two levels O_j and O_{j+1}.

The theory behind the “instance of” relation goes even further. Gašević, Kaviani, and Hatala show that the (ontological) “instance of” relation can be inferred from the two other relations. They suggest to treat each class as consisting of two parts: intentional and extensional [84]. Intentional part specifies common features of objects (e.g., “has long hair”) and is used for building abstractions; the relation between elements and the intentional part is called “conformant to”[21]. Extensional part represents a class as a set, to which certain elements (objects) belong; the relation between elements and the extensional part is called “element of”. For the purposes of this thesis I do not need the “conformant to” and “element of” relations. Still, I will need both variants (linguistic and ontological) of the “instance of” relation.

[21] This is not the same as “conforms to” defined above.

On the one hand, multiple ontological meta-levels and (usually) the three linguistic meta-levels give power to describe and organize models. On the other hand, it appears that it is difficult for a human to concentrate at more than two meta-levels at a time. While working on a generator for higher order model transformations[22], where multiple meta-levels are involved, A. Šostaks came to the following conclusion (I call it **Šostaks' conjecture**):

> *It is difficult for a human to think at more than two meta-levels at a time. Still, it is fairly easy for a human to focus on any two adjacent meta-levels*[23].

One of the arguments in favour of this conjecture is as follows. Students learning Java can work with classes and objects easily (two meta-levels are used). Java is an object-oriented language, and it has the *Object* class, which is a common superclass for all other classes — an easy-to-understand OOP construction. At the same time, Java has the reflection mechanism, which permits considering Java classes as objects. The class named *Class* has been introduced for that. Actually, it is a meta-class (i.e., it brings an additional meta-level to Java), but represented as an ordinary Java class. At this point students encounter difficulties, since three meta-levels are involved[24].

Certain RDF/OWL meta-levels can be considered either linguistic, or ontological. In Table 1.1, RDF can be considered as spanning three linguistic meta-levels: the RDFS level (M3), the RDF vocabulary level (M2), and the RDF individuals level (M1). Still, RDFS permits breaking this level hierarchy, since classes may be individuals at the same time. RDFS also allows meta-levels to be mixed (as in Figure 1.2). The same refers to OWL Full and OWL 2 with RDFS semantics. Thus, we can think of RDF/OWL TS as occupying only two linguistic meta-levels: L_2 containing RDFS/OWL Full definition, and L_1 containing the pool of ontological classes and instances (L_1 conforms to L_2). In this case, L_1 consists of two or more ontological meta-levels, which can be mixed. To deal with different interpretations of meta-levels (whether the given level is linguistic or ontological), I introduce quasi-meta-levels in Section 5.1 of this thesis.

[22] this term is explained below

[23] This conjecture has been mentioned by Agris Šostaks during a personal conversation. The conjecture has been published in one of my papers [13].

[24] Personal conversations with Edgars Celms.

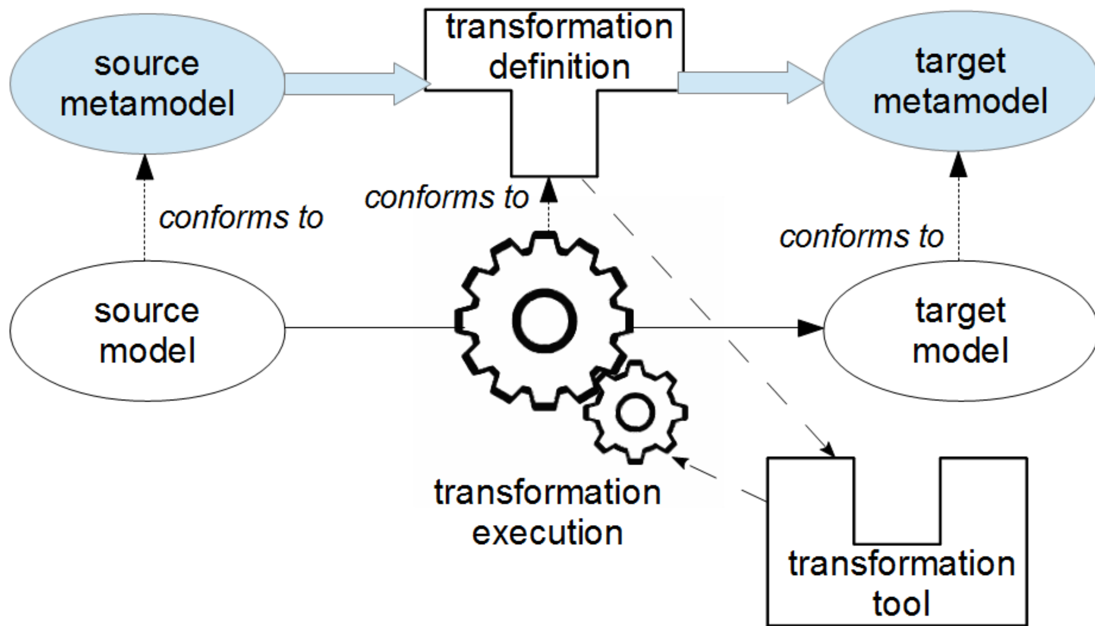


Figure 1.3: A model-to-model transformation: its definition and execution.

1.2 Model Transformations

The concept of model transformation is closely related to the concept of model. A transformation takes some source artefact and transforms it to some target artefact. In a **model transformation**, at least one of these artefacts is a model.

Different Types of Model Transformations

A transformation that transforms a model to a model is called a **model-to-model transformation** (Figure 1.3). Usually, the distinction is being made between the **transformation definition** (a set of rules that describe how to transform a source model into a target model [29]) and the **transformation execution** (a process of applying the rules at runtime). While transformation definition lies at Level M2 and is described in terms of the source and target metamodels, transformation execution is performed on models (conforming to those metamodels) at Level M1[25].

Traditionally, it is considered that some transformation tool is needed to execute a transformation definition [29]. However, it depends on a language, in which the given transformation definition is written. For some languages, transformation definitions can be deployed as precompiled binaries. In this case an additional supporting tool is not required. Here we can see an analogy between a program and its execution, where the program can be executed either by its own, or by a supporting virtual machine or an

[25]M1 and M2 can be either two adjacent linguistic, or ontological meta-levels.

interpreter. For short, I will use the term **transformation** to refer to a transformation definition. In this sense, a transformation is like a program, while the source and target metamodels are like source and target data formats.

If the result of a transformation has to be put into the source model (that is, the target model coincides with the source model), then such transformation is an **in-place transformation**. In-place transformations can be applied, for example, to perform incremental changes (updates) on models.

Generally speaking, transformations can take multiple models as inputs and produce multiple models as outputs. Moreover, these models can lie in different technical spaces.

Besides model-to-model transformations, there are also "**something-to-model**" and "**model-to-something**" transformations. Here, "something" can be[26]:

- Text or code (useful for code parsing and code generation). In this case the corresponding metamodel is replaced by a grammar or a text template.
- XML. In this case, XML schema plays the role of a metamodel.
- Database rows. A metamodel is substituted by a database schema.
- Some binary representation of a model. The data format specification replaces the metamodel.

Noteworthily, repositories from Table 1.1 perform such "something-to-model" and "model-to-something" transformations, when loading and saving models. These transformations are tailored for storing models and are not applicable for a general case.

In the context of TDA, we can talk also about "something-to-model" and "model-to-something" transformations, where "something" is a graphical presentation or a service.

This subsection presented only basic classification of model transformations. For a more detailed taxonomy of model transformations, refer to works by Czarnecki, Helsen, Mens, and Van Gorp [87, 88, 89].

Transformation Languages

When OMG introduced MDA in 2001, there were no specific model transformation languages in MOF TS [28, 29]. That is why in 2002 OMG issued a request-for-proposal called MOF QVT (Query/View/Transformation) to seek for MOF-compatible transformation languages [90]. The first version of the MOF QVT standard appeared only in 2008

[26] See also the "About model transformations" discussion at Models Everywhere [86].

[91]; the current version is 1.1 (appeared in 2011) [92]. Having the standard, tools that support it are also needed. Currently, MOF QVT is supported by such tools as *Eclipse M2M* [93] (implementing the QVT Operational and QVT Declarative[27] languages), *SmartQVT* [94] (QVT Operational), and *medini QVT* [95] (QVT Relational).

When the MOF QVT standard was in the development stage, many independent transformation languages appeared, and they continue to appear even now. There is a wide variety of transformation languages: textual and graphical, operational and declarative, unidirectional and bidirectional. As a rule of thumb, supporting tools for the given language are developed by the institution that developed the language. Some examples are: ATL [96, 97, 98], the Epsilon family consisting of several languages for different needs [99, 100], MOLA [101, 102], the Lx family [103, 104], lQuery [105], IBM Model Transformation Framework [106], GReAT [107, 108], GreTL [109], Tefkat [110, 111], VIATRA2 [112, 113], PTL [114], BOTL[115, 116], RubyTL [117, 118], etc. Some transformation languages such as COPE [119] and Epsilon Flock [120] are tailored for the model migration task.

It can be noted that ATL, Epsilon and MOLA have certain abstraction layers that allow them to be independent on the underlying model repository. Still, these abstraction layers are different, since they come from different projects.

It is interesting that even before the MOF QVT request-for-proposal, model transformation languages already existed in non-MOF technical spaces. In 1990-ties and in the very beginning of the XX century, the Typed Attributed Graph technical space already had graph transformation languages such as AGG [121, 122], PROGRES [123], TGG/-FUJABA [124, 125, 126], and VIATRA [127, 113][28]. In 1999, XML TS got the standard for the XSLT 1.0 language (Extensible Stylesheet Language Transformations) intended to transform XML documents (the current version is XSLT 3.0, a working draft) [130, 131]. Even stored procedures in the Relational Database technical space can be used to describe transformations on data from database tables. In this thesis by “transformation languages” I mean all such transformation languages from all technical spaces (not just from MOF).

Mapping languages can be considered as an abstraction of transformation languages. They can be viewed as specialized (“domain-specific”) languages for specifying model transformations. Instead of directly specifying the instructions how to transform

[27] consisting of QVT Core and QVT Relational

[28] The GrGen.NET project with its own graph rewrite language (as well as other languages) appeared in 2003, approximately two years after MOF QVT [128, 129].

a source model to a target model, a mapping specifies relations between these models, usually, at a higher level of abstraction than transformation definitions do. For instance, with MALA4MDSD, a graphical mapping language developed at IMCS-UL[29] with major contribution by E. Kalniņa [26], a mapping between two UML models can be specified easily. Languages such as D2RQ [132, 133], RDB2OWL [134], R2RML [134], and D2R MAP [135] are mapping languages between two TS's — RDB TS and RDF/OWL TS.

Mappings can be interpreted directly or translated into a lower-level transformation language for execution as ordinary transformations. The latter use-case is related to **high-order transformations** (HOT's), which are model transformations that perform certain actions on other model transformations. For instance, a HOT can be used for transformation synthesis from mappings (see the paper by M. Tisi et al. for applications of HOT's [136]). HOT's treat model transformations they work with as models conforming to some metamodel representing some transformation language. To provide better support for HOT's, existing transformation languages can be extended, or new languages can be invented. For example, in Template MOLA, a language for defining HOT's, the constructs for what to generate and the constructs for how to generate are expressed graphically in a MOLA-like syntax. There is no need to think about MOLA metamodel and its numerous technical details in order to specify what to generate (that would be necessary, if the ordinary MOLA was used).

In this thesis I assume that the term "transformation language" encompasses also mapping languages and languages for defining HOT's.

Other Languages

In many cases a language that describes a path from one model element to other element(s) is useful. Such languages are called **model navigation languages**. A well-known navigation language in XML TS is XPath [137]. Microsoft DSL Tools use a similar language [58].

A navigation language can be a part of another language. For instance, the OCL language (Object Constraint Language) from MOF TS has constructs to express navigation between model elements [138]. OCL itself is a **constraint language**, i.e., a language for expressing constraints on model elements. OCL constraints are being used in Atlas Trans-

[29]The Institute of Mathematics and Computer Science, University of Latvia

formation Language (ATL) [97]. Epsilon Validation Language (EVL) uses constraints that are "quite similar to OCL constraints" [139].

Semantic Reasoners as Model Transformations

Semantic Web is a broadening research area, which is parallel to MDE, but tightly related to it. OWL ontologies and RDF graphs can be considered as models (see RDF/OWL TS in Table 1.1). Besides, OWL has semantic reasoners — programs that can infer certain data from ontologies. These reasoners can be considered as specialized model transformations that are executed on models from RDF/OWL TS.

Classical types of tasks performed by reasoners are [140]:

- Consistency checking: whether the given ontology is not contradictory.
- Concept satisfiability: whether the given class can have at least one instance.
- Classification: computes the complete class hierarchy.
- Realization: find the most specific class for the given instance.

Reasoners are based on impressible Math such as description logic. There are various OWL variations, and there is a trade-off between what can be inferred and how fast that can be done (if that can be done at all). For instance, a reasoner for OWL Lite terminates in polynomial time, a reasoner for OWL DL terminates in exponential time, reasoners for certain OWL 2 variations terminate in double-exponential time, while OWL full is undecidable [64]. Examples of commercial reasoners are OntoBroker[30], and RacerPro[31]. Examples of free reasoners are Pellet[32] (with an option to obtain a commercial license), Fact++[33], and Hoolet[34].

Semantic reasoners infer data according to the **open world assumption** (in contrast to the **closed world assumption** used, for example, in Prolog). Stefano Mazzocchi explains the difference between both assumption in one sentence [141]:

> "[The closed] world assumption implies that everything we don't know is *false*, while the open world assumption states that everything we don't know is *undefined*."

[30]`http://www.semafora-systems.com/en/products/ontobroker/`
[31]`http://www.racer-systems.com/`
[32]`http://clarkparsia.com/pellet`
[33]`http://owl.man.ac.uk/factplusplus/`
[34]`http://owl.man.ac.uk/hoolet/`

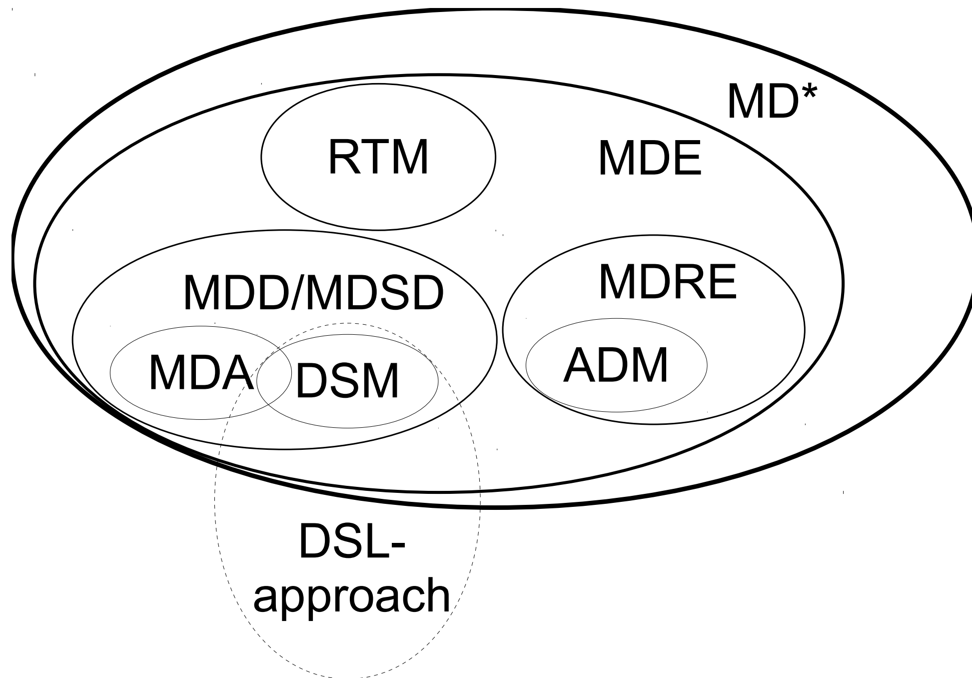


Figure 1.4: Research areas in the "model-driven" world, and their relationships.

The closed world is based on the "negation as failure" principle, while the open world is based on the "negation as contradiction" principle [142]. An OWL ontology can be processed by a semantic reasoner according to the closed world assumption by adding some additional OWL statements that "close" the world of the ontology.

In this thesis, semantic reasoners will be considered as in-place transformations that append existing models with inferred data. The main difference from ordinary model transformations is that a reasoner implements a set of predefined model transformations (like from the list above). Reasoners are usually written in third generation programming languages (3GLs) and are deployed in a binary form. The reasoning process can be ruled only in a declarative way by means of data stored in an ontology. Thus, no specific transformation language is needed in this case.

1.3 The World of "Model-Driven"

This section presents the relations between various research areas in the "model-driven" world. The survey is based on efforts of several people, including Jordi Cabot, David Ameller, Jean Bézivin, Markus Völter, and Elīna Kalniņa, who tried to clarify terms such as MDD, MDE, etc. [143, 144, 145, 146, 147, 26]. Refer to Figure 1.4, while reading.

Model-Driven Architecture (MDA) and Architecture-Driven Modernization (ADM)

In 2001 OMG launched the *Model-Driven Architecture, MDA,* — an approach for software development that encourages the usage of models and their transformations [27, 28, 29, 148]. The word "architecture" in MDA does not refer to a software architecture (i.e., the architecture of the system being built), but rather to the software development process according to OMG standards (including UML and MOF [42, 46]). According to MDA, a model that describes a system at a high level of abstraction is created first. Then, by means of a sequence of model transformations, this model is specialized, and finally an executable code is obtained. MDA is explained more in detail in Chapter 2.

Architecture-Driven Modernization, ADM (the reverse of MDA), is a process of modernization/reverse engineering of existing non-built-with-MDA software to leverage earlier investments in software development [149]. Currently, OMG's Architecture-Driven Modernization Task Force is developing standards and specifications to support ADM[35].

Model-Driven Engineering (MDE)

Model-Driven Engineering, MDE, is a broad research area, where models are first-class citizens. MDE is not tied to MDA standards. Neither MDE is tied to the software development process. In 2010, J. Bézivin tried to identify the application scope of MDE [145]. He listed three subsets of MDE:

- *Model-Driven Software Development (MDSD)*[36]. MDA is a special case of MDSD (MDA $\approx$ [MDSD according to OMG standards]).
- *Model-Driven Reverse Engineering (MDRE).* ADM is a special case of MDRE. MoDisco is an exemplary project, which can be used in practice for applying MDRE techniques to legacy systems [150, 151, 152].
- *Run-Time Modelling (RTM).* In RTM, models are used at run-time, not at the development/maintenance stage.

Markus Völter introduced the common moniker MD* to denote numerous model-driven approaches [147]. Following E. Kalniņa, I will use MD* to denote "Model-Driven

[35] See web-page `http://adm.omg.org`.

[36] MDSD is also called MDD (Model-Driven Development)

Everything", i.e. all possible (even yet unknown) applications of models at software development time or at runtime [26].

Jordi Cabot mentions also *Model-Based Engineering, MBE*. He defines MBE as "a process in which software models play an important role although they are not necessarily the key artifacts of the development (i.e. they do NOT "drive" the process as in MDD)" [143]. Thus, we can think of MBE as a superset of MD*. This thesis focuses on models that "drive" software or software development process. Thus, the thesis stays within the MD* area. Current applications of TDA, the topic of this thesis, are linked to the three MDE subsets listed above. Thus, I consider that TDA belongs to the MDE area.

Domain-Specific Modelling

A **domain-specific language, DSL,** is a specialized language for a particular problem domain. A DSL uses terms native to this domain [58]. A DSL needs some tool for specifying problems in this DSL and for solving them. Such tools are called **domain-specific tools** or **DSL tools**. Chapter 4 explains DSLs and DSL tools more in detail.

Domain-Specific Modelling (DSM) is an approach, when models are used in domain-specific tools [18]. Although domain-specific tools can be created without using models (at least explicitly, see "DSL-approach" in Figure 1.4), models bring certain benefits to these tools. A model can describe the abstract syntax of a DSL (the domain). Models can even describe textual and graphical presentations of DSL's. Model-to-model transformations can be used to establish the link between the domain and the presentation, while model-to-code transformations can be used to generate code from the domain.

DSM is the area, where TDA, the topic of this thesis, has been approbated.

Chapter 2

Motivation

This chapter prepares the ground for the Transformation-Driven Architecture, TDA, using the Model-Driven Architecture, MDA.

2.1 The MDA Idea

In MDA, the development process starts not with code, but with a model called *computation independent model, CIM*. This is a model at a high level of abstraction. As MDA Guide 1.0.1 states, "A CIM is sometimes called a domain model and a vocabulary that is familiar to the practitioners of the domain in question is used in its specification" [28, Section 2.2.11]. Then, CIM is transformed into a *platform independent model, PIM*, which "is a view of a system from the platform independent viewpoint. A PIM exhibits a specified degree of platform independence so as to be suitable for use with a number of different platforms of similar type" [28, Section 2.2.12]. Thus, whether new platforms appear, or existing platforms change, CIMs and PIMs can be reused.

PIM is transformed into one or more *platform specific models, PSM's*, by taking into a consideration additional information such as platform-specific aspects. If a system being built makes use of several platforms, the interoperability between these platforms can be ensured by introducing so called *bridges* during PIM-to-PSM transformation. At the final stage, PSMs are transformed into an executable system (code). There can be certain exceptions in this chain of transformations. PIM can be transformed directly to code. Or, there may be multiple PIM's/PSM's in the chain, where one PSM can be considered a PIM for a more detailed PSM. In some cases the process starts not with a CIM, but with a PIM.

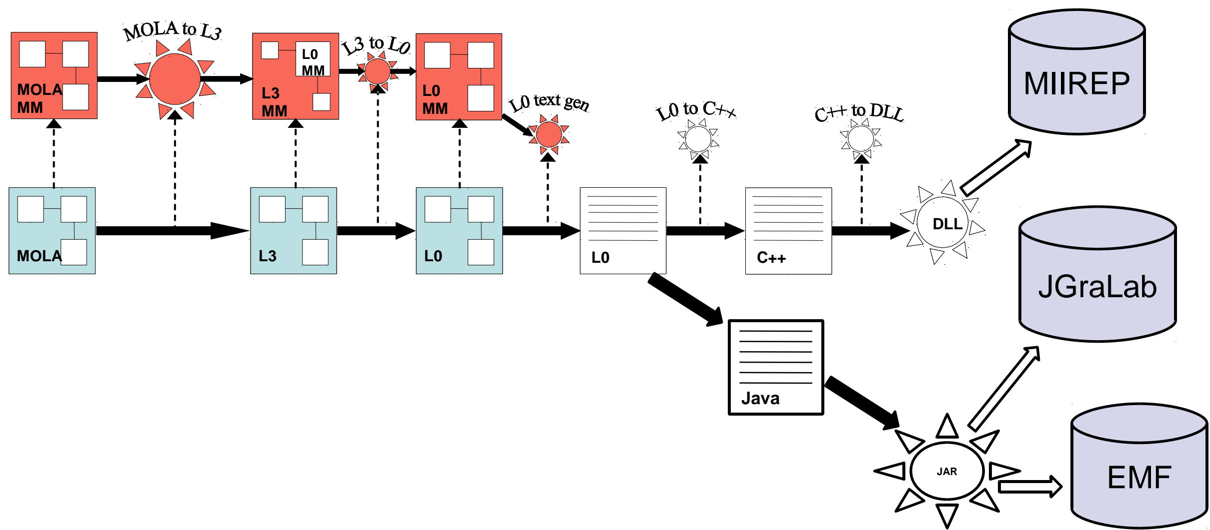


Figure 2.1: Compilation of a MOLA program. The process corresponds to MDA principles. (By kind permission of Agris Šostaks, the author of the drawing.)

Models in MDA are not just sketches on a sheet paper, but primary artefacts used to build a system. Such models do not lose their value throughout the life cycle of the system, even when the underlying platform changes. Obviously, it is more convenient to make changes in source code written in some high-level programming language than in machine code. Similarly, when using MDA, it is more convenient to make changes in models, transform them, and then to obtain (generate) code. Ideally, it should be possible to automate all MDA transformations. In reality, this is not usually the case.

An important research is being made under the direction of professor Jānis Osis at Riga Technical University. This research group presented a way of specifying the whole system, including its functional cycles, in CIM, with a possibility to automate other MDA steps [153, 154].

2.2 MDA: Pros and Cons

MDA ideas indeed are usable. The OMG homepage lists numerous success stories of using MDA [155]. The principles behind MDA are also used by the MOLA compiler developed at IMCS-UL [101, 156]. An initial PIM that describes a MOLA program is transformed into a chain of intermediate PIMs/PSMs. Finally, C++/Java code (being able to work with a particular model repository) is obtained, see Figure 2.1.

Still, pure MDA is not applicable everywhere. Scott W. Ambler, a notable MDA critic, provides a check-list to consider, when deciding on moving to MDA [157]. Nowadays, MDA is also being questioned by argumented objections such as practical usage of the

PIM-PSM chain, prescribed usage of UML, misuse of terms "platform-independent" and "architecture", and whether MDA will work [158, 159, 160, 32]. Some even consider MDA obsolete [161].

Nevertheless, there are attempts to get something useful out of MDA. Scott W. Ambler stands for "Agile MDA" [162, 163]. Microsoft, which did not support OMG standards for a certain period of time [164], currently is working with OMG on improving UML and bringing modelling into mainstream [165, 166]. Many agree that MOF, which finds its roots in UML/MDA, is a valuable standard by its own. Furthermore, MDA and the subsequent request for proposal on MOF QVT (Query/View/Transformation) became an impetus for the emergence of model transformation languages [90].

2.3 Why to Advance MDA Ideas?

Notwithstanding all the pros and cons, MDA is a noticeable step in incorporating models into the software development process. I would like to mention just some points in favour of using models.

- A model provides an overall view to objects, where each object is placed within its context. Models are associated with the "Everything is a model" principle, which extends the classical "Everything is an object" principle [30].
- Models are programming language neutral.
- Models can act as artefacts and as documentation at the same time.
- Models are flexible — they can describe data at different levels of abstraction: from low-level processor commands to high-level business processes. Transformations between models can be used to establish the link between different abstraction levels.
- Models provide a convenient way for describing domain data in domain-specific languages.

Modern software consists of multiple interrelated components, which need to communicate. I call such systems **interactive**. Although models are a convenient means to describe individual components, it is difficult to describe the dynamics of an interactive system (including the communication between its components) in a model. If we describe the

dynamics in a model, this description, in essence, would still correspond to some program. The only difference would be that this program would be encoded in a model (as contrasted to code in some programming language). This would complicate the model and nullify the benefits offered by the model-driven approach.

According to MDA, the whole system (including its dynamics) has to be described in a model. Then, transformations are used (at development time) to obtain the code. Thus, the classical MDA is not suitable for developing interactive systems. Consider the following modifications to MDA, tailored to interactive systems. Let models describe the components of an interactive system. Let transformations (not models!) describe the dynamics. This brings transformations to runtime and leads to a software architecture called **the Transformation-Driven Architecture, TDA**.

There are numerous acknowledged technologies in the model-driven world, e.g., model repositories, transformation languages, etc. TDA makes use of them. There are also numerous technologies in the traditional code world, such as dynamic link libraries (DLLs), the .NET platform, the Java platform, functional languages, etc. It is more convenient to implement certain functionality (e.g., platform-specific functionality) directly in these technologies, without the need to develop models for each such technology (Dave Thomas provides interesting arguments that this would require Herculean efforts, and even then such models would be practically unusable, since it would be difficult to maintain them [158]). TDA does not insist on moving all to the model-driven world. Traditional technologies can be used within TDA as well. Thus, TDA can be used to combine the best of the model-driven world and the traditional-code world. Having certain components developed in traditional-code world, the business logic still can be implemented in a platform-independent way by means of model transformations.

The next chapter (Chapter 3) presents the essence of TDA (details are provided in Part II).

An important application of TDA is development of domain-specific tools. The productivity increase being brought by domain-specific tools can be compared to the productivity increase brought by third generation programming languages (3GLs) [18]. Since domain-specific tools can be classified as interactive systems, TDA is a suitable approach to develop them. Chapter 4 explains this more in detail. As we will see, TDA has certain advantages over other approaches for creating DSL tools.

Chapter 3

The Transformation-Driven Architecture

This chapter introduces the Transformation-Driven Architecture, a model-driven approach for building interactive systems. In this chapter TDA elements and features are identified, and certain design-choices are explained.

3.1 The Outline View on TDA

The outline view on TDA is depicted in Figure 3.1. The following subsections explain elements from that figure.

Engines

When using the TDA approach, the auxiliary functionality that is required to implement the main functionality (business logic) has to be identified first. Auxiliary functionality refers to:

- graphical or textual presentations of data (such as graph-like diagrams, dialog windows, tree-like explorers, text/code editors, music score editors, etc.);
- common services (such as generation of documents, sending e-mails, printing services, cloud services, multimedia services, etc.).

Modules implementing the auxiliary functionality are called **engines** in TDA. Engines are usually platform-dependent. They can rely on the operating system functionality, or on a particular platform such as Java or .NET. An engine may depend on some library

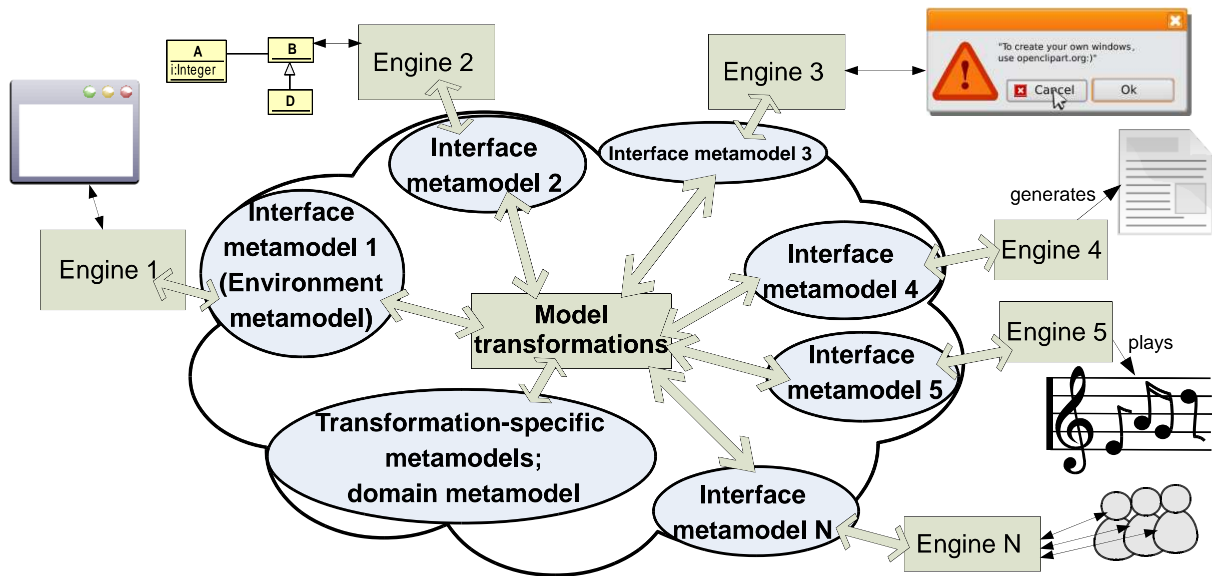


Figure 3.1: The outline view on the Transformation-Driven Architecture, TDA.

written in a particular programming language, thus, the library may force the engine to be written in the same language. Certain engines may be easier to implement in a functional or declarative programming language. TDA does not put restrictions on how engines are implemented. The only requirement is that engines have to be associated with their interface metamodels.

Interface Metamodels

The main functionality (business logic) of a system does not need to be aware of how engines are implemented internally. Thus, for each engine only the essence (from the business logic point of view) has to be identified. In TDA these essences are described by means of **interface metamodels**. Each interface metamodel contains:

- a description of data related to the functionality of the corresponding engine;
- a description of points of communication between the engine and the business logic (the communication mechanism is explained in Section 3.3).

Each engine must be able to work with models conforming to its interface metamodel (**interface models**; not depicted in Figure 3.1).

Since interface models contain only essential (and, thus, usually platform-independent) data, they can be compared to PIM's in MDA. Engines can be viewed as runtime transformations between these PIM's and actual (usually platform-specific) presentations and services. Notice that unlike MDA, TDA has multiple "PIM's".

Engines, their presentations/services, and interface metamodels can also be compared to the three-tier architecture [167], which makes a distinction between data, their processing, and their presentation. In this analogy, interface metamodels stand for the data tier, presentations/services offered by engines stand for the presentation tier, and engines stand for the logic tier, which is a mediator between the data tier and the presentation tier. The three-tier pattern can be observed for each presentation engine used within TDA.

Interface metamodels in TDA can serve as bridges between the traditional code world (where engines are usually implemented) and the model-driven world (where model transformations reside).

Model Transformations

In TDA, model transformations (in the middle of Figure 3.1) implement the main functionality (business logic) of a system. These are model-to-model transformations. However, a transformation in TDA can work with multiple models, some of them being source models, some being target models, and some being source and target models at the same time (i.e., when these models are being modified). This design choice permits using different kinds of transformations in TDA, for instance:

- If a transformations reads from one model and creates another model, then it is a classical model-to-model transformation from a source model to a target model.
- If a transformation performs incremental changes in a single model, then it is an in-place transformation.

In TDA, transformations are executed at runtime.

Transformations can be defined using

- traditional programming languages;
- specialized transformation and mapping languages (mentioned in Section 1.2);
- OWL constructions. A semantic reasoner uses these constructions to update the model with inferred data. Thus, OWL constructions correspond to a transformation definition, while the process of launching a reasoner corresponds to transformation execution, see Section 1.2.

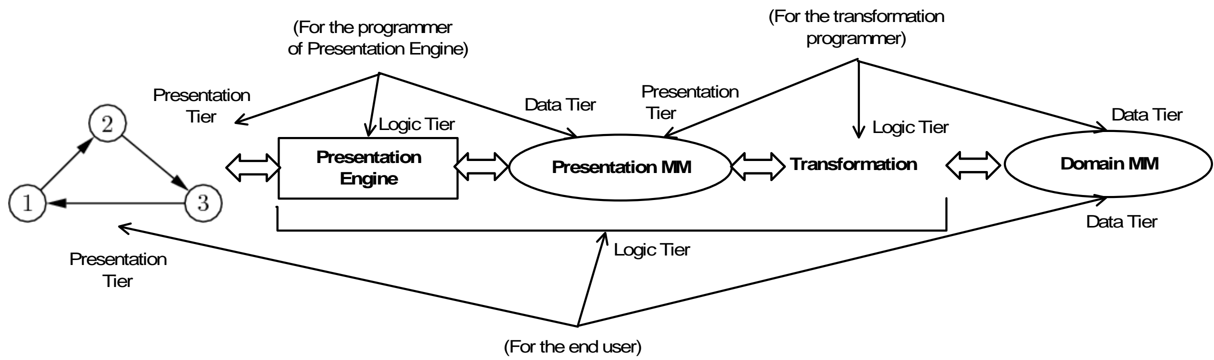


Figure 3.2: Matching the three-tier pattern in TDA (triple-lines show relations to the three-tier architecture). Among several possible engines, only one engine ("Presentation Engine") and its interface metamodel ("Presentation MM") are shown in this example. The domain metamodel ("Domain MM") is an additional metamodel introduced by model transformations.

Other Metamodels

In addition to interface metamodels, TDA permits defining other metamodels as well. Transformations can define metamodels that are useful for implementing the business logic. For instance, if the interface metamodel of some engine contains too many details, a simplified interface metamodel can be introduced. This simplified metamodel could act as a "view" to the full interface metamodel, and a special helper transformation could establish the link between them.

For domain-specific tools, a domain metamodel can be introduced. This metamodel could act as a pivot metamodel for describing data by means of domain concepts. If we recall the three-tier architecture mentioned above, then adding a domain metamodel to TDA allows us to establish the three-tier pattern also w.r.t. this domain metamodel (see Figure 3.2).

3.2 The Technical View on TDA

Figure 3.3 depicts the technical view on TDA. This section only identifies new TDA elements presented in this view. Technical details of these elements are provided in Chapter 5.

TDA models and metamodels have to be stored in some repository (see Table 1.1 from Section 1.1). To be independent on a particular repository, TDA introduces a common repository abstraction layer called **Repository Access API[1], RAAPI**. Since each repository uses its own API, some adapter is required to translate RAAPI calls to

[1]Application Programming Interface

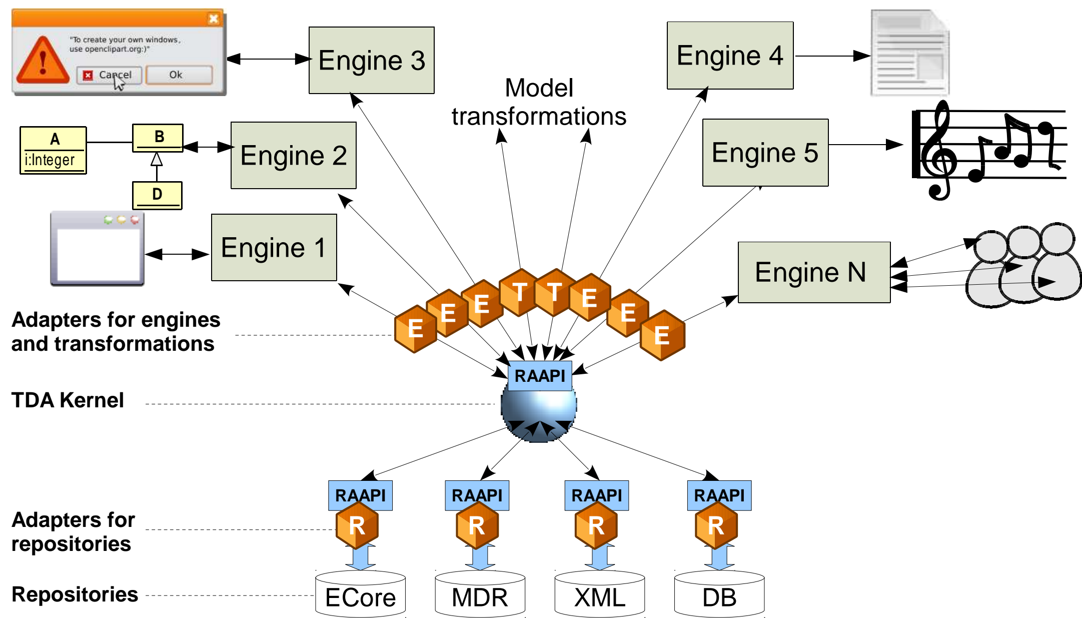


Figure 3.3: The technical view on TDA.

calls of that particular repository. Such **adapters for repositories** are denoted by the letter “R” in Figure 3.3.

Engines and transformations use RAAPI to access models and metamodels. RAAPI calls are passed to an adapter of a particular repository through a special component called **TDA Kernel**. This design choice allows TDA Kernel to intercept RAAPI calls, which is very useful for implementing the communication mechanism, the undo/redo mechanism, the multi-repository mechanism and other features. TDA Kernel has its own (technical) interface metamodel called **Kernel Metamodel** (not depicted in Figures 3.1 and 3.3).

Engines and transformations can be written using different languages and technologies. To be able to call engines and transformations in a uniform and platform-independent way, **adapters for engines** and **adapters for transformations** are introduced in TDA (denoted by the letters “E” and “T” in Figure 3.3, respectively). These adapters can introduce **RAAPI wrappers**. For instance, if an engine or a transformation needs to use ECore API instead of RAAPI, a wrapper that implements ECore API and forwards its calls to RAAPI can be developed.

Having all TDA elements identified, the communication mechanism of TDA can now be explained.

3.3 The Communication Mechanism in TDA

TDA needs a communication mechanism. Transformations implementing the business logic need to access presentations and services implemented in engines. Besides, one engine may need to use the functionality provided by another engine (this is the case with Environment Engine, to which other engines attach their windows; see Chapter 8). Also, one transformation may need to call another transformation.

Each communication step involves two parts, each part being either a transformation or an engine. First, I discuss the communication, when one of the parts involved is an engine. Then, I explain the communication between two transformations. Finally, I provide an example of several communication steps, where two engines and three transformations are involved.

Communication Between Two Engines, or Between an Engine and a Transformation

The communication with an engine is bi-directional:

- an engine may be commanded to perform some actions (e.g., to refresh the presentation to reflect the changes in the corresponding interface model, or to execute some service the engine provides);
- an engine may produce certain events (e.g., events about certain changes in the presentation, or user interface events), in which other parts (engines or transformations) may be interested.

I call these two concepts in communication, respectively, **commands** and **events**. One of the possible solutions to implement events and commands is by means of MOF-like operations. However, this solution has the following shortcomings:

- The underlying repository has to support operations. While ECore supports operations, RDF and OWL repositories (and certain model repositories such as JR [55]) do not, and additional efforts are required to store operations there.
- We must be able to call operations somehow. This is a non-model-based element in the communication, since calls have to be implemented in native code. Although

certain model transformation languages support calling operations, it may be a non-trivial job (if accomplishable) to redirect these calls to engines. Since engines, in their turn, may produce events, a way to specify callback operations is also needed.

A simpler solution, which does not use operations is as follows: commands and events are classes (subclasses of the special classes *Command* and *Event*, defined in Kernel Metamodel). Parameters are described by means of attributes, or associations to other classes (these classes define the **context** of an operation).

> *Note.* This design choice is not my personal contribution. It was proposed by other colleagues, when working on the GrTP and Metaclipse platforms at IMCS-UL in 2006-2008. Andris Zariņš implemented the initial variant of the event/command mechanism. The text below presents my additions and modifications to it.

When a particular command needs to be called, the corresponding command class instance is created. Then, parameters (attribute values and association links to the context) are specified. In order to process this command, the command has to be linked to a special singleton object called the **submitter** (defined in Kernel Metamodel). When a command object has been prepared and linked to the submitter, we can say that the command has been **issued**. At this moment TDA Kernel intercepts the action of creating a link to the submitter, and calls the corresponding engine to process the command. When the engine finishes executing the command, TDA Kernel deletes the command from the repository.

This design choice does not require introducing new constructs in transformation languages to be able to pass commands to engines. Similarly, RAAPI is sufficient for engines to be able to execute commands. Besides, it is unnecessary to specify which engine to call, since the engine can be uniquely identified from the command class (this class belongs to only one interface metamodel).

When an engine is active, it can inform transformations and other engines about certain events (e.g., the user has entered the data, some job has been finished, etc.). In this case the engine creates an instance of the corresponding event class and sets the parameters. Then, the engine links the event to the submitter. We can say that the event has been **emitted**. TDA Kernel intercepts this and calls engines and transformations registered as listeners to this event.

To register event listeners the following design choice is used. For each event type of the given engine the corresponding ***on*-attribute** has to be defined somewhere in the

interface metamodel of that engine, preferably, in the context of the event. The name of an *on*-attribute is a concatenation of the prefix "on" and the corresponding event class name, e.g., *onClickEvent* for the *ClickEvent* class. The value is a string that encodes which engines and transformations have to be called, when this event occurs (the encoding is explained in Chapter 5).

Communication Between Two Transformations

When a transformation needs to call another transformation (perhaps, written in another transformation language), it needs to issue a *LaunchTransformationCommand*. This command is defined in Kernel Metamodel and has only one attribute, *uri*, which encodes the name of the transformation to call (the encoding is similar to the encoding of on-attributes; it will be explained in Section 5.3 of Chapter 5).

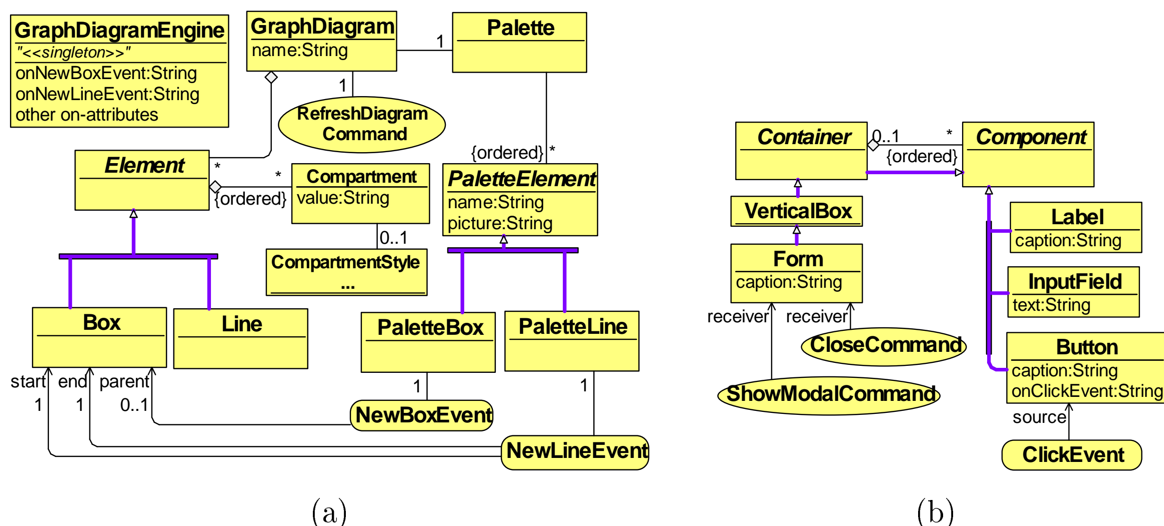
It is up to transformations how to specify their parameters. One option is to define parameters in the domain metamodel. Another option is to introduce a subclass of *LaunchTransformationCommand*, and define parameters in that subclass.

Since TDA does not put restrictions on internal implementation details of engines, one can consider an option to create an engine implemented by a set of transformations. In this case the communication is the same as described in the previous subsection.

An Example of Communication

Assume we are developing a tool that is able to edit some kind of activity diagrams, e.g., business process diagrams. The auxiliary functionality would include Graph Diagram Engine [2, 5] for editing graph-like diagrams and Dialog Engine [6, 12] for entering certain data, e.g., names of actions. Figure 3.4 depicts fragments of interface metamodels of these two engines.

Assume there already exists a graph diagram represented in a model as a *GraphDiagram* instance. Assume that this diagram already has a *Palette* with some *PaletteElement*-s, where one of the elements (an element of type *PaletteBox*) is responsible for adding a new action to the diagram. When a user adds a new action to a diagram, Graph Diagram Engine creates a *NewBoxEvent* and attaches it to the *PaletteBox* element used to create the box (action). If a new box is created within some other box, a link to that parent box is also created. Then the event instance is attached to the submitter. At this moment, TDA Kernel calls the event handler specified in the *onNewLineEvent* attribute



(a) (b)

Figure 3.4: Fragments of (a) Graph Diagram Engine Metamodel and (b) Dialog Engine Metamodel.

of the *GraphDiagramEngine* class. Let this event handler be a transformation that creates a dialog window for entering the action name.

To create a dialog window, the event handling transformation creates a dialog *Form*, to which components such as a *Label* (with *caption* = "Please, enter the action name:"), an *InputField*, and two *Button*-s (with captions "OK" and "Cancel") are attached. For each of the two buttons the *onClickEvent* value is set to denote the transformation for processing the click. To show the form, a *ShowModalCommand* is issued by attaching it to the submitter. TDA Kernel calls Dialog Engine, which displays the form and waits for the user input.

When the user clicks the "OK" button, the corresponding "OK"-transformation is called. It creates a *Box* instance followed by a *Compartment* instance that is attached to this box. Then, the transformation sets the *value* for the compartment (according to the *text* entered in the *InputField*), creates and initializes a *CompartmentStyle* object (this object stores the text colour, the font size, etc.), and attaches this style to the compartment. After that, the transformation issues a *RefreshDiagramCommand* to repaint the diagram with the newly created box. Finally, the transformation closes the form with *CloseCommand* and deletes the form from the repository. The "OK" transformation can delegate some steps to other sub-transformations by means of *LaunchTransformationCommand*.

The "Cancel"-transformation just closes the form and deletes its model from the repository.

Chapter 4

Creating Domain-Specific Tools

In 1986, Jon Bentley noticed that programmers, while performing their work, usually invent and use "little languages" [168]. Bentley suggests to consider the problem of printing a floating point number occupying six characters. These characters include a decimal point and the two decimal digits after it. To specify what to print we could use the format F6.2 in FORTRAN or 999.99 in COBOL. Today C/C++ and Java are more widely used, and there we can use the %6.2f format, which can be handled by the printf function. Each of the formats just described is an expression in some "little language". While Bentley calls these language "little" (in contrast to the "big" ones like FORTRAN, COBOL, or Pascal), nowadays we would call them **domain-specific languages, DSLs** (in contrast to the general-purpose ones), since each such language is tied only to one particular class of problems.

4.1 Domain-Specific Languages and Tools

In their excellent book, four specialists, who worked on *Microsoft DSL Tools*, a toolset from Microsoft supporting development of DSLs, provide the following definition [58]:

> "A Domain-Specific Language is a custom language that targets a small problem domain, which it describes and validates in terms native to the domain."

DSLs may be textual or graphical. Here are some examples.

- Textual (in addition to those mentioned above):
  - regular expressions;

  - a language for querying a database (e.g., SQL or SPARQL [169]);
  - languages for describing context-free grammars (e.g., EBNF or its derivatives like the language used in Yacc[1] [39, 170]).

- Graphical:

  - a language for specifying states of mobile applications graphically (a good example is provided by S. Kelly and J.-P. Tolvanen [18]);
  - a language for specifying GUI (Graphical User Interface) forms (e.g., a form editor in the NetBeans or Microsoft Visual Studio environment);
  - flow charts describing certain business processes (e.g., BPMN [171]).

When third generation programming languages (3GLs) replaced assembler, the productivity of programmers increased drastically. The same is true for DSLs. There are two main factors making that possible: 1) a better level of abstraction, and 2) a possibility to map it automatically to the existing lower level techniques.

Certain DSLs may be designed so close to the problem domain and use the exact notions from the domain, that domain experts, who are not programmers, can understand and express problems in a DSL. A problem description in a DSL is like source code at a very high level of abstraction, where it is easier to make changes and perform validations. When a problem has been described in a DSL, this description can be used to perform the required task, for instance, to generate certain artefacts (e.g., code or documents).

A DSL needs some tool for specifying problems in this DSL and for solving them. Such tools are called **domain-specific tools** or **DSL tools**. Since DSLs are domain-specific, no universal tool exists for them. More DSLs will require more DSL tools. To minimize the cost of developing and maintaining DSL tools, DSL tool-building platforms have emerged. These platforms are, in effect, DSL tools for creating other DSL tools.

The syntax of a DSL usually consists of two parts: the **concrete syntax** (textual or graphical presentation) and the **abstract syntax** (describing the essence without unnecessary details). The concrete syntax is a notation that is more convenient for a DSL user. The abstract syntax, in its turn, is more convenient to access and automatically process the data expressed in a DSL.

[1] Yet Another Compiler Compiler

Example. The C++ language constructs define a concrete syntax for programs. Taking a program as an input, a C++ parser produces an abstract syntax tree, which is a convenient data structure for further processing of the code (e.g., performing code optimizations and generating the binaries).

Metamodels are a universal means to encode the abstract syntax. Modern DSL tool-building platforms use some kind of meta-metamodel to define metamodels for domain data (**domain metamodels**)[2]. In some platforms, presentations (concrete syntax) are also defined as metamodels.

Let us look at several tool-building platforms and their approaches for creating DSL tools. Then, I explain how TDA can be used to build DSL tools.

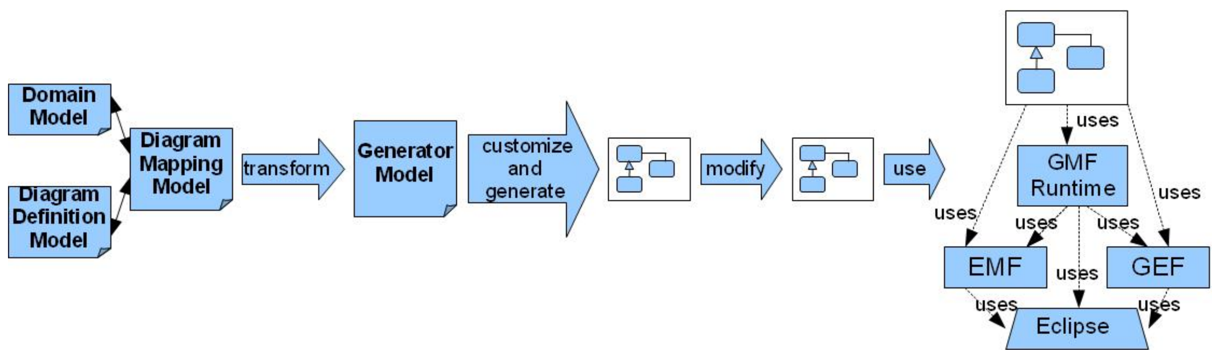
4.2 Existing Model-Driven Tool-Building Platforms

To get some insight on typical ways of developing and executing DSLs by means of these platforms, let us look first at the three big players in the area — GMF, MetaEdit+ and Microsoft DSL Tools.

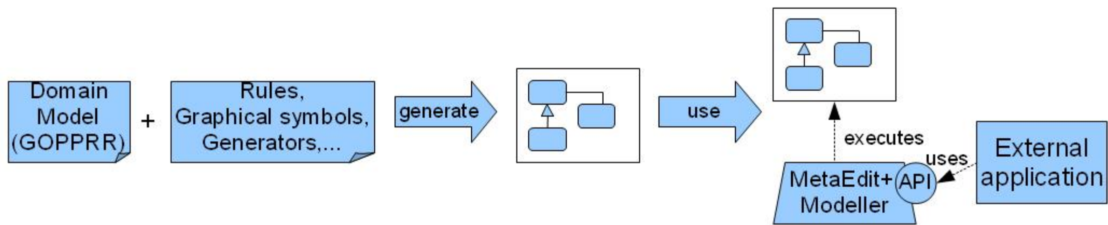
In GMF[172, 173] (Figure 4.1(a)), three models have to be specified first. They are a domain model, a diagram definition model, and a diagram mapping model. The latter model establishes mappings between the first two. Then, from these three models (which together may be viewed as a platform-independent model, PIM, according to MDA principles) are transformed to the Generator model (which may be considered a platform-specific model, PSM). The generation parameters in the Generator model may be adjusted, and then Java code is generated. Common features are not generated; they lay in the GMF Runtime library. The generated Java code can be manually edited[3] to enhance the functionality of the generated DSL tool. The tool is executed in the Eclipse environment and uses EMF (for modelling support [48]) and GEF (for editing graphical diagrams [174]) as well as GMF Runtime, which also uses EMF and GEF.

In MetaEdit+ Workbench [59] (Figure 4.1(b)) the domain model is specified by means of dialog windows in the GOPPRR[4] syntax. Then, graphical symbols (palette elements) for domain concepts are created. Reports and generators for the DSL can also be specified. Then, by pressing the "Generate" button, the tool is generated and stored in the

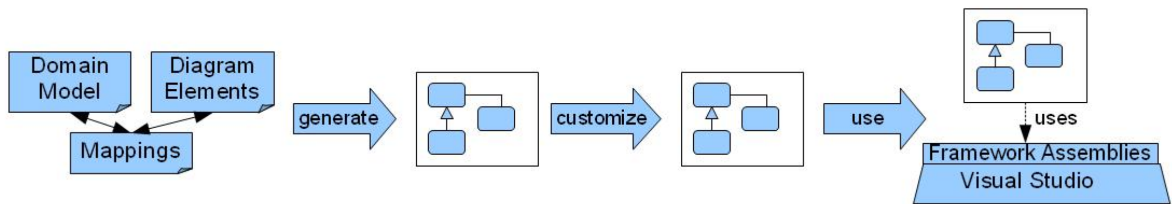
[2]The domain model corresponds to CIM in MDA.
[3]Modified regions are marked, thus, they will not be lost, if the generation process is relaunched.
[4]Graph-Object-Property-Port-Role-Relationship



(a) GMF



(b) MetaEdit+



(c) Microsoft DSL Tools

Figure 4.1: Creating DSLs in a) GMF, b) MetaEdit+, and c) Microsoft DSL Tools.

repository. The generated tool can be executed in the MetaEdit+ Workbench itself, or in the MetaEdit+ Modeller environment, which is a lite runtime version of the MetaEdit+ Workbench. Although MetaEdit+ has only finite customization capabilities, these capabilities are very convenient, easy-to-use and have a good coverage of business-specific needs. In case some additional functionality is required, it may be implemented outside MetaEdit+. In this case, the model may be accessed through MetaEdit+ API, which is a reasonable way to access data in a closed-source platform.

In Microsoft DSL Tools, MS DSL [58] (Figure 4.1(c)), the domain model and the corresponding diagram elements are specified side-by-side in one graphical view. Mapping associations between model elements and diagram elements are also specified there. MS DSL allows the DSL developer to customize some aspects of the generated tool (such as validation and XML serialization) by means of graphical interface. If the built-in customization capabilities are not sufficient, lacking features can be specified in the C# language[5]. After the generation process, compiler error messages can be used to see, where the required C# code is to be inserted. At runtime, MS DSL uses the precompiled Framework Assemblies, which include Domain Model Framework, Design Surface Framework (for handling diagrams, shapes and connectors), frameworks for model validation and code generation, and a framework for hosting the generated tool in the Microsoft Visual Studio environment during runtime.

All three platforms from my short survey provide a set of basic elements for creating typical graphical DSLs. DSL users benefit from entering information in concrete graphical syntax, while the tool automatically maintains the mapping between the concrete and abstract syntaxes. All three platforms provide facilities for code generation from the domain model (the abstract syntax) at runtime.

Now I will briefly discuss other platforms.

KOGGE [175] is a generator for CASE tools that are described declaratively at high level of abstraction. KOGGE appeared in mid-nineties, and its architecture is quite limited.

The Generic Modeling Environment (GME) is a platform for creating tools, where the presentation needs to be close to the domain [176, 177]. GME has a mechanism for

[5]In order not to mix the hand-written code with the generated code, MS DSL uses partial classes, i.e., classes, whose parts can be spread among several files.

checking domain constraints. GME also permits adding new components via the COM technology[6].

Meta-tools DiaGen and DiaMeta [178] are focused on building tools that are based on graph diagrams.

Pounamu is a meta-tool that allows the user to prototype graphical tools rapidly [179]. The Pounamu team has also developed a set of Eclipse plugins called Marama intended to support DSM in the Eclipse platform. Marama Meta-Tools are intended to replace Pounamu [7].

DoME (Domain Modeling Environment) is a quite old project [180], which coincides with TDA in the intention to provide a platform-neutral modelling environment. DoME has limited capabilities. It seems that the project is not being maintained now.

There are also transformation-based platforms such as ViatraDSM Framework [127], Tiger [181], and ATOM3 [182]. In these platforms, the domain is mapped to the presentation by means of model transformations. This gives more freedom for defining mappings, while keeping the mappings in the model-driven world. The ViatraDSM team is now maintaining the project called VIATRA2 [112, 113].

From numerous transformation-based platforms, METAclipse [183] has to be emphasized, since it was developed at IMCS-UL. METAclipse is an Eclipse plugin and uses four fixed engines provided by Eclipse. METAclipse defines a presentation metamodel that describes the functionality of these four engines. On certain presentation events, METAclipse calls MOLA transformations, which implement the business logic of the tool. METAclipse has been successfully used to implement the MOLA editor, which has been used to define transformations for the RedSeeDS project. Section 4.5 explains the historical relation between METAclipse and TDA.

The MOFLON framework [184] is intended to be fully compliant with the MOF [46] standard. MOFLON uses Triple Graph Grammars (TGG) to define transformations in a declarative way.

AMMA [185] and openArchitectureWare [186] provide useful "bricks" for model-driven development. As TDA, these platforms put model transformations to the centre of model-driven development, although in a different context: they are mainly focused on code generation. There are interesting ideas of megamodelling and model weaving in AMMA [187]. Megamodelling is aimed to describe connected models within one terminal model

[6] COM stands for Component Object Model, a Microsoft technology that allows components to communicate via binary interfaces.

[7] https://wiki.auckland.ac.nz/display/csidst/Marama+Meta-tools

called a megamodel. The model weaving, in its turn, allows to establish links between metamodels (to "weave" them).

4.3 Extensibility and Customization Obstacles

While existing tool-building platforms try to factor out certain functionality for certain classes of DSLs, they cannot factor out all possible functionality. That is why extensibility and customization of such platforms is an important issue. Considering this issue, the following three obstacles can be identified:

1. *Extended functionality is intended to be implemented either in the OOP "world", or in the model-driven "world", but not in both.* However, a DSL may require features from both "worlds". For instance, features like model validation and complex mappings can naturally be implemented in the model-driven "world", while features like connection to the database is easier to implement in the traditional OOP "world".

2. *A deep understanding of the platform is required, when a new feature not provided by the platform is to be added.* Assume, that we are going to create some music notation-based DSL. While we are ready to provide a module for displaying sheet music (which most likely is not provided by tool-building platforms), we want to utilize other already existing features and the infrastructure of the chosen platform. Incorporating the sheet music presentation into an existing platform either requires the deep knowledge of the internals of this platform, or the presentation has to reside outside the platform, and the communication between the platform and the presentation has to be ensured.

3. *A certain environment is involved.* For example, GMF uses GMF Runtime on top of the Eclipse platform, MS DSL uses Microsoft DSL Tools Framework Assemblies on top of Visual Studio, and MetaEdit+ uses itself or its lite runtime version called MetaEdit+ Modeller. The platform prescribes which modelling (and other) technologies to use (e.g., GMF uses EMF and Java; MS DSL uses "the Store" and .NET), and the DSL tool becomes environment-dependent. Also, if some feature has been implemented for one environment (like an excellent TwoUse toolkit for bridging models and ontologies [78]), it is difficult to reuse it in another environment.

TDA overcomes these obstacles in the following way.

[Obstacle 1] TDA acts as a hub by connecting features implemented in both traditional code world (engines) and model-driven world (transformations). To ensure this, TDA provides a common communication mechanism.

[Obstacle 2] In TDA, engines are described by interface metamodels that represent their essential functionality. Interface metamodels can be used as documentation. When a new engine is added, it must have the corresponding interface metamodel defined. Developing an interface metamodel and establishing the link between it and the engine requires additional efforts, but this task, when once accomplished, gives continuous benefits. All the engines can be accessed in a uniform way, without diving into their technical details.

[Obstacle 3] TDA lies outside a particular environment (e.g., Eclipse or Microsoft Visual Studio). The main window (application), to which other windows are attached, is not prescribed; it is provided by a replaceable Environment Engine (explained in Chapter 8 of this thesis). TDA uses abstraction layers to call engines and transformations, and to access repositories in an environment-independent way.

Using abstraction layers to factor out environment-specific aspects resembles how the operating system factors out device management. Besides, TDA provides useful services for components — this resembles how the operating system provides useful services for applications. That is why TDA may be viewed as an *operating system superstructure for model-driven software*.

4.4 TDA as a Foundation for DSL Tools

Figure 4.2 depicts three possible ways of building a model-driven tool based on the TDA foundation. These ways are:

- Write all the transformations implementing the functionality of a tool manually (Tool 1 in Figure 4.2). This is the most effort-consuming way.
- Develop a tool definition framework, TDF, implementing typical functionality for certain class of tools (e.g., tools based on graph-like diagrams). Tool-specific aspects can be expressed as an instance of some tool definition metamodel.

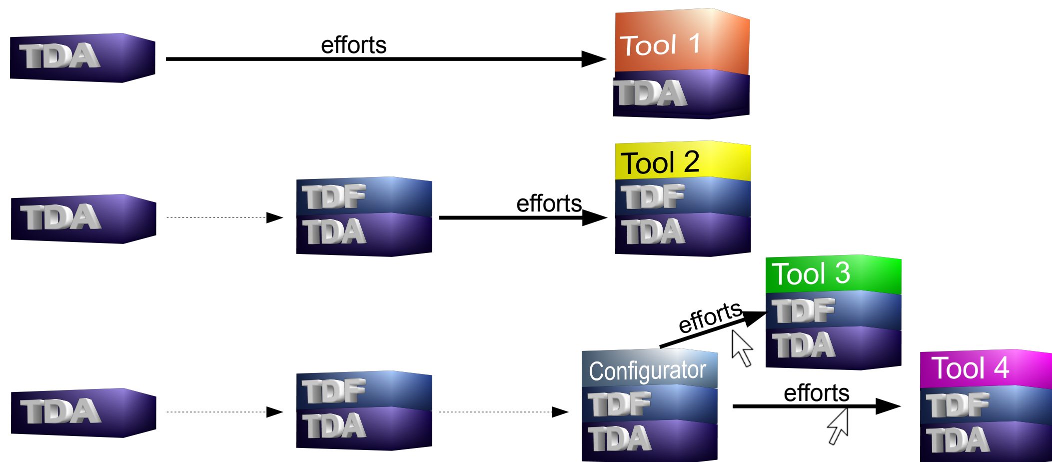


Figure 4.2: TDA as a foundation for building domain-specific tools.

- In addition to TDF, write a special DSL tool (the "Configurator" in Figure 4.2), which can be used to configure TDF graphically for the needs of the DSL tool being developed. Then, the configured tool (e.g., Tool 3 or Tool 4) is interpreted by TDF.

The last approach have been implemented in the TDA-based tool-building platform GRAF for developing DSL tools based on graph-like diagrams [4, 7]. GRAF was developed at IMCS-UL. GRAF consists of TDA, TDF and the Configurator. If the business logic implemented in TDF needs to be augmented, the extension point mechanism can be used to extend the default behaviour of a configured tool.

The topic of building model-driven world on the TDA foundation is beyond the scope of this thesis.

TDA proved to be a good foundation for building domain-specific tools. By means of GRAF [7, 4] several tools including ProMod and BiLingva (business process management tools), GradeTwo (a UML tool), ViziQuer (a semantic data graphical query tool) and OwlGrEd (graphical OWL editor) have been developed [3, 33, 34, 35]. J. Bārzdiņš, K. Čerāns, L. Lāce, R. Liepiņš, E. Rencis, S. Rikačovs, A. Sproģis, A. Zariņš, M. Zviedris, and others (including myself) have contributed to these tools as well as to GRAF itself.

An interesting approach for building domain-specific tools has been presented in the paper "Towards Open Graphical Tool-Building Framework" (I am a co-author) [11]. The paper introduces the metamodel specialization operation. This operation is used to define sub-metamodels of a given metamodel in a way similar to defining subclasses of a given class. Metamodel specialization can be used to define a metamodel that is a sub-metamodel of one or more interface metamodels in TDA. When carefully designed, this sub-metamodel can be used as a definition of a domain-specific tool. Although I and

the co-authors believe that by using this approach tools can be easily defined, metamodel specialization has certain limitations, e.g., a sub-metamodel must follow the design of the base metamodel(s).

4.5 The History of TDA

In 1964 IBM announced System/360, a family of mainframes, where the architecture was separated from the implementation. The Soviet Union copied the architecture and developed machines known as "ЕС ЭВМ" ("Единая система электронных вычислительных машин"). Soviet Secret Service managed to obtain also flowcharts and source code of IBM software, including Conversational Remote Job Entry (CRJE), a part of the IBM System/360 operating system. Printing out such flowcharts on a text printer by means of "x" symbols was the first experience of working with graphics in IMCS-UL. It was in 1970-ties.

By that time, J. Bičevskis and others worked on the system called "Система макрокоманд обработки данных" ("СМОД") [188, 189]. Nowadays we would say that this was a textual domain-specific language. In the late 1980-ties IMCS-UL had an experience of building a graphical editor by means of SDL (Specification and Description Language), a language for specifying reactive and distributed systems [190].

One of the most known tools developed at IMCS-UL is GRADE. Although research and development of GRADE started in 1986, the GRADE history page states [191]:

> "GRADE origins can be traced back to work done by C. Everhart at Bell Laboratories and Teledyne Brown Engineering, Rudolf Duschl of Siemens and Professor Manfred Broy at the University of Munich for which he received the prestigious Leibniz Prize in 1994."

On October 2, 1990, Jānis Gobiņš, a founder of Infologistik, Inc., presented GRADE ideas at IMCS-UL. The work on the "real" GRADE began (there were already some GRADE prototypes before 1990).

In 1994, IMCS-UL cooperated with Exigen Services, when developing the Exigen Business Modeler tool.

In 2003, after OMG initiated MDA, IMCS-UL started to develop the MOLA transformation language.

In 2006-2008 two tool building platforms (with different purposes in mind) have been developed at IMCS-UL. One of them was METAclipse [183] and another was GrTP [192].

In 2008, at a seminar, where me, Jānis Bārzdiņš and Edgars Rencis discussed GrTP, the idea of TDA was born. I was fascinated with the idea, and became the main author of the first publication on TDA [1]. Then I started to develop the idea, which can now be read in this thesis. It must be noted that the birth and recognition of TDA in 2008 would be impossible without the previous experience hold by IMCS-UL and the laborious work performed by IMCS-UL staff (see the Acknowledgements section in this thesis).

In the first half of 2009, the first version of TDA has been implemented. It was based on the GrTP platform. Currently, the work on the next major TDA version, TDA 2.0, is in progress. This thesis describes TDA 2.0. For the status of TDA 2.0 development, refer to `http://tda.lumii.lv`.

Part II

TDA Kernel And Engines

Chapter 5

The Kernel Of TDA

As was depicted in Figure 3.3 on page 49, TDA Kernel is an intermediate between model repositories and their clients (transformations and engines). TDA Kernel implements RAAPI and forwards RAAPI calls to actual repositories. This gives TDA Kernel the ability to intercept RAAPI calls and to implement certain useful services.

Section 5.1 presents RAAPI. Section 5.2 explains three APIs used by adapters for repositories, adapters for engines, and adapters for transformations, respectively. Section 5.3 presents the core of TDA Kernel Metamodel and explains the basic services provided by TDA Kernel.

The next two chapters (Chapters 6 and 7) explain two advanced TDA Kernel services and the corresponding extensions of TDA Kernel Metamodel. These two services are the undo/redo mechanism and the multi-repository mechanism.

5.1 The Fundamentals of RAAPI

RAAPI is an API for accessing model elements. As was mentioned in Section 1.1, models are usually organized in multiple meta-levels. There can be linguistic and ontological meta-levels. First, I explain how to organize those different meta-levels. Then, I give a summary on RAAPI.

Organizing Meta-Levels into Quasi-Meta-Levels

To deal with numerous meta-levels in a common way, I organize meta-levels into two groups. The first group will contain the fixed (and, thus, read-only) linguistic meta-metamodel (at Level M3) of some technical space. The second group will contain all

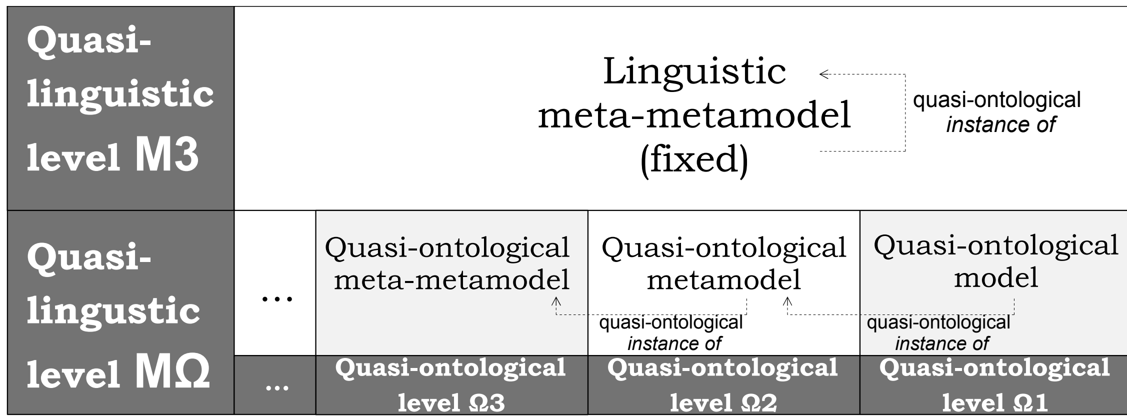


Figure 5.1: The organization of meta-levels into the two quasi-linguistic meta-levels M3 and MΩ, where MΩ contains several quasi-ontological meta-levels.

other meta-levels that are possible in that technical space. The meta-levels from the second group are not fixed.

> *Example.* In the Grammarware technical space, there is a fixed meta-metamodel at M3 called EBNF [39]. EBNF is able to describe grammars at M2 (variable w.r.t. M3), and grammars at M2 are able to describe char sequences at M1 (variable w.r.t. M2). Thus, EBNF belongs to the first group, while grammars (Level M2) and the corresponding char sequences (Level M1) belong to the second group.

I call these two groups **the quasi-linguistic meta-level M3** and **the quasi-linguistic meta-level MΩ**, respectively. The quasi-linguistic level M3 is fixed. In most cases, it will contain the linguistic meta-metamodel of some technical space (hence, the name "M3"). The level MΩ will contain all other meta-levels that are variable w.r.t. to M3 (either directly, or via other meta-levels). I call these variable levels **quasi-ontological meta-levels** (in reality, they may be either linguistic, or ontological meta-levels; see below). Figure 5.1 illustrates the organization of quasi-meta-levels. Figure 5.1 is similar to meta-levels depicted in Figures 1.1(c) and (d) on page 30. The difference is that now we have exactly two quasi-linguistic meta-levels.

Such an organization of quasi-meta-levels has the following benefits:

- It separates the fixed part (M3) from the variable part. The fixed part does not need to be stored in a model repository. Model repositories from Table 1.1 on page 28 usually provide a means to store only the variable part, i.e., models lying at MΩ.

Quasi-linguistic level M3	Linguistic meta-level M3	
Quasi-lingustic level MΩ	Linguistic meta-level M2	Linguistic meta-level M1
	Quasi-ontological level Ω2	Quasi-ontological level Ω1

Figure 5.2: Levels M1-M3 from the three-level conjecture represented as quasi-meta-levels.

- The three linguistic meta-levels (from the three-level conjecture) used by technical spaces can be easily re-organized as quasi-meta-levels as depicted in Figure 5.2 on the following page.

- Depending on a particular technical space, the meta-model at M3 can describe either M2 only (M2 then describes M1), or both M2 and M1 at the same time. Examples for the first case, where M2 and M1 are strictly separated, are EMOF, ECore, EBNF, and XML. Examples for the second case, where M2 and M1 co-operate, are the JR repository (which contains the meta-classes *Class* and *Object* directly at M3) and the OWL language, which can be used to describe ontologies containing classes and individuals at the same time (individuals may be required to specify certain OWL restrictions). Quasi-meta-levels are suitable for both cases.

- When a particular repository or technical space supports infinitely many meta-levels (e.g., JR or OWL Full), all these variable meta-levels can be considered quasi-ontological meta-levels. This ensures the same organization of meta-levels for OWL Full (that supports multiple ontological meta-levels) as well as for strict OWL variants, where classes (at M2) and individuals (at M1) are disjoint (Figure 5.3).

The Essentials of RAAPI

RAAPI is a procedural API consisting of primitive operations on model elements such as creating a class with the given name, creating an instance of the given class, creating an association between two classes, creating a link (corresponding to the given association) between the two given objects, etc. Taking into a consideration Šostaks' conjecture,

<table>
<tr><td>Quasi-linguistic level M3</td><td colspan="4">OWL Full
(M3)</td></tr>
<tr><td rowspan="2">Quasi-lingustic level MΩ</td><td>...</td><td>OWL meta-classes (ontological)</td><td>OWL classes</td><td>OWL individuals</td></tr>
<tr><td>...</td><td>Quasi-ontological level Ω3</td><td>Quasi-ontological level Ω2</td><td>Quasi-ontological level Ω1</td></tr>
</table>

Figure 5.3: The organization of multiple ontological meta-levels of OWL Full.

RAAPI mainly consists of functions, which can access two adjacent meta-levels (either quasi-linguistic, or quasi-ontological).

The essence of primitive operations working on two adjacent meta-levels is borrowed from APIs found in the repositories developed by K. Podnieks (the MII_REP model repository [193][1]) and M. Opmanis (the JR repository [55]). Besides numerous adaptations and cosmetic changes, for certain functions I assigned new semantics to support the quasi-linguistic level M3. I have also added functions for managing relations between the elements at M3 and the elements at MΩ. The RAAPI documentation can be found in Appendix A. The up-to-date version of RAAPI can be found at `http://tda.lumii.lv/raapi.html`.

The points below summarize the main characteristics of RAAPI.

- In RAAPI, model elements are referenced by 64-bit integers, which may represent indexes or pointers to elements.
- Every repository adapter (which implements RAAPI) has to support at least two quasi-ontological meta-levels: usually they are the levels M1 and M2 of the corresponding technical space. Additional supported meta-levels (not counting M3) become additional quasi-ontological meta-levels (as in the case of RDF and OWL Full).
- Switching between meta-levels can be performed by passing a reference to an element at one meta-level to an RAAPI function, which expects a reference to an element at another meta-level. For instance, a reference to a class may be passed to an RAAPI

[1] codenamed “OUR” in this paper

operation, which expects a reference to an object. This trick can also be used to mix multiple meta-levels (where the underlying repository supports that).

- RAAPI supports multiple classification and dynamic reclassification found in OWL and SMOF (i.e., one object can belong to many classes). Multiple inheritance is supported as well.

- To create an adapter implementing RAAPI for a particular repository, only essential functions have to be implemented. For instance, the given repository may be able to iterate either through proper attributes of a class, or through all attributes (including derived). An adapter may implement only one of these cases, and TDA Kernel will implement the other. Also, if the given repository does not support some capability (e.g., support for multiple classification), the adapter may either simulate it, or to discard it. The latter case resembles how certain file system operations are discarded for some file systems in UNIX (e.g., when UNIX-style file permissions are not supported on a mounted file system).

- Repository adapters are not required to provide access to M3, when implementing RAAPI operations. However, M3 may be useful, if there is a need to access technical-space-specific features that are defined at M3, but are not directly supported by RAAPI.

- If a repository adapter provides access to M3, it has to implement RAAPI functions in such a way that M3 behaves like a quasi-ontological instance of itself (see M3 in Figure 5.1 on page 67). This can be used to access M3 features that would not be accessible otherwise. For instance, RAAPI does not support EMOF-like operations in classes directly. Still, RAAPI can be used to access the meta-class EMOF::Operation at M3. By traversing quasi-ontological instances of this meta-class, operations found in EMOF itself can be discovered. To create/obtain operations for classes at MΩ, linguistic instances of EMOF::Operation can be created/traversed.
  The same approach can be used to support generics found in ECore since EMF 2.4[2].

- If a repository adapter provides access to M3, it is assumed that each element from MΩ may have only one linguistic type at M3. This type cannot be dynamically changed.

[2] See `http://www.kermeta.org/docs/org.kermeta.ecore.documentation/build/html.chunked/Ecore-MDK/ch02.html`.

- RAAPI is technically simple: it is procedural and uses a restricted set of data types. Thus, RAAPI can be easily adapted to and used from different platforms and programming languages. Currently, RAAPI is available for the Java platform, C/C++ (dynamic library calls), the .NET platform, and for the CORBA middleware platform.

5.2 Adapters

This section presents APIs used by adapters for repositories, adapters for engines and adapters for transformations.

Adapters for Repositories

For each repository type (ECore, JR, JGraLab, OWLIM, etc.) that needs to be used within TDA, an adapter must be created. Each adapter for repository must implement the *IRepository* interface. This interface is a union of two interfaces:

- RAAPI, which consists of operations on elements in the repository;
- *IRepositoryManagement*[3], which contains operations related to the repository itself, e.g., *open*, *close*, and operations for performing save. The two important highlights of the *IRepositoryManagement* interface are:
  - TDA Kernel can work with multiple repositories, and a failure can occur in each of them during save. For this reason, there are three functions for saving the repository, namely, *startSave*, *finishSave*, and *cancelSave*. Having these functions, the save process is performed in two steps. First, for each repository *startSave* is called. If a failure occurs, repositories are rolled-back via *cancelSave* calls. Otherwise, *finishSave* is called for each repository. This process is similar to the two-phase commit protocol [194].
  - TDA Kernel uses URIs (Uniform Resource Identifiers) to specify locations of repositories [195]. A repository URI is a string consisting of the corresponding adapter name, followed by a colon that is then followed by the repository-specific location, for instance, "`jr:/repositories/data1/`" ("`jr`" is an adapter

[3]For a detailed description of this interface, refer to Appendix A or to the TDA homepage at `http://tda.lumii.lv/raapi.html`.

name, and "`/repositories/data1`" is a location in a file system). When a repository is accessed via TDA Kernel, the location has to be specified as a full URI. TDA Kernel then extracts the adapter name and passes the remaining part (the repository-specific location) to the corresponding adapter.

Note. TDA Kernel also implements the *IRepository* interface.

Adapters for Engines

For each technology platform (Java, DLL, .NET, etc.) used to implement engines, an adapter must be created. Each such adapter must implement the *IEngineAdapter* interface, which consists of the following functions:

- `boolean load (String name, RAAPI rappi);`
  Finds the engine with the given name and loads it[4]. The search is platform-specific: a Java adapter should search for a jar implementing the engine, a DLL[5] adapter should search for a DLL, and so on. The adapter has to use certain technologies to ensure the communication with the engine. For instance, Java Native Interface (JNI) can be used to access DLLs from Java. Or, some kind of inter-process communication can be used to access engines running as parallel processes. The adapter has to pass a pointer to RAAPI (or to an RAAPI wrapper) to the engine. The engine will use this pointer to access its own interface metamodel and the corresponding model in the repository. If the engine is being loaded for the first time, then its interface metamodel (and, probably, some instances) have to be put into the repository. The engine can associate itself with certain events (e.g., with *ProjectOpenedEvent* from Environment Engine Metamodel explained in Chapter 8).

- `boolean executeCommand (long r);`
  Executes the command specified as a 64-bit reference `r` to the corresponding command object.

- `boolean handleEvent(long id);`
  Handles an event specified as a 64-bit reference `r` to the corresponding event object. This function is used only when the engine specifies itself as event handler in an *on*-attribute for some event.

[4]Engines are identified by their names (not URIs).
[5]dynamic-link library

- `void unload ();`

  Unloads the engine in a platform-specific way.

Adapters for Transformations

Each transformation language used within TDA needs an adapter. Each adapter for transformations must implement the *ITransformationAdapter* interface consisting of the following functions:

- `boolean load (RAAPI raapi);`

  Loads the corresponding tool or library for launching model transformations written in the language the adapter supports.

- `boolean launchTransformation (String location, long argument);`

  Launches the transformation at the given location. The argument specifies either the event object the transformation needs to handle, or an *ExecuteTransformation-Command* instance used to launch this transformation.

- `void unload();`

  Unloads the infrastructure used to launch transformations.

Like repositories, transformations are identified by means of URIs. A transformation URI is a string consisting of the corresponding adapter name, followed by a colon, which is then followed by the location (name) of the transformation, for instance, "`atl:/transformations/copy_model.atl`" ("`atl`" is the adapter name and "`/transformations/copy_model.atl`" is the location of the transformation code). Such URIs are used in *on*-attributes to specify event handling transformations. Note: when an engine is used to handle an event, the value of the *on*-attribute must be constructed from a special adapter name "`engine`" and the engine name, for instance, "`engine:DialogEngine`".

5.3 The Basic Services of TDA Kernel

The core of TDA Kernel Metamodel is depicted in Figure 5.4, where the basic services of TDA Kernel are described. The undo/redo mechanism and the multi-repository mechanism, which are also implemented in TDA Kernel, extend this core.

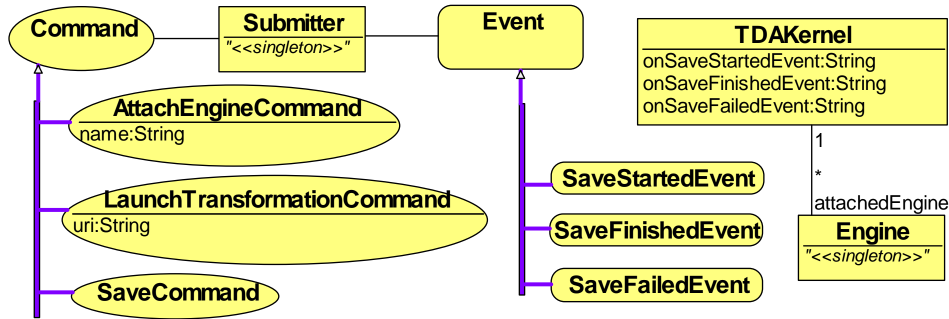


Figure 5.4: The core of TDA Kernel Metamodel.

TDA Kernel Metamodel defines the *Command* and *Event* classes, which are the main superclasses for commands and events used in the TDA communication mechanism. As was explained in Section 3.3, to manage events and commands, TDA Kernel intercepts the link creation operation between them and the submitter.

When a TDA Kernel command is issued, TDA Kernel simply executes that command. Figure 5.4 lists three TDA Kernel commands:

- The *AttachEngineCommand* is used to attach an engine. Each engine needs to be attached only once. After closing and re-opening the repository, TDA Kernel will load previously attached engines automatically. To simplify this process, the following convention has to be followed: each engine in its interface metamodel should define a singleton class representing the engine. It has to be a subclass of *Engine*, with the name equal to the engine name. TDA Kernel maintains the list of attached engines by means of the *attachedEngine* association between the classes *TDAKernel* and *Engine*.

  To attach an engine, TDA Kernel consequently passes the engine name to all known adapters for engines until some adapter eventually finds the engine and loads it and its interface metamodel.

- The *LaunchTransformationCommand* can be used to launch the given transformation, which is specified by its URI.

- The *SaveCommand* is used to save the repository (repositories) on demand. This command emits *SaveStartedEvent*, then saves all active repositories, and, finally, emits a *SaveFinishedEvent* or a *SaveFailedEvent* depending on whether all the repositories indicated that the save was successful.

When a command for some engine is issued, TDA Kernel passes it to the adapter of the corresponding engine (when a command is issued, the engine and its adapter are already loaded).

When an event is emitted (it may be either a TDA Kernel event, or an event of some engine), TDA Kernel passes this event to the corresponding transformation or engine specified in the *on*-attribute for this event. First, the adapter name is extracted from the value of the *on*-attribute. If the adapter name is "engine", then the event is passed to the engine, whose name is taken from the remaining part of the attribute value (the aforementioned function `handleEvent` is used to handle this event). Otherwise, the adapter is a transformation adapter. In this case, the adapter-specific transformation name is taken from the *on*-attribute value and passed to the `launchTransformation` function of the corresponding adapter for transformations.

5.4 Related Work

The need for a common API for accessing different types of repositories has already been realized by some teams. For example, ATL Virtual Machine [98], Epsilon Model Connectivity level (EML) [99, 100], and the CDO [54] repository use some kind of common API, which plays the same role as RAAPI in the proposed multi-repository mechanism. In contrast to ATL and EML (which use a few API functions being able to work with lists) as well as ECore (which uses object-oriented API), RAAPI is procedural and uses only primitive data types. Also, RAAPI was designed to support SCMOF capabilities.

Repository adapters used by TDA Kernel resemble how ModelBus uses tool adapters to connect multiple modelling tools [196, 197]. In ModelBus, adapters are mainly used to access data according to the check-in/check-out principle, while RAAPI adapters are intended to perform their functions on-the-fly. To implement repository adapters for certain technical spaces, numerous existing technologies such as ORM[6]-technologies (Java Persistence API[7], .NET Persistence API[8]), D2RQ [132, 133], object-oriented databases [198], etc. can be utilized. If on-the-fly data access is impossible, the check-in/check-out principle can be used as well.

[6]object-relational mapping

[7]`http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html`

[8]`http://www.npersistence.org/`

Chapter 6

The Multi-Repository Mechanism

The motivation to support multiple model repositories from different technical spaces (TS) is driven by the following considerations:

- One TS and its repositories can be more suitable for the given purpose and more convenient than another. This resembles how one programming language can be more suitable for certain applications than another.
- A person can be more familiar with (i.e., have skills and knowledge in) one TS than with (in) another. If the efforts to study a new TS are big enough, it may be reasonable to stay in a more familiar TS.
- A capability not available in a desired TS can be borrowed from another TS that implements that capability. This encourages "more cooperation than competition among alternative technologies" [40].

To ensure interoperability between different technical spaces, Bézivin et al. suggest using projectors and extractors — offline transformations of models between TS's. In this thesis I propose a mechanism for working with multiple repositories (which may belong to the same or different TS's) online. The idea is to mount several repositories into packages in the same way as file systems are mounted into directories in UNIX. Deep copying of the data is not required, and all the changes in models become visible immediately.

While mounted repositories can be accessed from their packages (with the possibility to create relations between their elements), certain manipulations with packages can be performed as well (for instance, two packages may be merged). Mounting into packages and manipulating the packages — these are the two pillars of the proposed multi-repository mechanism.

6.1 Multiple Repositories as a Single Repository

TDA Kernel represents multiple repositories as a single repository to model transformation and engines (I call them *clients* of the multi-repository mechanism). One of the repositories (called the **pivot repository**) is used to store the information about inter-repository relations. The pivot repository acts as a fully fledged repository as well.

This design choice is based on the following considerations:

- Clients can access multiple repositories in a uniform way using a single RAAPI.
- Clients do not need to switch between different repositories or to specify the desired repository as an argument for each operation on a model. For example, to create a link between two objects from different repositories, a client may assume that the objects are in the same repository: it is the responsibility of TDA Kernel to store and handle this inter-repository link correctly.

6.2 Packages as Mount Points

A **package** is a group of model elements similar to a UML package.[1] I will consider only the case when all the packages form a tree. This resembles how directories are usually organized as trees in a file system.

TDA Kernel maintains a rooted tree of packages called the **kernel package tree**. Each kernel package is associated with a package in some repository. The simple (unqualified) names of kernel packages are usually equal to the simple names of the corresponding repository packages, but can be changed at runtime. The names of kernel packages are used by the clients: to refer to a package, a client specifies its fully qualified name consisting of simple names starting from the root kernel package.

Initially, the kernel package tree corresponds to the package structure of the pivot repository. When an additional repository is mounted, a new package (a **mount point**) is added to the kernel package tree. The content of the repository will be available via this mount point. For instance, a class can be accessed by concatenating the fully qualified mount point name with the fully qualified class name in the mounted repository.

If a repository can store several models, there are two options:

- treat each model as a separate repository and mount it into a separate package;

[1] If packages are not supported by a particular repository, they can be encoded directly in class names (e.g., "`Package::SubPackage::Class`").

- mount the whole repository with all its models at once, but treat each model as a package inside that repository. While it may seem more convenient, there are two shortcomings. First, the repository adapter becomes more complex since it has to perform all necessary actions to represent models as packages. Second, if the repository does not support relations between models, TDA Kernel will not support them as well.

6.3 Proxy References

TDA Kernel deals with elements from different repositories, but needs to represent them as if they were in a single (multi-packaged) repository. This can be performed by introducing **proxy references** to elements. In RAAPI, model elements are referenced by 64-bit integers, which may represent indexes or pointers to elements. Proxy references are also 64-bit RAAPI references, but TDA Kernel ensures they are unique among all the repositories. TDA Kernel maps each proxy reference to the corresponding repository and to the corresponding reference in that repository (proper repository references are called **domestic references**). However, the mapping to the repository is not direct: each proxy reference maps to a package in the kernel package tree, and each package maps to a repository. Such design choice permits changing the repository associated with a package at runtime.

When a particular repository returns a domestic reference, TDA Kernel either creates a new proxy reference, or returns a previously created one. For this, TDA Kernel maintains a reverse map, which maps pairs <package, domestic reference> to proxy references. Translation between proxy and domestic references is performed by TDA Kernel automatically, thus, repositories do not need to be aware of proxy references (although, they can, if they need to).

The clients assume that there is only one (multi-packaged) repository like UNIX programs assume there is only one file system. When TDA Kernel processes an RAAPI operation, the following simple algorithm is used to determine, in which repository the changes are to be stored:

- if all the elements involved are from the same repository, TDA Kernel forwards the call to that repository through the corresponding adapter;

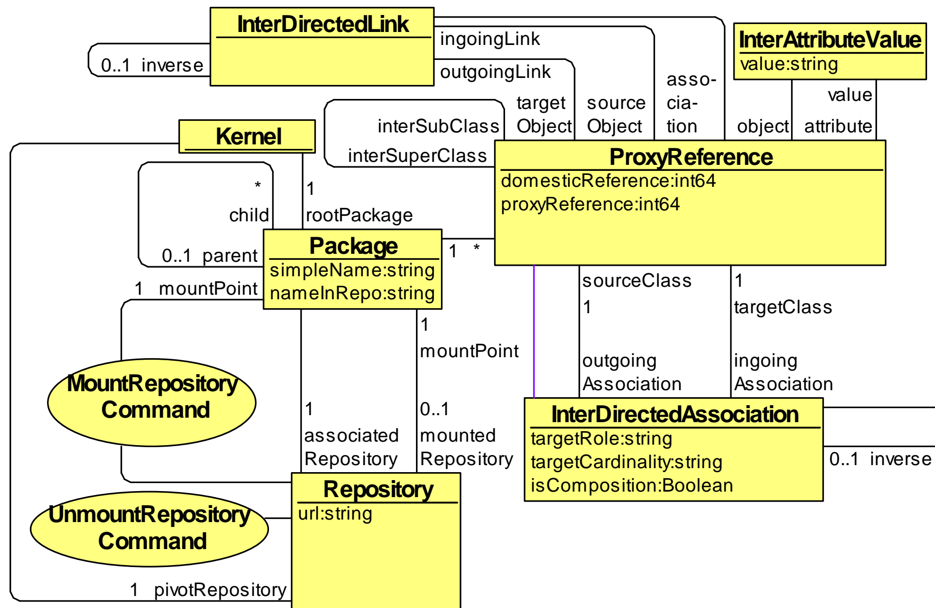


Figure 6.1: An extension to TDA Kernel Metamodel for the multi-repository mechanism.

- if the elements involved are from different repositories (e.g., an association between two classes from different repositories is being created), TDA Kernel treats this operation as an inter-repository change and stores it in the pivot repository.

TDA Kernel stores information about the kernel package tree, mounted repositories and inter-repository relations according to the extension of TDA Kernel Metamodel depicted in Figure 6.1. Notice that the two additional TDA Kernel commands are introduced for the multi-repository mechanism: *MountRepositoryCommand* and *UnmountRepositoryCommand*.

TDA Kernel performs only minimal constraint checking for inter-repository relations. More sophisticated constraint checking can be performed by introducing an additional layer over RAAPI, or by means of external model transformations (constraint checking lies beyond the scope of this thesis).

6.4 Manipulating the Packages

Instead of forwarding RAAPI calls to a particular physical repository, a repository adapter can also represent a *computable* (or *virtual*) *repository*, which does not exist physically, but relies on the data from other repositories. When a virtual repository is mounted into some kernel package, its repository adapter transforms RAAPI operations to operations on other kernel packages on-the-fly. I use the term **on-the-fly model transformation** to

denote such translations of RAAPI operations performed by virtual repositories. On-the-fly model transformations can be used to implement certain manipulations with packages. Here are some examples.

Virtual Copy. Having an existing package P, the "virtual copy" manipulation creates a virtual package P', which acts as a copy of P. However, in reality, no data are copied! This manipulation can be implemented by mounting two special virtual repositories into P and P', but keeping pointer to the old package P. When the old data are accessed, both repositories forward the call to the old P. At the same time, both virtual repositories record changes in new P and P', and provide an illusion that P and P' are being modified independently. One of the use cases for "virtual copy" is implementing transactions. First, a virtual copy of a package is created to fix its state without deep copying of the data. Then, some transactional changes are performed, and, finally, these changes are either discarded, or stored in (committed to) the old P. Another use case is providing the space for semantic reasoning. If a package contains an ontology, its virtual copy can be created, and a semantic reasoner can be launched on that copy. When the reasoner finishes, the original package and its copy will contain the data before and the data after the reasoning, respectively.

Virtual Merge. A virtual repository can be mounted into an existing package P, keeping a pointer to the old P and a pointer to some other package Q. The repository provides an illusion that P and Q are merged (in the sense of UML package merge).

Introducing derived (calculated) properties and relations. To create a calculated relation between classes A and B from a package P, another virtual repository is introduced. This repository contains classes with the same names as A and B, and also adds the required calculated association, which is computed on-the-fly, when this virtual repository is accessed via RAAPI. To complete the picture, the repository has to be mounted into some package Q, and then P and Q have to be virtually merged.

Symbolic links. Although the kernel package tree is a tree, packages can be organized into a graph-like structure by means of UNIX-style symbolic links. A symbolic link L on package P may be implemented by introducing a virtual repository, which is mounted into L, but forwards all RAAPI calls to P.

Using volatile temporary data in models. A volatile temporary repository, whose content is lost on exit, may be introduced. Temporary data can be transparently combined with the persistent data by means of "virtual merge".

Indexing of model elements. Certain calculated relations can just reorder the elements from the relations they rely on. Thus, when traversing the corresponding calculated element list, elements will appear in the desired order. To implement this behaviour, the virtual repository can use indexes internally.

Views on metamodels. One metamodel (say, a complex one) can be represented as another metamodel (e.g., simpler) by means of views. Like file systems can be read-write and read-only (e.g., CD-ROM), views can be read-write (implementing bi-directional on-the-fly transformations) and read-only (implementing unidirectional on-the-fly transformations). Read-only views simply discard modificating RAAPI operations.

6.5 Using Multiple Repositories in DSL Tool-Building

Besides offering powerful manipulations with packages (such as manipulations listed above), the multi-repository mechanism can be useful for the following use-cases.

- A tool-building platform (such as TDA-based platform GRAF [7]) can define a tool T and store its definition in a repository R_T. Without the multi-repository mechanism, for each task (project), the tool definition and the actual task-specific data have to share the same repository. Thus, the content of R_T needs to be copied for each new task (project)[2]. With the multi-repository mechanism, the content of R_T does not need to be copied; it can be mounted. If certain modifications to the definition of T are required, they have to be made only in R_T.
- Assume there is a need to implement a tool offering a multi-user mechanism, where
  - the model is logically split into multiple shared parts (e.g., a model representing multiple graph diagrams);
  - at times, some user can take some shared part (e.g., a diagram) for modification (this part is being marked as locked during the modification).

  Such multi-user mechanism can be implemented by storing each shared part in its own repository on a shared server[3].

[2] The current implementation of GRAF uses this approach.

[3] The current implementation of the multi-user mechanism used in the TDA-based tool ProMod serializes parts of a model as text files, which are transferred to/from the SVN server. With the multi-repository mechanism, it would be possible to store those parts in true model repositories. Such repositories could be accessed either directly at a server, or via check-in/check-out without the need to serialize them as text files.

6.6 Related Work

Certain research on model merge is being performed, but in a different context than the proposed “virtual merge” operation. For instance, Epsilon Merge Language is an excellent language intended for describing merge-like operations on models. These operations are then executed in the offline mode (not on-the-fly) [99, 100]. MOF 2 for Java implements the merge capability, but the goal was MOF 2 compliance, not merging different repositories [52].

The live model transformation framework proposed by the VIATRA team treats complex model changes as elementary changes [199, 113]. This resembles on-the-fly transformations, which could be considered split into a set of elementary RAAPI operations.

The “virtual copy” operation is based on the concept of worlds, which is a way to control side effects arising of using the same data from different parts of the program [200].

Interesting ideas about read-only views have been presented by E. Rencis [201]. His mechanism modifies the code of a model transformation in such a way that the view is executed on-the-fly. On-the-fly transformations mentioned in this paper are intended to perform the same job, but without modifying client code (thus, on-the-fly transformations are not tied to a particular transformation language, but only to RAAPI). Ideas and code fragments provided by E. Rencis can be adapted to generate code for virtual repositories implementing views through RAAPI.

Chapter 7

The Undo/Redo Mechanism

The TDA undo/redo mechanism is common for all systems and tools using TDA as a foundation. When a TDA-based system is being developed, usually there is no need to think about undo. The undo/redo capabilities are "miraculously" provided by TDA Kernel at runtime. In certain cases, however, the built-in undo mechanism has to be adjusted or extended. For instance, an engine dealing with exotic presentations may need a peculiar way of storing its states. There may also be multiple undo streams (e.g., in case of multiple diagrams). Furthermore, changes in one diagram may depend on changes in another. TDA undo/redo mechanism solves these issues.

The undo/redo mechanism benefits from the following TDA design choices:

- Since access to the model data is by means of TDA Kernel, it is easy to intercept model changes and to store them for undo.
- Since usually several model actions are performed at once, they should also be undone/redone at once. TDA communication mechanism helps to infer when a new "bundle" of actions starts (in most cases it starts on certain user events).
- Since engines and transformations rely on references pointing to model elements, these references should remain the same when undo and then redo are performed (and objects are deleted and re-created). The undo/redo mechanism can benefit of using proxy references (introduced in Chapter 6) to provide an illusion that deleted and re-created elements have always stayed in the repository.

7.1 Basics Notions

When a user works with a system, the system changes its states. In TDA, the system state includes states of repositories, states of engines and, perhaps, other external states. When a "bundle" of logically bound actions is applied to the system, the system goes to a new state. For example, the user adds a new class to a class diagram. This invokes several actions, which include creating a box for representing the class (i.e., adding a box object to the repository) and attaching it to the diagram (i.e., creating a repository link from the box object to the diagram object). Intermediate states such as the state, when the class has been created, but has not been attached to the diagram yet, are not considered.

The undo/redo mechanism allows the user to revoke the last "bundle" of actions to be able to return the system to the state, which is equivalent to the previous (non-intermediate) state. The system state after undo may be not the same as the original one, that is why I use the word "equivalent" here. For instance, during undo/redo, deleted objects may be re-created, and their identifiers in the repository may change. The transformations and engines can still assume that the recreated objects are the same as the original ones, since transformations and engines do not address these objects directly, but by means of proxy references. TDA Kernel ensures that these proxy references remain the same after undo/redo. However, they may point to other (recreated) objects.

I call the actions that change the state of the system **modificating actions**. Modificating actions can be divided into two groups:

- repository modificating actions such as creating/deleting a class, an object, an association, a link, etc., in a model repository; these are RAAPI modificating operations;
- external modificating actions: these can be actions that modify states of engines (outside the repository), actions that save information to external files, etc.

The first group of actions can be handled in a common (universal) way by TDA Kernel. It simply acts as a proxy by hooking repository modificating actions and storing them in the **undo history** (to describe the undo history, TDA Kernel Metamodel is extended by Undo Metamodel). The second group of actions cannot be handled universally, since new engines with new modificating actions not known in advance may be added to TDA. However, these engines can inform TDA about their changes that need to be registered in the undo history. To avoid the infinite recursion, the changes in the undo history itself are not traced.

Modificating actions are "bundled" in **transactions**. Since transactions represent differences between the system states, executing inverse actions of a transaction in reverse corresponds to undo. Re-executing the transaction in the original direction corresponds to redo.

TDA Kernel itself cannot determine to which states undo must be able to revert the system. Thus, the *CreateUndoCheckPointCommand* is defined in TDA Kernel Metamodel. This command can be used by engines and transformations to mark the current state as a checkpoint, i.e., the state, to which the system will be able to revert. At this moment, a new transaction is started, and the following modificating actions are being attached to this new transaction. The *CreateUndoCheckPointCommand* command is usually issued by engines on certain user events (e.g., events, which start some modification of a diagram). Thus, if the way the engines create checkpoints is satisfactory, the transformations do not need to set the checkpoints at all.

7.2 Undo Metamodel: the First Approximation

Having the basic notions explained, it is time to introduce the basic metamodel for storing the undo history (Figure 7.1).[1] In its basic variant, the *UndoHistory* is an ordered list of *Transaction*-s. The *currentTransaction* link specifies the transaction, to which modificating actions are being added. When neither undo, nor redo has been performed yet, the current transaction is the one that was created when the last checkpoint was set. During undo and redo the current transaction changes. In this first approximation we can assume that if some transactions have been undone, and a new transaction is being started, the undone transactions (the "tail") are deleted from the undo history.

Each transaction consists of *ModificatingAction*s. In the first approximation, I consider only the repository modificating actions (these actions correspond to RAAPI modificating actions) represented by the common superclass *RepositoryAction*[2]. Each modificating action can either create or delete some element in the repository. During undo, an inverse of a delete-action is a create-action, and vice versa. Since each action can be undone and redone, information for both creation and deletion needs to be stored. Thus, in Undo

[1] The initial variant of this metamodel has been offered by E. Rencis. Then, certain discussions on that metamodel and other undo-related topics were held. The participants of those discussions were me, E Rencis, S. Rikačovs, and K. Čerāns. Although I am not the only contributor to the initial variant of the undo mechanism, the extensions to the undo mechanism (see Sections 7.3–7.5) as well as its integration with TDA make up my personal contribution.

[2] Only the major modificating RAAPI actions have been reflected in Figure 7.1.

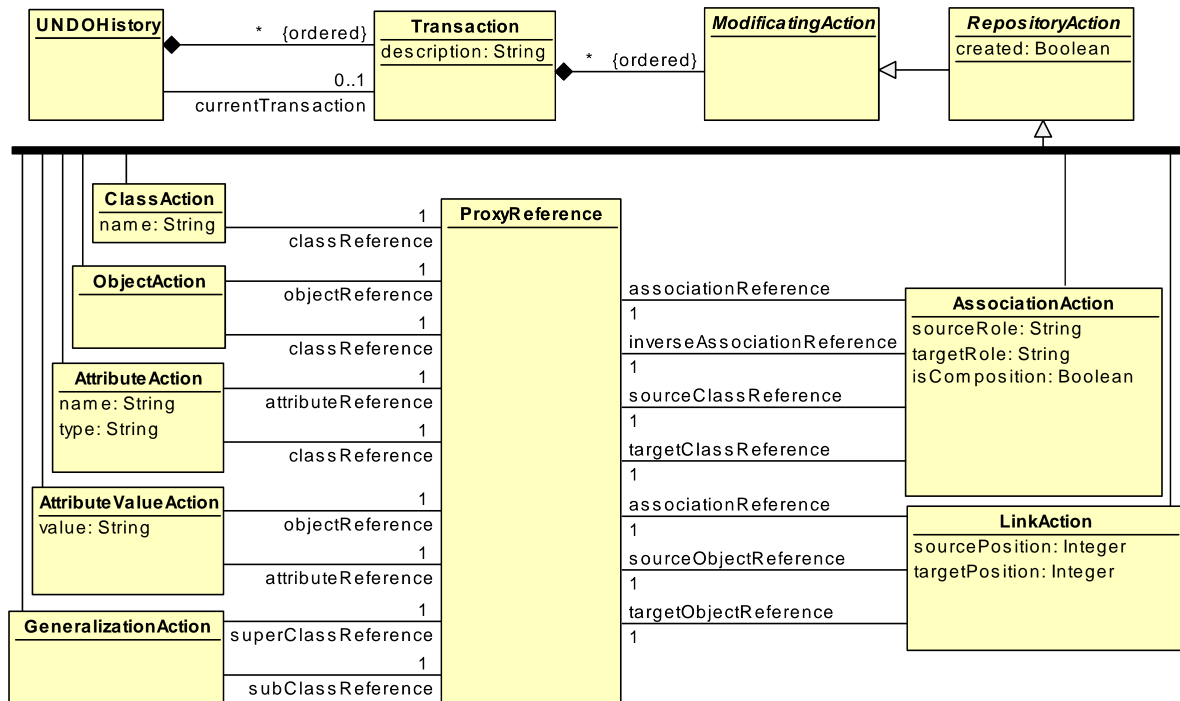


Figure 7.1: The basic variant of Undo Metamodel.

Metamodel, each repository action is grouped with its inverse in a single class, having a dual nature. The *created* attribute in the *RepositoryAction* class specifies whether the original action created or deleted something.

When some object is being deleted, TDA Kernel does not delete the corresponding *ProxyReference* instance. This ensures that after undo or redo the proxy reference will be the same, although, it can be later redirected to another (recreated) object.

When storing delete-actions in the undo history, an important precaution has to be taken. If some object is deleted, its attributes, links, and probably other objects (in case of composition) can also be deleted. TDA Kernel has to store all these cascade deletions in the undo history to be able to revoke all of them.

The *description* attribute of the *Transaction* class is used to describe the transaction. This description may be shown to the user as a hint before he is going to perform undo or redo.

In the next three sections, I introduce extensions to the basic undo metamodel.

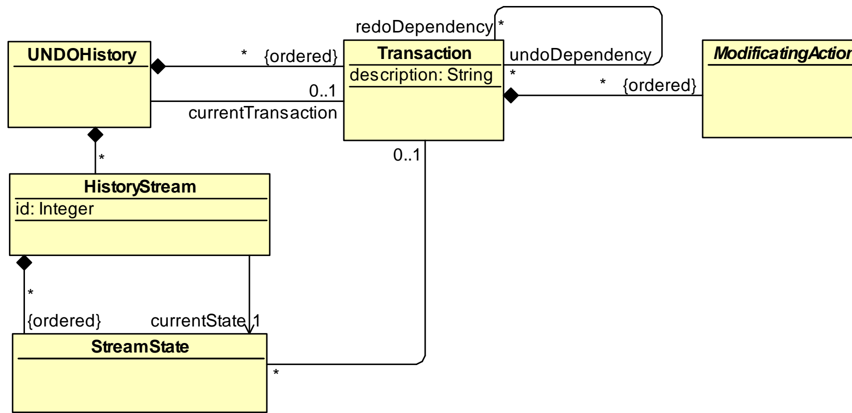


Figure 7.2: An extension to the basic undo metamodel introducing history streams.

7.3 Non-linear Undo: Multiple Undo History Streams and Dependencies Between Them

The need for non-linear undo may arise, for example, when working with several presentations (e.g., graph diagrams). On the one hand, it is reasonable to provide a separate undo history stream for each presentation. On the other hand, changes in one presentation may affect other presentations, and dependencies can occur. For instance, one diagram may contain an element that is referenced from another diagram. When the transaction that created the element is being undone, the transaction that created a reference to this element should also be undone. This section describes a generic solution for sharing the common undo history between several presentations that may depend on each other.

Let us look more narrowly at the TDA-based system. It may contain different presentations (e.g., diagrams), which may be created and destroyed at runtime. Each such presentation contributes some data to the system state. When the system state changes during some transaction, this may affect also some presentation states, but leave other presentation states unchanged. That is, the presentation state may remain the same while transactions being performed do not affect it, until some transaction eventually changes the presentation state transforming it to a new state. A sequence of different states for the given presentation is called an **undo history stream**. Each presentation state (stream state) except the initial one, is associated with exactly one transaction, which led to it.

Figure 7.2 depicts how the notion of history stream is added to the basic undo metamodel. The *UndoHistory* now may contain several *HistoryStreams*. Each history stream consists of *StreamState*-s. Each stream state in the history stream (except the initial one) is associated with a *Transaction*, which led to that stream state. Stream

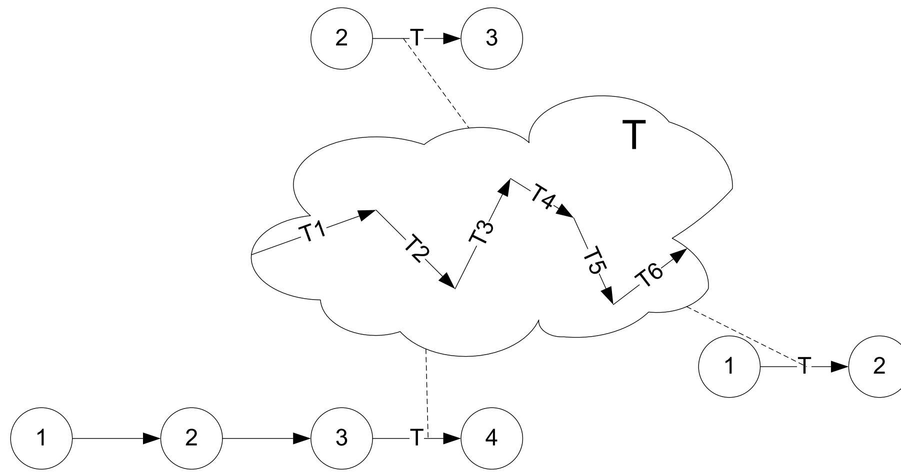


Figure 7.3: Transaction T (consisting of modificating actions T1-T6) is associated with several history streams (chains of circles). Circles correspond to stream states, and the arrows between them show the order. Transaction T led the topmost stream to State 3, the leftmost stream to State 4 and the rightmost stream to State 2.

states are ordered according to the sequence of their appearance. One of the states is the current (*currentState*).

Undo now is invoked not at the level of the whole undo history, but at the level of one history stream (a "global" history stream, which trails all the transactions, can be introduced, when needed). Positions in the undo history for each history stream are now determined by the corresponding *currentState* links, and the ordering is now specified by the composition between the *HistoryStream* and *StreamState* classes. Transactions are simply appended to *UndoHistory*, and the *currentTransaction* link now simply points to the transaction, to which modificating actions are added. If the transaction changes some stream state, a new stream state is associated with that transaction.

To handle dependencies, a distinction has to be made between implicit and explicit dependencies.

Implicit dependencies are dependencies implied from the association between transactions and stream states. For instance, all three undo streams from Figure 7.3 depend on each other, since undo in one history stream will cause undoing transaction T, and, thus, the other two history streams will change their states.

Explicit dependencies are dependencies between different transactions. For that reason the *undoDependency/redoDependency* association is introduced. The semantics is as follows: when some transaction T is being undone, then all undo-dependent transactions (role *undoDependency*) are also undone (if not undone earlier). Similarly, when T is

being redone, the corresponding redo-dependent transactions (role *redoDependency*) are also redone.

Explicit dependencies are always bi-directional: in one direction they are undo-dependencies, while in another direction they are redo-dependencies. Explicit dependencies can be added manually or automatically. Automatically added dependencies allow the developer not to think much (or at all) about specifying dependencies for correct undo behaviour. In this section I describe automatic explicit dependencies reasoned from repository actions. The next section will introduce automatic explicit dependencies reasoned from external states.

When TDA Kernel is processing repository modificating actions, it can reason about certain relations between transactions, and add corresponding explicit dependencies. For example, when a link between two objects is added in the current transaction C, TDA Kernel can find transactions A and B, where those two objects were created. Then, evidently, there are two dependencies: between C and A, and between C and B. If, for example, A is being undone, then C also has to be undone, since the link cannot exist without its end object. Similarly, when C is redone, A also has to be redone: A has to re-create the end object for the link, which will be re-created in C.

Table 7.1 lists, which dependencies can be reasoned automatically. Only redo-dependencies, which refer to earlier transactions that have to be redone when the current transactions is redone are shown (the corresponding undo-dependencies will be added automatically, since dependency links are bi-directional). If there are several transactions with the given criteria, the redo-dependent is the most recent transaction.

When TDA Kernel is processing a repository modificating action, it simply finds the corresponding redo-dependent transactions (if any), and creates the corresponding dependencies between those transactions and the current transaction. To be able to find redo-dependent transactions easily, TDA Kernel maintains a map that maps descriptions of the actions (as in Table 7.1) to the corresponding transactions.

7.4 External Actions and States

This section extends the undo mechanism with the ability to register external actions and states in the undo history. The need for such actions and states may occur, when an engine makes changes outside the repository, for instance, in some external database or in a graphical window with visualization that needs to be updated after undo or redo.

Repository action	**Redo-dependent transactions** (if can be found in the history)
ClassAction (created)	When the class with the same name was deleted.
ClassAction (deleted)	When this class was created.
ObjectAction (created)	When the class with the given *classReference* was created.
ObjectAction (deleted)	When this object was created.
AttributeAction (created)	When the class with the given *classReference* has been created. When the attribute of the given class with the same name was deleted.
AttributeAction (deleted)	When this attribute was created.
AttributeValueAction (created)	When the attribute was created. When the object was created.
AttributeValueAction (deleted)	When this attribute value was set/created.
AssociationAction (created)	When the class corresponding to the *sourceClassReference* was created. When the class corresponding to the *targetClassReference* was created. When the association from the class corresponding to the *sourceClassReference* with the same *sourceRole* was deleted.
AssociationAction (deleted)	When this association was created.
LinkAction (created)	When the association with the given *associationReference* was created. When the object corresponding to the *sourceObjectReference* was created. When the object corresponding to the *targetObjectReference* was created. When the link corresponding to the *associationReference* was deleted between objects corresponding to *sourceObjectReference* and *targetObjectReference*.
LinkAction (deleted)	When this link was created.
GeneralizationAction (created)	When the class corresponding to the *superClassReference* was created. When the class corresponding to the *subClassReference* was created. When the same generalization was deleted.
GeneralizationAction (deleted)	When this generalization was created.

Table 7.1: Dependencies: how they can be reasoned automatically.

When some engine creates a new presentation (e.g., a diagram) on the screen, it can also create an undo history stream for that presentation. When the presentation changes, the engine has to inform TDA Kernel about that change. There are two options:

- Inform that a state has been changed. This option is suitable, when it is more convenient to store the whole previous state instead of the delta between the previous and the next state. For instance, for a graph diagram, it is more convenient to store the coordinates of all elements at once, since the whole diagram needs to be repainted during undo.
- Inform that some undoable action has been performed. This is suitable, for example, if the memory size to store the action is much smaller than the memory size required to store the whole previous state.

In order to describe external states (the first option), engines have to create subclasses of the *StreamState* class. For each such subclass, the following commands have to be also defined (let *StateName* denote the name of the subclass):

- *RevokeStateNameCommand* – this command is issued by TDA Kernel, when the given state has to be removed from the presentation.
- *RevertStateNameCommand* – this command is issued by TDA Kernel, when the presentation has to renew its state. Commands *RevokeStateNameCommand* and *RevertStateNameCommand* are issued in a pair: the first one for the old state and the second one for the new state.
- *DestroyStateNameCommand* – this command is issued, when the memory occupied by the state has to be freed (i.e., when the state is being removed from the undo history, or the user closes the system).

For external actions (the second option), engines have to create subclasses of the *ModificatingAction* class (let *ActionName* denote the name of this subclass). Similarly, commands *UndoActionNameCommand*, *RedoActionNameCommand*, and *DestroyActionNameCommand* have to be defined for each subclass. However, unlike *RevokeStateNameCommand* and *RevertStateNameCommand*, which are called in a pair when the state changes, for actions only one command (either *UndoActionNameCommand,* or *RedoActionNameCommand*) is issued depending on whether the action is being undone or redone.

Reasoning Automatic Dependencies from External States. Assume a diagram, which initially was in State 1, has been brought by transaction T1 to State 2, and then, by transaction T2, to State 3. Thus, State 1 contains changes neither from T1, nor from T2, State 2 contains changes from T1, but not from T2, while State 3 contains changes from both T1 and T2. Then it is impossible to undo T1 independently of T2 since there is no state with changes from T2, but without changes from T1 in the undo history. This observation allows TDA Kernel to create another kind of automatic explicit dependencies: when TDA Kernel is notified about changes from State 1 to State 2 and from State 2 to State 3, it automatically creates a dependency between the transactions T1 and T2.

7.5 Adding Support for Multiple Redo Branches

An important property of the undo mechanism is the ability to revert to any recent state. However, in classical undo implementation, where the history stream is an ordered list, a modification after undo causes the "tail" of the history to be cleared, thus, the user may loose the ability to revert to certain states. Instead of clearing the "tail", a new history branch can be created for new modifications. If the user reverts to the same state several times, and starts new modifications, then several branches may arise.

One of these branches is marked as current. The modifications from this branch will be applied, if the user clicks the redo button. I call the state, to which the current branch leads, the **next state**. The branches before the current branch lead to the **early next states**, and the branches after the current branch lead to the **late next states**. According to these notions, the composition between *HistoryStream* and *StreamState* in the undo metamodel is replaced by three forward unidirectional associations *earlyNext*, *next* and *lateNext*. The backward unidirectional association *previous* is added as well (Figure 7.4).

In Figure 7.5, if the history stream is in State 2 (state numbers correspond to time moments when the state has been initially reached before undo or redo), and if the current next state is State 8, then the early next states are State 3 and State 5, while State 10 is the late next state.

One may think that in order to support branching, we have to add additional buttons to the standard two "Undo" and "Redo" buttons. Yes, we can do that. However, we can still continue using only two buttons, but they will change their behaviour slightly. I call them "Smart Undo" and "Smart Redo" buttons.

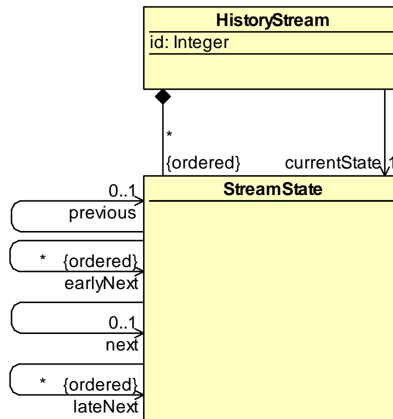


Figure 7.4: Replacing the composition between *HistoryStream* and *StreamState* with unidirectional associations to support redo branching.

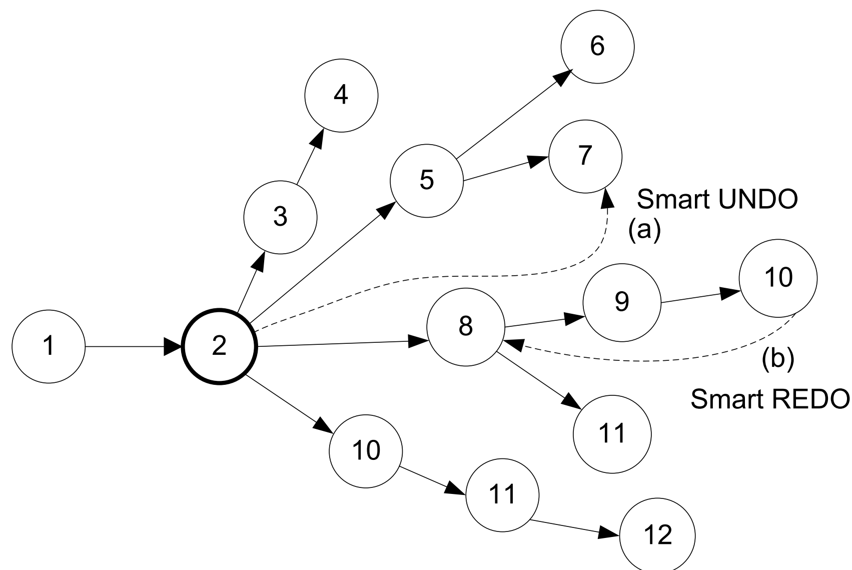


Figure 7.5: Redo branching (transactions are not shown) and the behaviour of the "Smart Undo" and "Smart Redo" buttons.

If there are early redo branches, the "Smart Undo" button goes to the most recent state among the states in these early branches (see Figure 7.5(a)). If there are no early redo branches, the "Smart Undo" goes back like the ordinary "Undo" button.

If there is the current branch, the "Smart Redo" button simply goes forward like the ordinary "Redo" button. If there are no branches, the "Smart Redo" button goes to the state before the next nearest branch (see Figure 7.5(b)). This behaviour of the "Smart Undo" and the "Smart Redo" buttons ensures that the user can traverse all the states. Besides, the same state may be visited several times depending on the number of branches outgoing from that state.

7.6 Related Work

Microsoft DSL Tools [58] use the domain model framework, which supports undo and redo. Although undo/redo logs the changes of the model (similar to repository modificating actions in TDA), it supports also propagation of changes. However, when describing rules for propagation of changes, the developer needs to distinguish between repository changes and external changes: repository changes must not be propagated during undo/redo, while external changes still have to be propagated. In TDA, external changes are added to the undo history as external actions. TDA Kernel will issue the corresponding command (*UndoActionNameCommand* or *RedoActionNameCommand)* to undo (redo) external changes, while no such command is required for repository changes.

A good example of a though-out solution to the undo problem is Eclipse [202]], on which a graphical tool building platform GMF is based [172, 173]. The undo history contains operations, and a context can be associated with each operation. Similarly, in TDA, history streams (≈contexts) are associated with transactions (≈operations). In Eclipse, an operation approver may be assigned to an operation. This approver is consulted before undoing/redoing that operation. The approver, for example, can forbid undoing an earlier operation, if later operations have not been undone yet. In TDA, undo within the same history stream is linear, while, at the level of the global undo history, earlier transactions can be undone without undoing later transactions. One may choose between creating the single history stream for totally linear undo and creating an individual history stream for each transaction. To force several transactions to be undone/redone at once, dependencies between transactions can be created.

An interesting implementation of undo in Collaborative Modeling Tool is described by D. English [203]. Two special commands are introduced there: *Save* and *Cancel*. The *Cancel* command returns the system to the state, when the last *Save* was performed. A remarkable point is that *Cancel* itself is added to the undo stack, and, as a consequence, the *Cancel* action by itself can be undone. This allows “forks” of user actions, starting from the last saved state, to be created. This resembles redo branching in TDA. The process of undoing the *Cancel* command resembles the behaviour of the “Smart Undo” button.

S. Rikačovs presented a solution for implementing the undo functionality in a transformation language [204]. Although worth by its own, this solution is not applicable to TDA, since undo/redo must be available not just to transformations, but also to engines.

In the AToM3 tool [182], undo is implemented by simply saving the model. This seems to be a good solution specifically for that tool, since undo support was not there originally.

An interesting application of undo for debugging purposes is described in the paper of Hartmann and Sadilek [205]. Assume some model (e.g., Petri net) is being executed and at some step an incorrect behaviour is found. When the execution semantics has been corrected, undo can be used to revert the model to the last correct state, and the execution may be re-started from that state instead of performing all the execution from the beginning.

The concept of worlds is a way of controlling the scope of side-effects. In this approach, states of the system form a tree, where each node (except the initial one) handles access to its parent's data and stores read and modified values [200]. Undo can be performed by switching from a child world to its parent world. Since worlds form a tree, a redo branching is also supported in this concept.

An noteworthy application of constraint evaluation in order to determine undo dependencies has been proposed by Groher and Egyed [206]. This approach allows inferring the dependencies at the time of undo depending on the model elements the user wants to revert to previous states. In TDA, dependencies have to be set before undo, although they can be calculated based on some constraints.

Chapter 8

Environment Engine

TDA can potentially be used in different environments such as:

- integrated development environments (IDEs), e.g., Eclipse or Visual Studio;
- large legacy or new applications requiring certain model-driven functionality to be integrated within them;
- tablet devices (with a multi-touch screen). In this case some TDA engines may need to be adapted to accept gestures from such multi-touch devices;
- a web-based application, where the main window is inside the web browser. This also may require certain modifications in engines, since engines will need to send their presentation to the user's device via the network.

To deal with multiple environments, I propose the following solution. One of the interface metamodels in TDA is designated to abstract environment-specific aspects. This metamodel is called Environment Metamodel. To launch TDA in a particular environment, an engine implementing Environment Metamodel within that environment is required. This engine is called Environment Engine. When I speak about Environment Engine, I actually mean the whole class of such engines: one engine for each environment. When deploying a system, an appropriate engine is selected. Thus, Environment Engine is replaceable. Other engines have to deal with a particular Environment Engine through Environment Metamodel.

Environment Engine is a module that launches TDA Kernel. TDA Kernel requires at least one repository to operate, but it may use other data stores as well. Hereinafter the term **project** will denote a logically united set of data stores used by TDA Kernel. In

its minimal variant a project consists of a single model repository. A more complex case could involve multiple repositories as well as links to external files (images, documents, etc.) and, perhaps, links to some shared databases. A project is TDA "memory". It can be compared to a document in a text processor. TDA Kernel works with only one project at a time, but it is possible to initialize multiple TDA Kernels for dealing with multiple projects at a time.

When launching TDA Kernel, Environment Engine calls the *open* function of TDA Kernel to open the project (TDA Kernel implements the *IRepositoryManagement* API containing the *open* function, see Appendix A.3). When the project is being closed, Environment Engine calls the *close* function.

Besides launching TDA Kernel, Environment Engine is responsible for the following:

- It stores system-specific paths (e.g., a path to the "bin" directory) in the repository in order TDA Kernel, other engines, and transformations could rely on these paths, when needed.

- It informs transformations and engines when the user creates a new project, opens an existing one, or closes a project, by emitting corresponding events. It is up to Environment Engine how to implement the user interface for these user actions. One option is to provide menu items (e.g, File→Create, File→Open, File→Close) to the user. Buttons or a toolbar can be used as well. If TDA is used in a task that does not require user intervention, Environment Engine may emit project-related events automatically.

- It saves the project, when necessary (e.g., when the user chooses the "save" menu item). To save the project, Environment Engine has two options: either to issue a *SaveCommand* defined in TDA Kernel Metamodel, or to invoke the *startSave/finishSave* functions of TDA Kernel. Transformations and engines do not use the *IRepositoryManagement* API. They can assume that the project has already been open. If they need to save the project programmatically, they can issue a *SaveCommand* to TDA Kernel.

- It provides access to the main window, to which other engines can attach their graphical presentations.

- It provides a means to include/exclude certain choice options (e.g., copy/paste, undo/redo, and others) at runtime. Usually, such options are available via the main

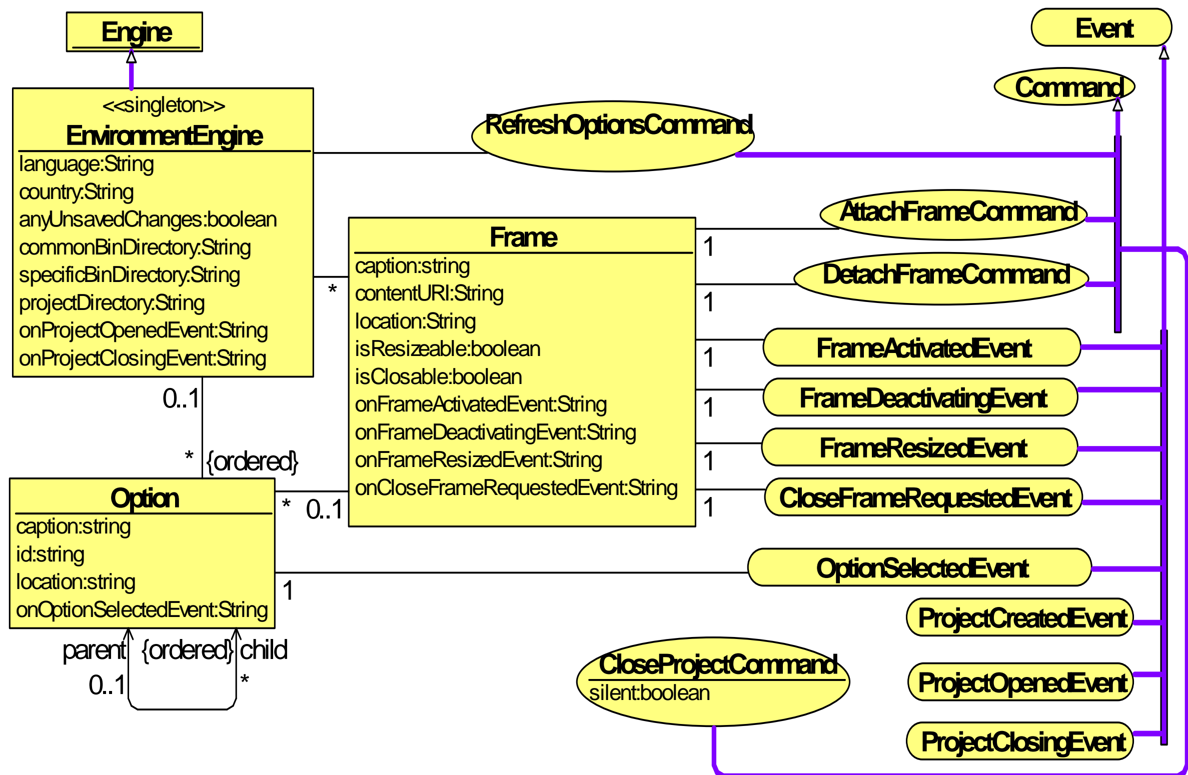


Figure 8.1: Environment Metamodel.

menu bar, or from the toolbar. Options may be global (e.g., an option to validate the entire project) or specific to a particular presentation (e.g., undo/redo may be different for different presentations).

These functions of Environment Engine are reflected in Environment Metamodel (also called Environment Engine Metamodel) depicted in Figure 8.1.

The central singleton class is *EnvironmentEngine*. This is a TDA convention: each engine should use a singleton class named as the engine. The instance of this class is usually used as a starting point to access certain engine-specific information encoded by means of attributes or links to other objects.

The main points of Environment Metamodel are explained below.

Locale-specific information. The *language* and *country* attributes of the *EnvironmentEngine* class encode the locale according to the ISO-639 and ISO-3166 standards (e.g., *language*="en" and *country*="GB"). These attributes can be used by other engines to adapt their behaviour according to country-specific settings and to translate their user interface.

Asking for a confirmation. The *anyUnsavedChanges* attribute is a common place where engines and transformation can inform Environment Engine that they have unsaved changes. When finalizing TDA Kernel, Environment Engine can ask for user confirmation depending on the value of this attribute. When Environment Engine saves the project (e.g., by issuing a *SaveCommand* to TDA Kernel, when the user chooses File→Save), it resets the value of *anyUnsavedChanges* to `false`.

Storing system-specific paths. The following three attributes of the *EnvironmentEngine* class encode system-specific paths:

- *commonBinDirectory* - the directory, where TDA binaries reside;
- *specificBinDirectory* - the directory, where TDA-based systems can put their specific binaries (e.g., tool-specific transformations)[1];
- *projectDirectory* - the directory, where the current project resides.

These attributes are re-written each time a project is being created/opened.

Informing TDA Kernel when the project is being created, opened, saved or closed. Events *ProjectCreatedEvent*, *ProjectOpenedEvent*, and *ProjectClosingEvent* are used to inform TDA Kernel and other interested parties (transformations and engines) when the project has been created, opened, or is being closed, respectively. On *ProjectClosingEvent*, engines, for example, may perform some clean-up.

Notice that the on-attribute *onProjectCreatedEvent* does not exist. The reason for that is simple: when a project is being created, no transformation or engine has been called yet. Thus, there was no possibility to set the desired value for this attribute. Thus, when Environment Engine emits *ProjectCreatedEvent*, TDA calls a transformation with the predefined name “main”[2]. This transformation brings TDA to life: it can attach other engines, set values for *on*-attributes, call other transformations, and so on.

When a transformation or an engine needs to close the project, it may issue a *CloseProjectCommand*. If the value of the *silent* attribute is true, then Environment Engine must not ask for the user confirmation. Otherwise, it is up to a particular Environment Engine. After *CloseProjectCommand* (and the user confirmation, if it was

[1] If Environment Engine is specially designed to work only with one particular tool, *commonBinDirectory* and *specificBinDirectory* may point to the same location.

[2] Technically, this name is sequentially passed to all adapters for transformations until some adapter eventually finds the transformation and launches it.

requested) Environment Engine emits a *ProjectClosingEvent*. Processing of this event is the last possibility to access the repository (repositories) before closing it (them).

Attaching graphical presentations to the main window. Engines can create graphical presentations. The term **frame** denotes a surface, where a presentation is visualized. Usually, a frame is a container, where graphical user interface (GUI) components are placed. Unlike dialog boxes, a frame may be without the border and it may be attached to other windows, including the main window provided by Environment Engine. Each frame is represented as an instance of the *Frame* class in the model repository. It is up to an engine whether to attach a presentation to the main window of Environment Engine, or to simply store in the repository a *Frame* instance, which then can be used by a model transformation to attach the presentation somewhere else (e.g., to a Dialog Engine window).

Frames may be identified differently, for example:

- by means of operating system window handles (e.g., in Windows OS there is a special type called HWND for such handles);
- as components in some object-oriented programming languages (e.g., subclasses of java.awt.Component in Java);
- as HTML documents (for frames to be displayed in web browsers).

To distinguish between different ways of identifying frames, the URI syntax is used [195]. The URI of the frame is a string starting with the name of the way of identifying frames (the "protocol"), followed by a colon, after which the frame is described in a "protocol"-specific manner. For instance, the URI hwnd:123456789 may be used to identify the frame by means of an operating system window handle, while the URI like html:http://www.example.org/html_with_javascript.html may be used to identify frames described as HTML documents[3].

When an engine needs to attach a frame to the main window, it creates an instance of the class *Frame* and specifies the URI as a value for the attribute *contentURI*. Also, a caption (attribute *caption*) for the frame may be specified as well as the flags whether this frame is resizeable (*isResizeable*) and closable (*isClosable*). Then, an *AttachFrameCommand* is created, linked to the frame, and executed (via the TDA submitter object). The

[3] C++ objects may be identified by the memory address. Addressing Java objects is not so trivial, but still that can be performed by using Java Native Interface, JNI, and its global references [207]. Another solution is to introduce a global hash map for indexing Java objects passed to Environment Engine.

parameter *location* specifies where to put the frame within the main window. A special *location* value "MODAL" means that the frame will be displayed as a modal window. Other values mean that the frame is non-modal and that it has to be attached somewhere within the main window (possible values are "WEST", "EAST", "NORTH", "SOUTH", "CENTER"; particular Environment Engine implementations may accept other specific values as well). Environment Engine takes care of adding a border to the frame, when the border is needed.

For a modal window, Environment Engine only prepares the environment for the frame, but does not execute the modal cycle waiting for user actions within the frame. The engine that created the *AttachFrameCommand* is responsible for that. After the modal cycle finishes, a *DetachFrameCommand* instance must be issued to Environment Engine to turn the environment back into the non-modal state.

DetachFrameCommand may be created also for non-modal frames, when they are not have to be displayed any more.

If the frame is closable, Environment Engine provides the close button (e.g., the "X" button). However, Environment Engine does not closes the frame by its own. Environment Engine creates a *CloseFrameRequestedEvent* instead. The engine that owns the frame should catch this event and decide whether to close and detach the frame, or not. Such a behaviour allows the engine, for example, to ask for user confirmation, or not to close the window at all, if there are reasons for that.

The *FrameActivated* and *BeforeFrameDeactivated* events may be used, for example, to change options (e.g., main menu items, see below) depending on the active window.

The *FrameResized* event is useful, when the content of a frame needs to be repainted on resize.

Including/excluding choice options. There may be different choice options, which may be used to perform certain actions with presentations or to access services provided by engines. For example,

- cut/copy/paste;
- undo/redo;
- "Export as..." (for exporting the given presentation into a different format);
- "Validate" (for validating the data in the given presentation);

- “Generate code” (for generating domain-specific code from a description in a given DSL).

These options may be introduced by engines or transformations.

Choice options are described by means of the *Option* class. The *location* attribute specifies where the option should be placed, e.g., the value “MENU” can be used to denote the main menu, while the value “TOOLBAR” denotes the main toolbar.

Environment Engine is allowed to change the list of possible locations it supports.

If Environment Engine does not support some value of the *location* attribute, it should replace it by another reasonable value (this may be actual for touch screen devices, where menus and toolbars can be replaced by other location types).

When the user chooses an option, Environment Engine emits an *OptionSelectedEvent*.

The following two chapters present Dialog Engine and Error Engine. Like other graphical presentation engines, they depend on Environment Engine, since they need to attach their windows to the main window.

Chapter 9

Dialog Engine

It is hard to imagine a graphical DSL tool without graphical user interface (GUI). Thus, when thinking about the graphical presentation, we have to think not only about visualizing the main functionality, but also about dialog windows.

This chapter describes Dialog Engine. Like other TDA engines, Dialog Engine also has its interface metamodel, by means of which dialog windows can be specified. Given an instance of a dialog window, Dialog Engine is able to visualize it automatically at runtime. Exact coordinates do not need to be specified — Dialog Engine will calculate the coordinates by its own. Thus, transformations may generate dialog windows on-the-fly, and Dialog Engine will find the appropriate coordinates. However, transformations can specify certain constrains on coordinates, should such a necessity occur.

Besides visualizing dialog windows, Dialog Engine passes user events (such as button clicks) to model transformations. Model transformations, in their turn, issue commands (such as "Close" and "Refresh") to Dialog Engine.

In order to calculate coordinates, Dialog Engine reduces a dialog window specified by means of Dialog Engine Metamodel to an instance of the quadratic optimization problem, which is then passed to the quadratic optimization solver. When a solution to the quadratic optimization problem is found, Dialog Engine can easily translate it to the coordinates of GUI components, and to position these components accordingly. Coordinates need to be calculated in these cases:

- when a dialog window is being displayed for the first time;
- when a dialog window is being resized;

- when a transformation asks Dialog Engine to refresh a dialog window or a significant portion of it (e.g., to update certain data in the dialog window, or to show/hide certain GUI components).

The next section (Section 9.1) presents Dialog Engine Metamodel. Section 9.2 explains how a dialog instance can be laid out by means of the quadratic optimization technique.

9.1 Dialog Engine Metamodel

The core of Dialog Engine Metamodel is depicted in Figure 9.1. The metamodel is able to describe simple dialog components such as the button, the check box, and the list box. There are two additions to the core metamodel called Tree Metamodel (Figure 9.9) and Table Metamodel (Figure 9.10) to express non-trivial components (the tree and the table), which may also be put within dialog windows. The components found in these metamodels are supported by a wide range of GUI toolkits, thus, there should be no problem to use any of those toolkits for handling instances of Dialog Engine Metamodel.

The following subsection describes the core of Dialog Engine Metamodel. The next two subsections briefly describe Tree Metamodel and Table Metamodel.

The Core of Dialog Engine Metamodel

The tree of components

The main notion in Dialog Engine Metamodel is the notion of the component (see class *Component* in Figure 9.1). There are two types of components:

- terminal components (like the button, the combo box, and the list box), which, from the metamodel point of view, do not contain other components (for a detailed description of these components, refer to my paper on Dialog Engine Metamodel [6]);

- containers (see class *Container* in Figure 9.1), which are components that may contain other terminal components and containers.

The generalization and the composition between the *Component* and *Container* classes define a tree structure, which is used to specify dialog windows. A dialog window is such a tree, rooted at an instance of the class *Form*.

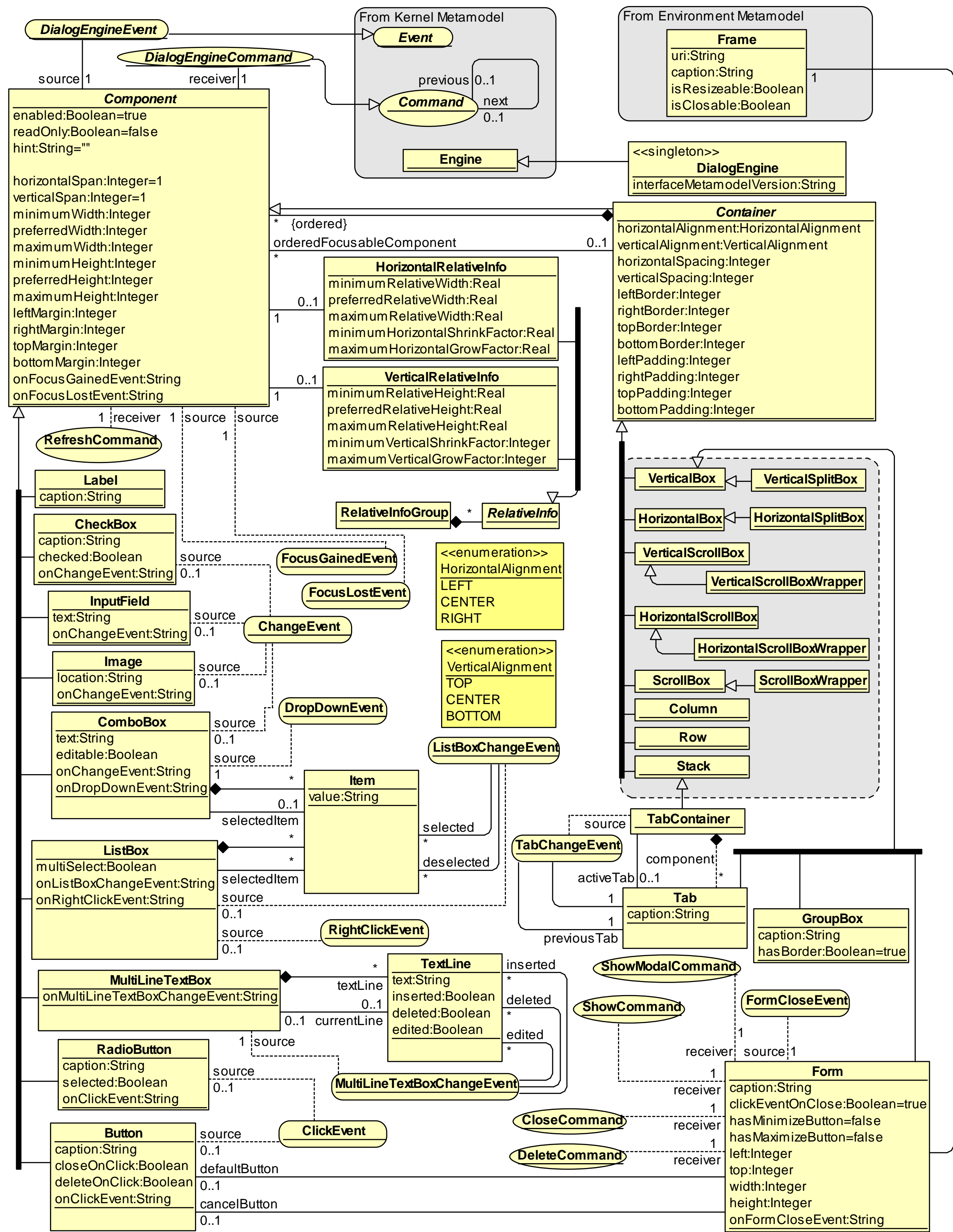


Figure 9.1: Dialog Engine Metamodel.

Interactivity

Dialog windows are interactive — the user may click on the components, and certain actions may be expected to process these user events. Besides, transformations can issue certain commands related to a dialog window, e.g., a command to refresh a part of that window. For that reason, Dialog Engine Metamodel contains certain events and commands assigned to certain components. For example, a click event (a *ClickEvent* instance) may be linked to a *Button* or to a *RadioButton*, and a *RefreshCommand* may be issued to update data in the given component, or to update all the components within the given container. In Figure 9.1, events (descendants of the *Event* class) are depicted as rounded rectangles, and commands (descendants of the *Command* class) are depicted as ellipses.

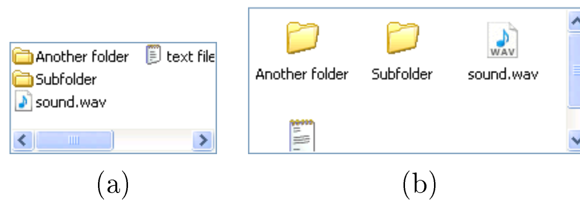
Laying out components

One of the important features of Dialog Engine Metamodel is the possibility to specify the layout of dialog elements. If we sketch a dialog box on a sheet of paper, we usually don't worry about exact coordinates, but we think about the mutual layout of components for grouping them and for aesthetics. Just the same kind of layout information is expected in instances of the proposed metamodel.

When imaging a dialog box, I assume that all begins with the form, which is the top-level (root) container. This container can be logically divided into several parts, or cells. Each cell may be divided again, and so on, recursively. Some cells are occupied by visible components or containers, while other are simple invisible "borders" used as intermediate cells for further division. Because of this recursive division, dialog windows are represented as a trees in Dialog Engine Metamodel.

There are several ways how the given cell can be divided. They are represented by 13 classes in the rounded dashed rectangle on the right of Figure 9.1.

The two natural ways of laying out the components is laying them horizontally and vertically. Thus, *HorizontalBox* and *VerticalBox* appear. Also, one or two scrollbars may be presented. To handle the single scrollbar case, I added *HorizontalScrollBox* (the children are put first vertically and then horizontally; see Figure 9.2 (a)) and *VerticalScrollBox* (the children are put first horizontally and, when there is not enough space, the next "row" is added; see Figure 9.2 (b)).

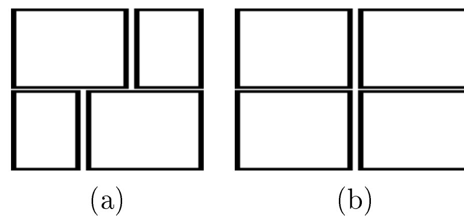


(a) (b)

Figure 9.2: Examples of (a) a horizontal scroll box and (b) a vertical scroll box.

To handle the case with both scrollbars, I introduced only one container, *ScrollBox*, where the children are put vertically. If the components need to be laid out horizontally, a horizontal box can be put inside a scroll box.

The container types mentioned so far do not allow creating structures like in Figure 9.4 (a). Besides, they cannot be used to create grid-like structures. So, two more container types appear: the row (class *Row*) and the column (class *Column*). In fact, a way for specifying grids could be borrowed from HTML, where tables (tag `<TABLE>`) consist of rows (tag `<TR>`), and rows consist of cells (tag `<TD>`). However, in order not to overwhelm the metamodel, I do not introduce the classes `Table` and `Cell`. Instead, rows and columns are allowed to lay in any container (which will play the role of the `<TABLE>` tag), and the components put within a row or a column can themselves be considered cells. If there are several rows (columns) inside the same container, the components inside these rows (columns) are aligned to form the grid structure (see Figure 9.3).



(a) (b)

Figure 9.3: Examples of (a) horizontal boxes with components that are not aligned to grid, and (b) rows where the children have been aligned to grid.

In order to be able to create structures like in Figure 9.4 (a), the components have to be allowed to span several rows or columns. This may be done by specifying corresponding values to the attributes *horizontalSpan* and *verticalSpan*[1], see Figure 9.4 (b).

A special kind of container type is needed to implement the tabs (see Figure 9.5).

Since tabs occupy the same space, we may think that the components are put one over another like cards. So, I introduce the *Stack* container type where all the children occupy the same space.

[1]These attributes are available for all components, but have the meaning only for components that lie inside a row or a column, and form a grid-like structure.

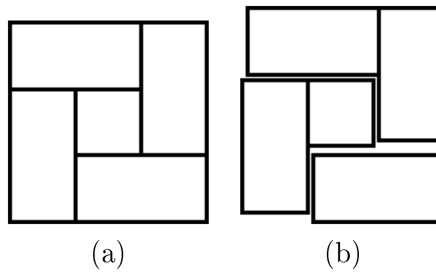


Figure 9.4: (a) An example of five containers that cannot be laid out using horizontal and vertical boxes only. (b) The arrangement of the same five containers using rows. The first row contains two components: the first one spans two columns (horizontally), and the second one spans two rows (vertically). The first component of the second row spans two rows; neither rows nor columns are spanned by the second component (or, we may say, one row and one column are spanned). The third row has only one component that spans two columns.

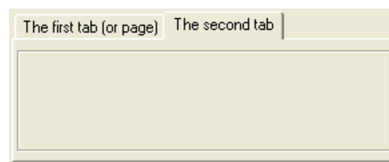
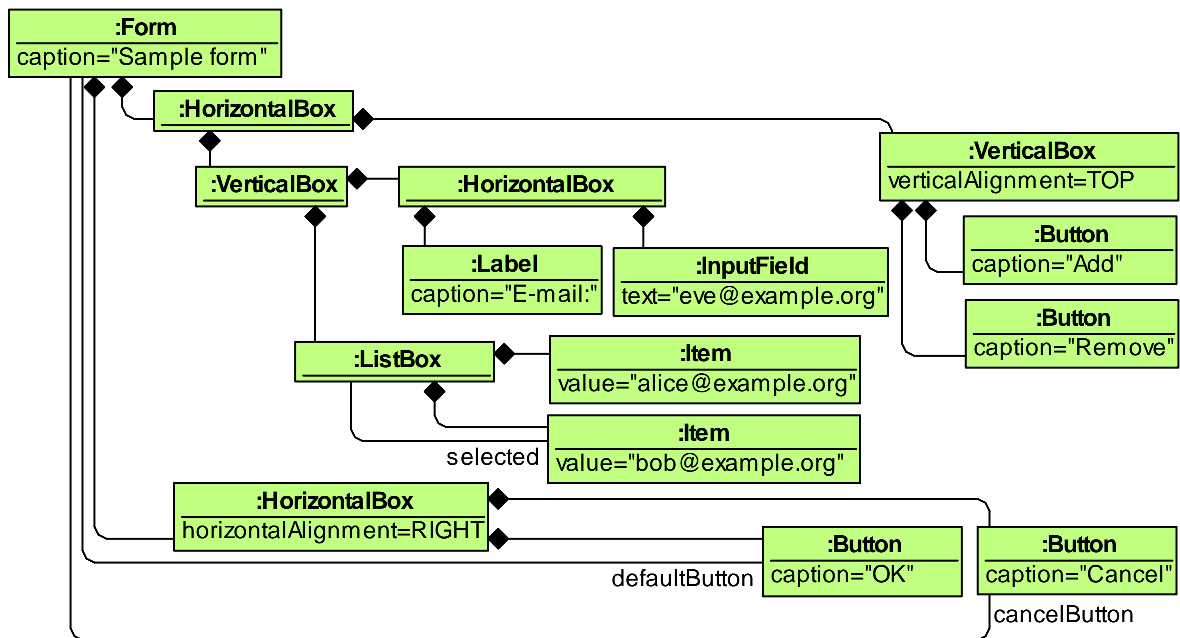


Figure 9.5: An example of tabs (pages).

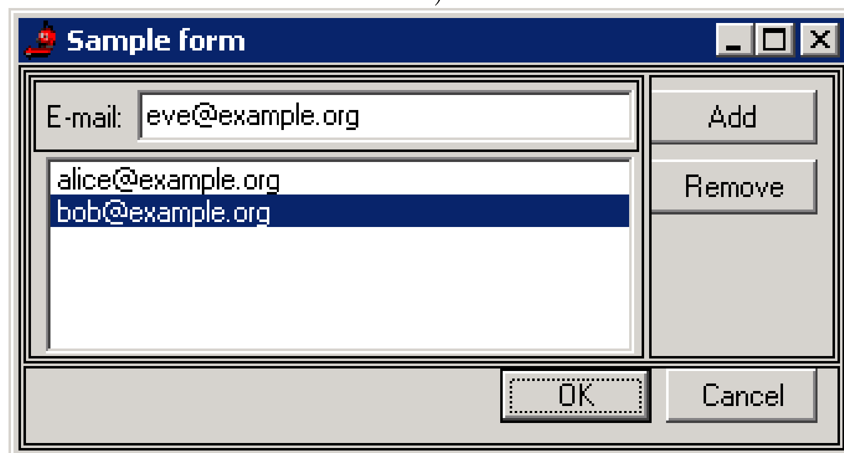
In a majority of cases there is no need for using other types of containers than the just mentioned. However, to support splitters, which allow the size of one component to be increased at the expense of decreasing the size of another component in the same container, I add also the *HorizontalSplitBox* and *VerticalSplitBox* classes. The splitters appear automatically between components lying in these split boxes. Also, I add three wrappers: *HorizonalScrollBoxWrapper*, *VerticalScrollBoxWrapper*, and *ScrollBoxWrapper*. They are intended to wrap a single large component inside a scroll box (with a horizontal, a vertical, or with both scrollbars).

Table 9.1 summarizes the types of the containers I described and tells which containers are visible and which are invisible. In case a visible analogue for an invisible container is required, a *GroupBox* can be used (it has a visible border with an optional caption). A group box is also used to group radio buttons (only one of the radio buttons can be selected within a group).

Figure 9.6 shows a sample Dialog Engine Metamodel instance (a component tree) and the corresponding form on the screen.



a)



b)

Figure 9.6: (a) An instance of Dialog Engine Metamodel for the sample form. (b) The sample form: the rectangles (in reality invisible, but shown here) outline horizontal and vertical boxes.

Invisible container types	**Visible container types**
VerticalBox	*VerticalSplitBox*
HorizontalBox	*HorizontalSpllitBox*
Column	*VerticalScrollBox*
Row	*VerticalScrollBoxWrapper*
Stack	*HorizontalScrollBox*
	HorizontalScrollBoxWrapper
	ScrollBox
	ScrollBoxWrapper

Table 9.1: Container types.

Specifying dimensions for components

There is a dilemma regarding the usage of exact sizes and/or coordinates for components. On the one hand, exact coordinates tell Dialog Engine the desired sizes of components in case the components with the default (or somehow calculated) sizes do not look well. On the other hand, the system font and depth-per-inch (DPI) settings may differ from one computer to another, thus, it may be preferred to avoid exact absolute sizes and coordinates. Dialog Engine Metamodel has features that may help dealing with this dilemma:

- The metamodel permits specifying absolute sizes, including minimal, preferred, and maximal sizes. But all these sizes are optional, and when they are not specified, Dialog Engine selects the values by itself. These values are specific to a particular platform and/or widget toolkit. They have to be chosen to form nice look and feel, and to allow resizeable components to be resized. When applicable, the DPI settings and the size of the font used have to be taken into an account.

- The metamodel allows specifying also relative sizes for components. Thus, one can specify, for example, that the given input field has to be twice wider than the given button. Or, that the aspect ratio of the dialog form should be 4:3.

Absolute dimensions. Absolute dimensions are set by the six attributes from *minimumWidth* to *maximumHeight* of the *Component* class. The *maximumWidth* and *maximumHeight* values are allowed to be increased to satisfy other constraints. Thus, *maximumWidth*= 0 means that the width of the component should be as small as possible (*maximumWidth* value will be no less than the *minimumWidth* value).

The *horizontalAlignment* and *verticalAlignment* attributes of the *Container* class refer to the child components. If a child is resizeable, then it is docked to the border of the

parent container. However, if the child reaches its maximum width (or height) when the parent container is being resized, the child will be aligned according to the values of the *horizontalAlignment* and *verticalAlignment* attributes. If there are several children in a horizontal box, the *horizontalAlignment* refers to all of them as one component. The same is true for the vertical box and the *verticalAlignment* attribute.

The meaning of attributes for specifying margins (in *Component* class) as well for specifying borders, padding and spacing (in the *Container* class) is revealed in Figure 9.1. The margins specify the extra space outside the component (i.e., this space is not considered to be a part of the component). The borders in the *Container* class specify the size of the border (e.g., bevel). The border is a part of the container. Paddings are like margins, but lay inside the area bounded by the border. In non-scrollable containers the notions of padding and border are interchangeable. In scrollable containers, the border is outside the scrollable area, while the padding is inside.

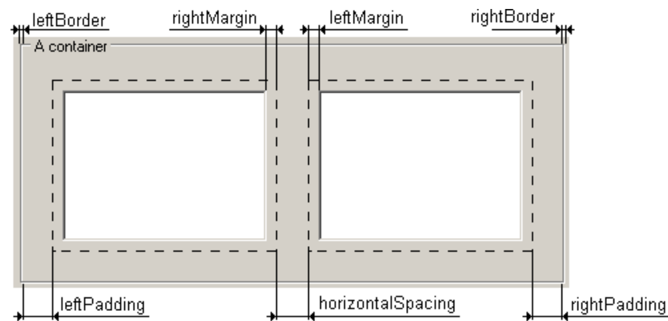


Figure 9.7: An example illustrating what do values for margins, borders, paddings and spacings mean.

Relative dimensions. The relative dimensions are related to the notion of the relative information group (see class *RelativeInfoGroup*). The group consists of widths and/or heights that depend on each other according to the given ratio of the lengths. To specify a ratio for widths and heights of the given components, the corresponding *HorizontalRelativeInfo* and *VerticalRelativeInfo* instances have to be linked to a *RelativeInfoGroup* instance. There is no need for a particular *HorizontalRelativeInfo* or *VerticalRelativeInfo* instance to be in several groups (otherwise, the groups depend on each other and may be replaced by one group by adjusting the ratio).

> *Example.* To specify the relative width ratio 2:3:4 between the three components, a *HorizontalRelativeInfo* instance has to be attached for each of these components and the corresponding values *preferredRelativeWidth* attribute

have to be set to 2, 3, and 4, respectively. Finally, these three *HorizontalRelativeInfo* instances have to be linked to the same *RelativeInfoGroup* instance to form a group. The relative heights may be specified in a similar way.

The minimum and maximum relative width and heights are useful for resizing. An example is given in Figure 9.8.

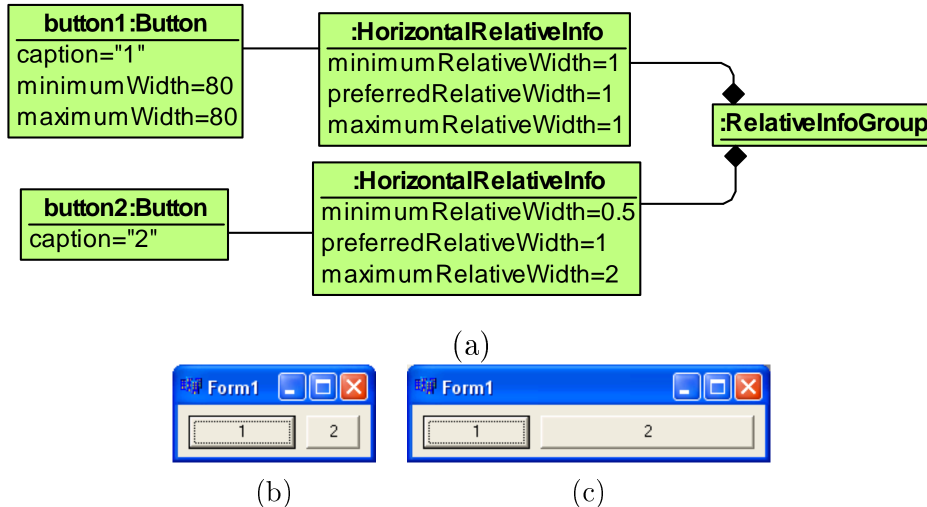


Figure 9.8: An instance (a) demonstrating the usage of minimum and maximum relative sizes. The minimum (b) and the maximum (c) sizes of button 2.

Button 1 is not resizeable, and the preferred width of Button 2 is the same as the width of Button 1. However, if the preferred ratio could not be met, Button 2 is allowed to be up to two times wider or shorter than Button 1.

Tree Metamodel

Figure 9.9 depicts Tree Metamodel, which describes "trees" (like a tree in a file system explorer). The *TreeNodeSelectEvent* is emitted when the user clicks on a tree item. This *previous* link denotes which tree node was selected last. A *TreeNodeExpandedEvent* is emitted when the user clicks on the expand sign ("+") of a collapsed tree node to see the children of that node. A *TreeNodeCollapsedEvent* occurs, when the user clicks on the collapse sign ("–") of an expanded node to hide the children of that node. A *TreeNodeMoveEvent* occurs when the user moves a tree node to a different position in the tree. This event may occur only when the value of the *movableNodes* attribute of the corresponding *Tree* instance is `true`.

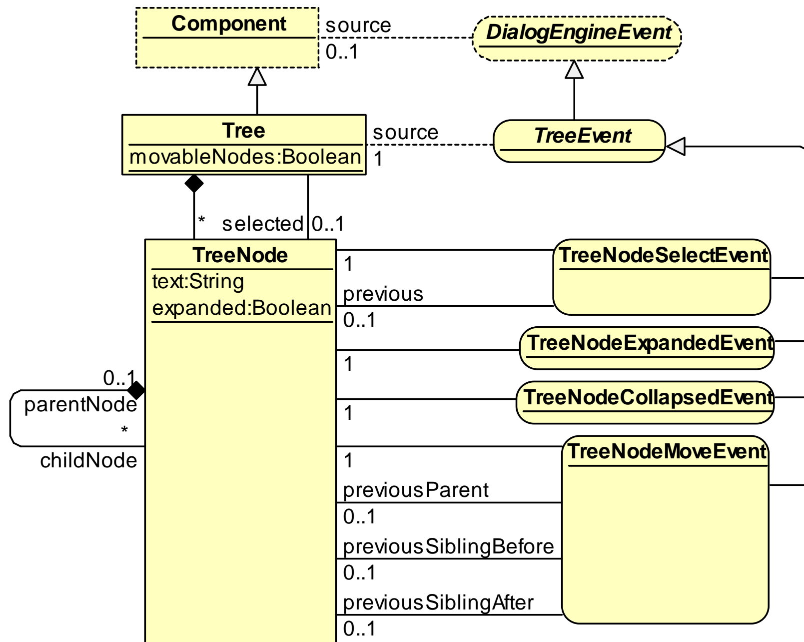


Figure 9.9: Tree Metamodel (an addition to Dialog Engine Metamodel)

Table Metamodel

Figure 9.10(a) depicts Table Metamodel used to express vertical tables, i.e., tables, where columns have labels, and data are shown in rows. Figure 9.10(b) depicts the semantics. Some comments on Table Metamodel:

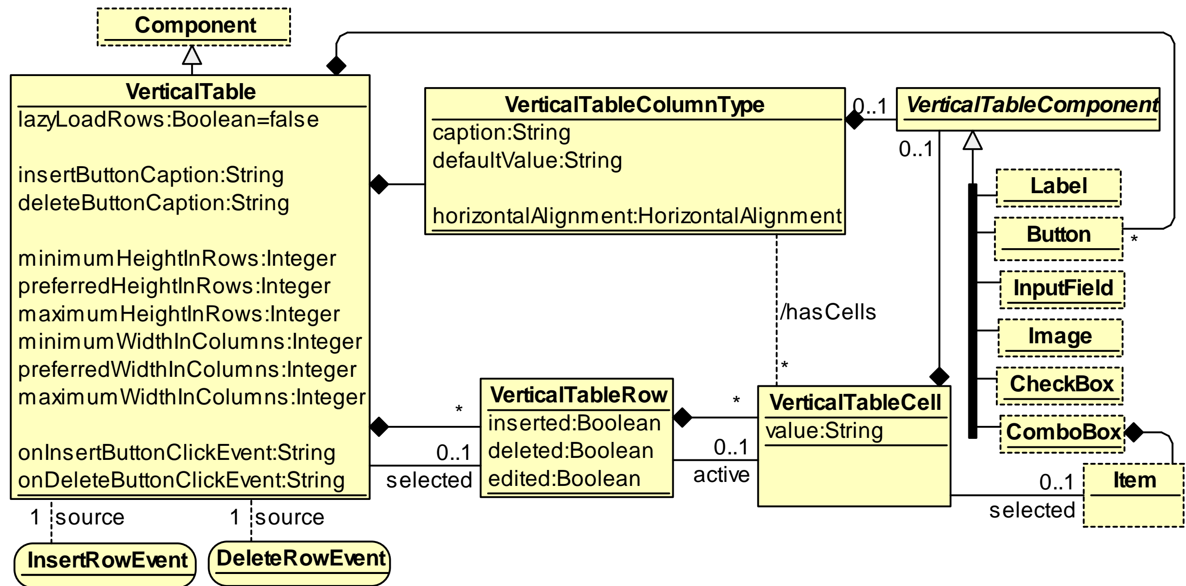
- The *lazyLoadRows* attribute means that the table rows should not be loaded all at once. Only visible rows have to be loaded first. Then, when the user scrolls the table, other rows may be loaded. Although this may speed up the table, not all cells are taken into a consideration when calculating preferred widths for the columns.

- The *insertButtonCaption* and *deleteButtonCaption* are useful only for non-read-only tables (when the *readOnly* value is `false`). These are captions for the buttons for inserting and deleting rows.

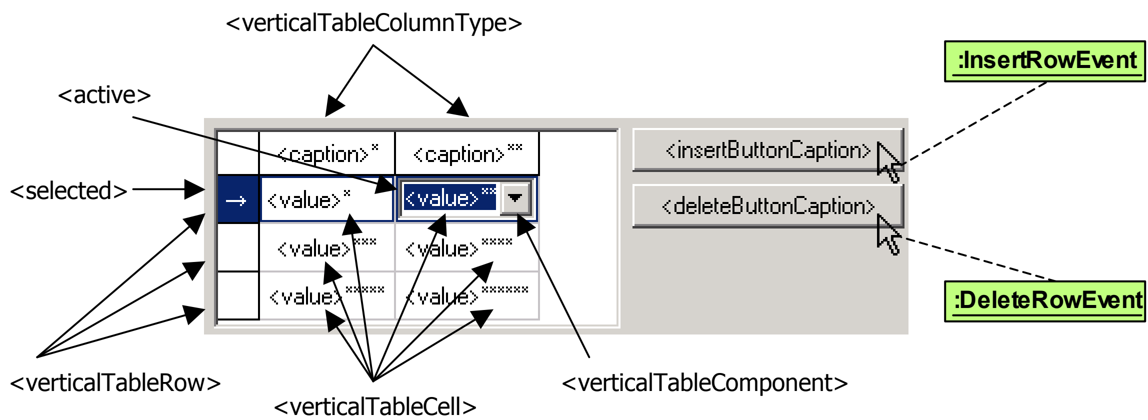
- The *hasCells* association is marked as derived since it can be calculated. The order of *VerticalTableCell*-s in a *VerticalTableRow* corresponds to the order of *VerticalTableColumnType*-s of the given *VerticalTable*.

- The *defaultValue* attribute values are used for the corresponding cells when the new row is added.

- The *VerticalTableComponent* can be linked either to a *VerticalTableColumnType* or to a *VerticalTableCell*. In case there is no particular component linked to a cell, it



(a)



(b)

Figure 9.10: a) Table Metamodel (an addition to Dialog Engine Metamodel) and b) its semantics.

is considered that the cell is occupied by the component linked to the corresponding *VerticalTableColumnType*. This component is also used as a default component for new rows.

- Since a component linked to the *VerticalTableColumnType* corresponds to several cells in the same column, the input value for each cell has to be taken from the *value* attribute of the *VerticalTableCell* class. For the *ComboBox*, there is also the *selected* association from the cell to the item. This association has to be used instead of the original *selectedItem* association of the *ComboBox* class.

9.2 Applying Quadratic Optimization

This section explains how, given a dialog instance specified according to Dialog Engine Metamodel, the quadratic optimization can be used to lay out GUI components in the corresponding dialog window.

The QMDC and the Extended QMDC Problems

The problem of quadratic minimization subject to difference constraints (QMDC) is as follows. Given n variables $x_0, x_2, \ldots, x_{n-1}$, minimize the quadratic function

$$\sum_{0 \leq i < n} a_i x_i^2 + \sum_{0 \leq i < j < n} b_{ij} x_i x_j + \sum_{0 \leq i < n} c_i x_i$$

subject to difference constraints

$$x_i - x_j \geq d_{ij}, \text{ where } 0 \leq i, j < n.$$

If also the constraints in the form

$$x_i - x_j \geq d_{ij} \geq m_{ij}, \tag{9.1}$$

are allowed, then we get the Extended QMDC problem (EQMDC). Here d_{ij} are the desired values and m_{ij} are the minimum values. In case the constraints taking into a consideration only d_{ij} are unsatisfiable, one or more of d_{ij} values may be decreased preserving $d_{ij} \geq m_{ij}$, i.e., d_{ij} cannot be decreased by more than by $d_{ij} - m_{ij}$.

There exists a method for solving QMDC in a quite moderate time [208]. The EQMDC problem can be reduced to the QMDC in the following way. First, a constraint graph $G = (V, E)$ is created [209, Section "Difference constraints and shortest paths"]. Here $V = \{s, v_0, v_1, \ldots, v_{n-1}\}$, where all v_i correspond to variables x_i and s is a special start vertex. Edge set E is

$$E = \{(v_i, v_j) : x_i - x_j \geq d_{ij} \geq m_{ij} \text{ is a constraint}\}$$

$$\cup\{(s, v_0), (s, v_1), \ldots, (s, v_{n-1})\}.$$

In the beginning, consider only d_{ij} values. Rewriting (9.1), we have:

$$x_j - x_i \leq -d_{ij}.$$

Then, we can assign the weight of the edge (v_i, v_j) to the value $-d_{ij}$, while the edges (s, v_i) have the weight 0.

Now, if G does not contain a negative cycle, then the system is solvable, and m_{ij} can be removed leaving only d_{ij}. If G does contain a negative cycle, then the weights of the edges in the cycle are increased to meet the constraints on m_{ij} (corresponding d_{ij} values are *decreased*). If the cycle cannot be eliminated, there is no solution. Otherwise, we continue until all the negative cycles are eliminated.

Note. To achieve practically good execution time, I use the Bellman-Ford-Tarjan algorithm with the subtree disassembly method for finding the negative cycles [210]. Since for directed acyclic graphs negative cycle detection can be performed in linear time, the strongly-connected components are searched in advance. So, the non-linear Bellman-Ford-Tarjan algorithm is executed only on strongly-connected components while considering the edges between these components takes linear time as these edges do not form a cycle.

The Application of EQMDC

This section explains how to transform a dialog instance to the input of the EQMDC problem.

The variables I need are as follows. For each component C four variables are introduced to specify the left, right, top and bottom coordinates: x^C_{L}, x^C_{R}, y^C_{T} and y^C_{B}. The variables

that bound the component with its margins are $x^C_{\mathrm{L}M}$, $x^C_{\mathrm{R}M}$, $y^C_{\mathrm{T}M}$ and $y^C_{\mathrm{B}M}$. If C is a container, then the variables for storing C bounds without its border are $x^C_{\mathrm{L}B}$, $x^C_{\mathrm{R}B}$, $y^C_{\mathrm{T}B}$ and $y^C_{\mathrm{B}B}$. Finally, the variables for bounding the inner area of C (without padding) are $x^C_{\mathrm{L}P}$, $x^C_{\mathrm{R}P}$, $y^C_{\mathrm{T}P}$, $y^C_{\mathrm{B}P}$.

The following subsections explain which constraints and terms to be minimized are introduced for 1) absolute sizes; 2) relative sizes; 3) margins, borders, padding and spacing, and 4) gravity and alignment. I provide the constraints for the x-dimension only since the constraints for the y-dimension are similar.

Constraints and minimization terms for absolute sizes

The constraint concerning minimum width, obviously, is:

$$x^C_{\mathrm{R}} - x^C_{\mathrm{L}} \geq \mathit{minimumWidth};$$

But the constraint for the maximum width is being written in the extended form:

$$\begin{aligned} x^C_{\mathrm{L}} - x^C_{\mathrm{R}} &\geq -\mathit{maximumWidth} \\ &\geq -\texttt{MAXIMUM_COMPONENT_SIZE}. \end{aligned}$$

This is the same as

$$\begin{aligned} x^C_{\mathrm{R}} - x^C_{\mathrm{L}} &\leq \mathit{maximumWidth} \\ &\leq \texttt{MAXIMUM_COMPONENT_SIZE}, \end{aligned}$$

but written in the form of (9.1). Here `MAXIMUM_COMPONENT_SIZE` is a big constant, which ensures that the components will not be very large since very large components, which extend far beyond the screen size, are practically unusable.

The preferred sizes are specified not by means of constraints, but as terms of the function to be minimized. I add to the minimization the term

$$c \cdot \left((x^C_{\mathrm{R}} - x^C_{\mathrm{L}}) - \mathit{preferredWidth}\right)^2.$$

Here the penalty for the actual width $(x^C_{\mathrm{R}} - x^C_{\mathrm{L}})$ to be distinct from the preferred width grows quadratically. This ensures that even the component cannot have the preferred size (due to constraints), the actual size will be close to the preferred. The constant c

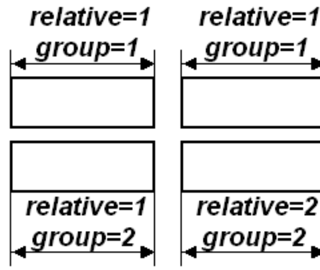


Figure 9.11: An example of the unsatisfiable constrains on relative sizes. The ratio of the widths in the first row is 1:1. The ratio of the widths in the second row is 1:2. Since the components have to be aligned to grid, the ratio constraints become unsatisfiable, if all the components have positive minimum widths.

may be determined in the experimental way taking into a consideration other terms to be minimized.

The minimization terms for relative sizes

Figure 9.11 shows that specifying relative sizes may easily lead to an unfeasible dialog model.

There are two rows, each consisting of two components. The components have to be aligned to grid. All relative widths (minimum, preferred and maximum) for the first two components are set to 1, i.e., the widths of the first row components should be equal. The second component in the second row must be two times wider than the first component in the same row. Obviously, the only solution is when all the components have zero widths. By specifying some positive *minimumWidth* values for all these components, the dialog model becomes unfeasible. To make the layout engine flexible, I use the method described below, which will find an approximate solution, should such a situation as in Figure 9.11 occur.

Assume there are two components, A and B, in the same relative width group (for the heights the method is similar). Let them have preferred relative widths r_1 and r_2, and let x_{L}^A and x_{R}^A be variables for the left and the right bounds of the first component as well as x_{L}^B and x_{R}^B for the second. Then, obviously, the desired equation is:

$$r_2 \cdot (x_{\mathrm{R}}^A - x_{\mathrm{L}}^A) = r_1 \cdot (x_{\mathrm{R}}^B - x_{\mathrm{L}}^B). \quad (9.2)$$

Since not always this equation may be satisfied by the reasons mentioned above, it is better to replace it with the approximate equation. Let's write (9.2) in this form:

$$r_2 \cdot (x_{\mathrm{R}}^A - x_{\mathrm{L}}^A) - r_1 \cdot (x_{\mathrm{R}}^B - x_{\mathrm{L}}^B) \approx 0.$$

This form may be rewritten as a term to be minimized by quadratic optimization algorithm:

$$\left(r_2 \cdot (x_{\mathrm{R}}^A - x_{\mathrm{L}}^A) - r_1 \cdot (x_{\mathrm{R}}^B - x_{\mathrm{L}}^B)\right)^2. \tag{9.3}$$

In case all the constraints can be satisfied, this term will be zero, and, thus, the desired relation will hold. Otherwise, the difference between the desired and the actual relation will tend to be zero (with a quadratic penalty).

Since for any positive k the relative widths $k \cdot r_1$ and $k \cdot r_2$ denote the same relation as r_1 and r_2, the quadratic term (9.3) has to be the same if r_1 and r_2 are multiplied by k. So, prior to creating the term (9.3), the following normalization has to be performed:

$$\begin{pmatrix} r_1 \\ r_2 \end{pmatrix} \leftarrow \begin{pmatrix} \frac{r_1}{r_1+r_2} \\ \frac{r_2}{r_1+r_2} \end{pmatrix}$$

If minimum/maximum relative sizes are also specified, then they are used as follows. Let $(C_1, r_1, s_1), (C_2, r_2, s_2), \ldots, (C_n, r_n, s_n)$ be the triples where r_i denote minimum relative widths and/or heights contained in the same group. C_i are the corresponding components and s_i are equal to corresponding (absolute) minimum width and/or height values.

A coefficient k is calculated first:

$$k \leftarrow \max\left\{\frac{s_i}{r_i}\right\}.$$

Then, for each C_i the corresponding *minimumWidth* or *minimumHeight* are adjusted to the value $k \cdot r_i$.

The same refers to maximum sizes with the following differences:

- For all the triples (C_i, r_i, s_i) with the r_i set to maximum relative width/height and s_i set to maximum width/height, r_i and s_i must be defined.

- To calculate k, we have to $\min\{s_i/r_i\}$ instead of $\max\{s_i/r_i\}$.

Constraints for margins, borders, padding and spacing

The following constraints have to be used for the margins:

$$\begin{aligned} x^C_{\mathrm{L}} - x^C_{\mathrm{L}M} &\geq \mathit{leftMargin}, \\ x^C_{\mathrm{R}M} - x^C_{\mathrm{R}} &\geq \mathit{rightMargin}. \end{aligned}$$

If C is a container, the borders are specified by these constraints:

$$\begin{aligned} x^C_{\mathrm{L}B} - x^C_{\mathrm{L}} &\geq \mathit{leftBorder}, \\ x^C_{\mathrm{R}} - x^C_{\mathrm{R}B} &\geq \mathit{rightBorder}. \end{aligned}$$

Finally, if C is a non-scrollable container, then the following constraints are introduced for padding:

$$\begin{aligned} x^C_{\mathrm{L}P} - x^C_{\mathrm{L}B} &\geq \mathit{leftPadding}, \\ x^C_{\mathrm{R}B} - x^C_{\mathrm{R}P} &\geq \mathit{rightPadding}. \end{aligned}$$

In case of a scrollable container the second constraint is not added.

The spacing between two components A and B (A on the left of B) is introduced by the constraint

$$x^B_{\mathrm{L}M} - x^A_{\mathrm{R}M} = \mathit{horizontalSpacing}$$

(equation can be replaced by two inequalities).

Gravity and alignment

Assume there are nested components (see Figure 9.12).

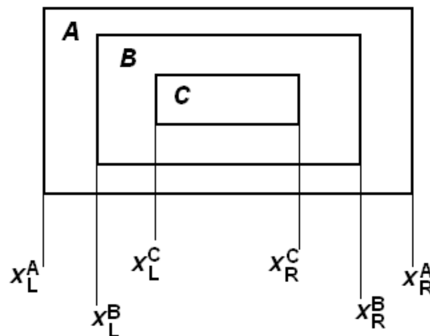


Figure 9.12: The nested components. There exists gravity between the borders that forces the inner component to be resized (unless it has the maximum size specified) when the outer component is resized.

If Component C is not resizeable, Component B should still follow Component A, when A is being resized. Thus, the gravity between components B and C should be less than between A and B.

Referring to Figure 9.12, gravity tends to minimize the following differences: $x_{\mathrm{L}}^{C} - x_{\mathrm{L}}^{B}$, $x_{\mathrm{R}}^{B} - x_{\mathrm{R}}^{C}$, $x_{\mathrm{L}}^{B} - x_{\mathrm{L}}^{A}$, and $x_{\mathrm{R}}^{A} - x_{\mathrm{R}}^{B}$. The first two differences must be "weaker". That is, if C is not resizeable, then there should be no gravity between B and C. In order to achieve this, I use the following *linear* terms to be minimized:

$$k \cdot (x_{\mathrm{L}}^{C} - x_{\mathrm{L}}^{B}) + k \cdot (x_{\mathrm{R}}^{B} - x_{\mathrm{R}}^{C}) + l \cdot (x_{\mathrm{L}}^{B} - x_{\mathrm{L}}^{A}) + l \cdot (x_{\mathrm{R}}^{A} - x_{\mathrm{R}}^{B}),$$

where $k < l$. Assume C is not resizeable, and A has just been stretched. That means the the sum $(x_{\mathrm{L}}^{C} - x_{\mathrm{L}}^{A}) + (x_{\mathrm{R}}^{A} - x_{\mathrm{R}}^{C})$ has been fixed. If $k < l$, then the sum of the four terms will reach its minimum, when the last two terms are zero. That is also the desired behaviour for the gravity. If B is a vertical box with n children, the "weight" k should be divided by n since the minimization terms involved by all the children sum up.

The following inequalities have also to be specified:

$$\begin{aligned} x_{\mathrm{L}}^{C} - x_{\mathrm{L}}^{B} &\geq 0, \\ x_{\mathrm{R}}^{B} - x_{\mathrm{R}}^{C} &\geq 0, \\ x_{\mathrm{L}}^{B} - x_{\mathrm{L}}^{A} &\geq 0, \\ x_{\mathrm{R}}^{A} - x_{\mathrm{R}}^{B} &\geq 0. \end{aligned}$$

If the children have to be left-aligned, then instead of gravity between the left container border and the first child the constraint

$$x_{\mathrm{L}}^{B} - x_{\mathrm{L}}^{A} = 0,$$

is introduced (here A is a container and B is the left child). The same approach is used, when the children have to be right-aligned. However, if the children are to be centred, the term

$$c \cdot \left((x_{\mathrm{L}}^{B} - x_{\mathrm{L}}^{A}) - (x_{\mathrm{R}}^{A} - x_{\mathrm{R}}^{B})\right)^{2}$$

is added to the minimization. This makes the distances from the component to the left and right borders of the parent equal. Finally, the constraints

$$\begin{aligned} x_{\mathrm{L}}^{B} - x_{\mathrm{L}}^{A} &\geq 0 \\ x_{\mathrm{R}}^{A} - x_{\mathrm{R}}^{B} &\geq 0 \end{aligned}$$

have to be added.

9.3 Related Work

There are several ways of specifying dialog boxes. One is to use graphical designers, which may be either stand-alone programs (like GLADE [211]), or parts of IDEs (Integrated Design Environments) such as Borland C++ Builder, Microsoft Visual Studio, and NetBeans. Such designers are usually developed for a specific widget toolkit or library (e.g., GLADE is tailored for the GTK+ library, Borland C++ Builder uses VCL (Visual Component Library), Microsoft Visual Studio uses the `Windows::Forms` library, while NetBeans uses the Swing library).

There are also user interface (UI) libraries that do not have designers. In this case dialog boxes are specified in program code that uses the routines of the particular library. Such code can also be written for the libraries that do have graphical designers.

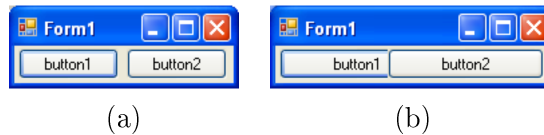
Another way for specifying dialogs is using textual languages. HTML (Hyper Text Markup Language) is an example of such language, since it allows GUI components to be placed to web pages. Other examples include User Interface Markup Language, UIML [212] and UsiXML [213].

Many toolkits allow the developer to specify absolute coordinates (like coordinates of the left-top corner) and dimensions (i.e., width and height) GUI components. Sometimes, absolute coordinated can be avoided by using tables (HTML), boxes (GLADE), or other mechanisms (NetBeans UI designer uses horizontal and vertical groups to lay out the components by means of the *GroupLayout* manager [214]).

Java *Swing* library contains several layout managers for laying out and resizing GUI components [215]. A layout manager is associated with a container, so the elements inside that container are laid out depending on the layout manager.

As of specifying resizeable components, some tools (Borland C++ Builder and Delphi, Microsoft Visual Studio) have a possibility to set up *anchors*, i.e., to fix the distance

between the component and one or several window borders. When a window is resized, the component is relocated or resized to keep these distances constant. This is useful when there is one large component that has to be resized along with the window. However, if several components have to be resized simultaneously, anchors can lead to the problem depicted in Figure 9.13: when the form is resized, the buttons overlap.



(a) (b)

Figure 9.13: (a) A form with two buttons where left and right anchors are set. (b) The form after resizing.

The Windows Presentation Foundation (WPF) [216] is a platform for creating rich user interfaces in Windows applications. WPF uses *panels* to lay out child components (panels are similar to containers). WPF uses alignments (similar to *horizontalAlignment* and *verticalAlignment* properties), padding (similar to borders and padding) and margins. The difference is in stretching the components: in Dialog Engine Metamodel, the component is stretched until it reaches the maximum size, while in WPF a special alignment constant "`Stretch`" is introduced to denote that the component has to be stretched.

An interesting idea for specifying both absolute and relative sizes is based on usage of linear constraints [217]. By means of constraints, the layout and behaviour of components can be specified in a more flexible way. However, defining the constraints explicitly by means of equations and inequalities is not a natural way for specifying properties of UI components. Moreover, the question arises concerning what to do if the constraints are unsatisfiable. UI may be generated at runtime, and the components should be laid out despite inconsistent constraints. Dialog Engine solves this problem by allowing the maximum sizes to be increased, when needed.

Several web-based techniques with very rich capabilities for creating user interfaces are available for developers today. AJAX is an approach, where modern web technologies are used to provide fast responses to the user [218]. If the client-side AJAX engine can handle the user request by its own, it does so. Otherwise, a request (usually, asynchronous) to the server is performed. Google Web Toolkit, GWT [219] is a framework that uses the AJAX principles. Its feature is that web-based applications are developed in Java, while at runtime web-based technologies such as JavaScript and HTML are used.

Microsoft Silverlight [220] and Adobe Flash [221] are two other platforms for providing enhanced user-interface experience including interactivity and animations.

The XForms XML format can be used for specifying user interfaces along with data processing at the client side. XForms is "the next generation of forms technology for the world wide web" [222].

Since there exist algorithms for user interface layout that use linear constraints [223, 224], one may be interested why the quadratic (and not linear, or, possible, cubic) optimization is used in Dialog Engine. The two reasons can be mentioned:

1. It is impossible to implement some constraints (like constraints for preferred sizes) by means of a linear function only.

2. Optimizing other non-linear (cubic et al.) functions can be very time-consuming.

Regarding the implementation of QMDC, any method can be used here. While the current implementation of Dialog Engine uses the method by Freivalds and Ķikusts [208], one could use the method by Hosobe, for instance [225].

There is an interesting difference in resizing policy between the approach used by Dialog Engine and the approach used in the QT library [226]. In QT, when a layout (such as vertical, horizontal, or grid layout) for a container is set, components inside this container are resizeable by default. To prevent resizing, special components called horizontal and vertical spacers can be used. Spacers act as springs that produce a counterforce for resizing. In contrast, my approach uses maximum width and height constraints to prevent resizing.

Model-driven graphical tool building platforms such as Eclipse GMF [172, 173], Microsoft DSL Tools [58], Metaclipse+ [183], and others, usually provide a standard mechanism for creating dialog boxes such as property editors. These standard mechanisms permit only limited customizations of a dialog box (e.g., we can specify the names of properties and their values, but we cannot add some extra buttons). While the expressiveness of such simple dialogs is sufficient for most cases, some platforms (like Microsoft DSL Tools) permit also specifying dialogs of arbitrary complexity in an object-oriented programming language (e.g., C#). But the additional knowledge and skills are required here. Moreover, the model-driven approach is lost.

As noticed by Dmitrijs Logvinovs, the number of variables used in quadratic optimization can be reduced up to two times by combining them (e.g., variables for margins can be combined with the corresponding component coordinates). He also started an alternative Java implementation that reduces the number of variables and takes an advantage of using Java reflection mechanism for loading GUI elements at runtime.

Chapter 10

Error Engine

Even when a system is model-driven, it needs some mechanism to display error messages to the user. "The connection to the network has been lost"; "The file already exists"; "The inheritance loop is detected (in the class diagram)" — these are just some of the examples. Error messages can contain variable parts, e.g., file names, numbers, etc. Thus, a way to describe the fixed message text and its variable parts is needed. When multiple similar errors occur, it may be reasonable to accumulate them in a single error message (so the user does not need to click "OK" 100 times). When an error was caused by other (technical) errors, developers and advanced users may need to obtain the detailed information about those technical errors, thus, some way to form error hierarchies is needed.

In TDA, a universal way for handling errors and to deal with the issues just mentioned is provided by Error Engine[1]. An interesting property of Error Engine is that Error Engine uses the meta-meta level of abstraction. Since for repositories in a given technical space the linguistic meta-metamodel is usually fixed, it is impossible to define new meta-classes there. Thus, when speaking about meta-meta level in this chapter, I refer to the quasi-ontological meta-level, Ω3. Error Engine defines certain meta-classes and meta-properties there. Thus, a repository that supports (or is able to simulate) at least three quasi-ontological meta-levels is required by TDA Error Engine.

[1] Although I call this engine Error Engine, it can be used as a notification engine as well.

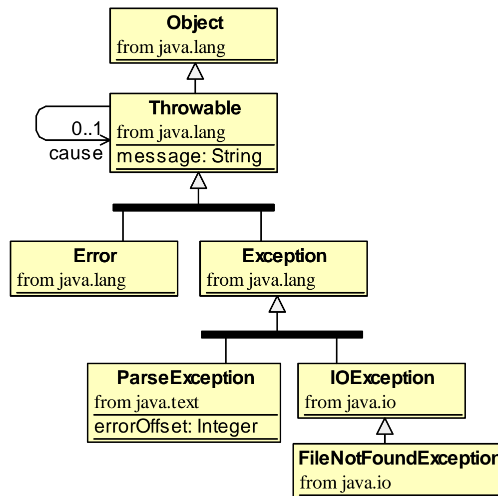


Figure 10.1: A fragment of the Java exception hierarchy.

10.1 Error Handling and the Quasi-Ontological Meta-Meta Level

Modern languages and environments supporting object-oriented programming (OOP) have the notion of exception. Usually, all exception types are defined as classes, which form some exception hierarchy. Figure 10.1 shows a fragment of Java exception hierarchy in the form of a metamodel [227].

As we can see, exceptions are subclasses of the *java.lang.Exception* class.[2] Each exception class may have attributes describing details of the exception (see attribute *errorOffset* in class *ParseException*, for example). The *cause* property allows exceptions to be chained, since one exception may cause another exception to be thrown.

The .NET Framework [228] also has its own exception hierarchy. Exceptions are derived from the *System.Exception* class, which has the *InnerException* property similar to the *cause* property in Java.

Exception classes naturally correspond to the quasi-ontological meta-level (Level Ω2), in which metamodels are described. Particular occurrences of exceptions are instances (objects) of those classes. These instances correspond to Level Ω1.

To be able to define error types in TDA, I introduce the *Error* class in TDA Kernel Metamodel — a root class for the error hierarchy. The *Error* class is similar to the *Throwable* class in Java. The *Error* class can be extended in interface metamodels of

[2]The class *Exception* is derived from *Throwable*, because there exists another subclass derived from *Throwable* — the class *Error*, and Java differentiates exceptions from errors.

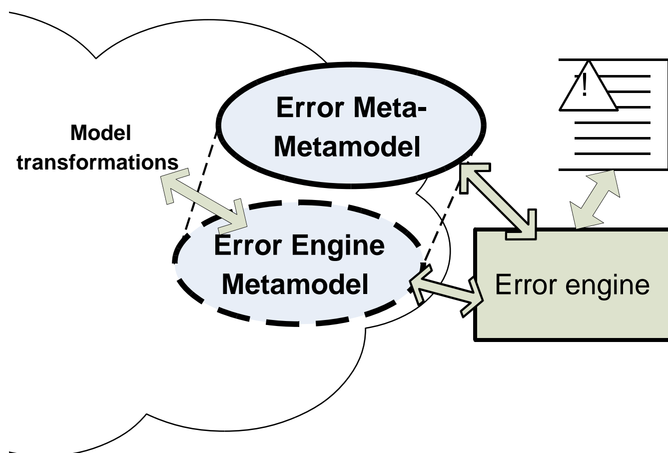


Figure 10.2: Adding Error Engine to TDA.

engines by defining types of errors the engines can produce. Thus, defining new types of errors is similar to defining new types of events and commands in TDA.

However, there is also a difference between events/commands and errors. Commands are processed by engines, and each particular engine has to be aware of a finite number of command types. The same is true for events. When some transformation is assigned to handle certain events, the number of events the transformation has to be aware of is also finite. But errors are processed by Error Engine, which cannot foresee which engines will be plugged in, and which errors types they will define. Thus, Error Engine has to deal with a potentially unbounded number of error types.

Still, Error Engine is aware of certain aspects. It can assume that error types form a hierarchy. Besides, it can assume that error messages for the same class of errors are similar (possibly, with some variations like "File <filename> not found."). Thus, some class-level attribute for each error message would come in handy. Since the notion of the class hierarchy and the notion of the class-level attribute are from the meta-meta level (Level Ω3), it can be reasoned that what Error Engine could be aware of in advance is not a metamodel (like in case of ordinary TDA engines), but a meta-metamodel.

Figure 10.2 depicts, how Error Engine embeds into TDA. Error Engine Metamodel consists of the whole TDA error hierarchy (class *Error* and all its descendants) as well as of the two events, which will be introduced later. Error Engine Metamodel is not simply yet another interface metamodel, since it contains error classes from several interface metamodels, and it is not known in advance how many error classes will be in the hierarchy. To deal with Error Engine Metamodel, Error Engine needs Error Meta-Metamodel, which can be used to discover the actual Error Engine Metamodel. Then, given a particular

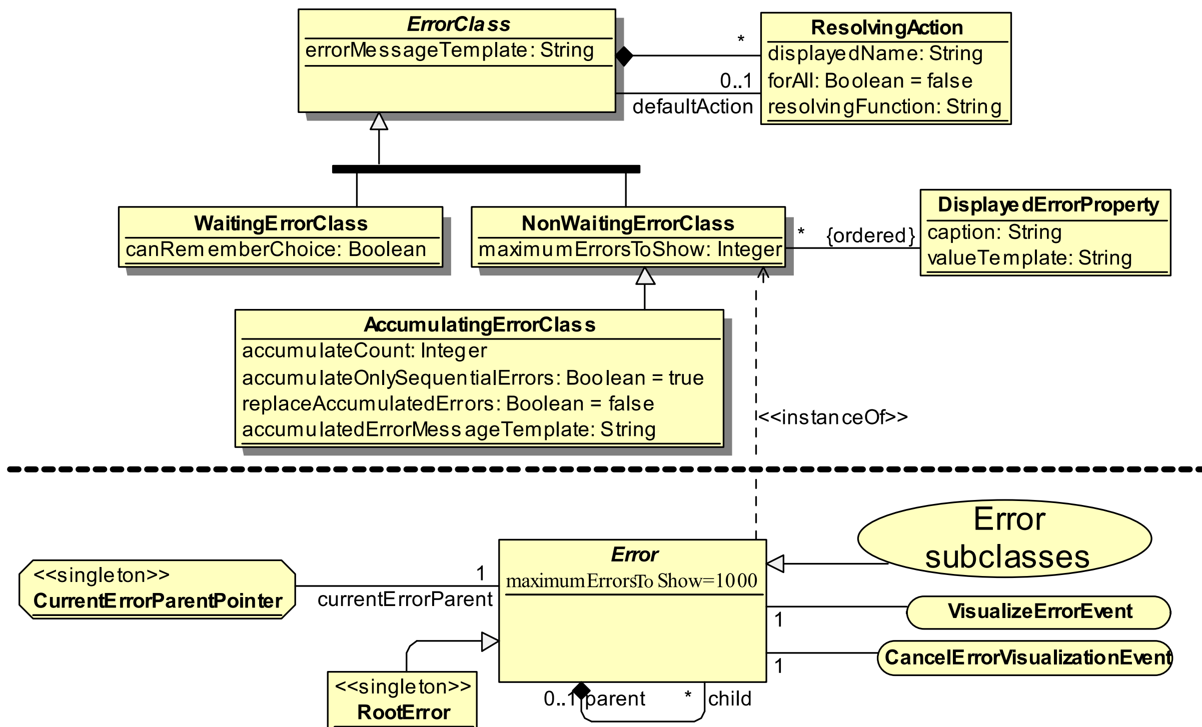


Figure 10.3: Error Meta-Metamodel (on top of the bold dashed line) and some Level Ω2 classes (such as *CurrentErrorParentPointer*, *Error* and *RootError*) defined by Error Engine.

error instance (at Level Ω1, not shown in Figure 10.2) and Error Meta-Metamodel (Level Ω3), Error Engine can obtain the required information from the error type residing in Error Engine Metamodel (Level Ω2) and display the error message.

Error Meta-Metamodel contains metaclasses (types for error types) that define certain class-level properties. These properties are used by Error Engine to decide how to display errors of each particular type. Since errors of the same type have to be displayed in the same way, it is reasonable to specify these properties only once for each error type (that is why these properties are class-level properties). The next section explains Error Meta-Metamodel.

10.2 Error Meta-Metamodel for Describing Errors

Error Meta-Metamodel for describing error types is depicted in Figure 10.3.

Classes with shadows are metaclasses for defining error classes at Level Ω2. Error types (error classes) will be instances of metaclasses *WaitingErrorClass*, *NonWaitingErrorClass*, and *AccumulatingErrorClass*, which all inherit from the common abstract super-metaclass *ErrorClass*. Error types will also be direct or indirect subclasses of the common superclass

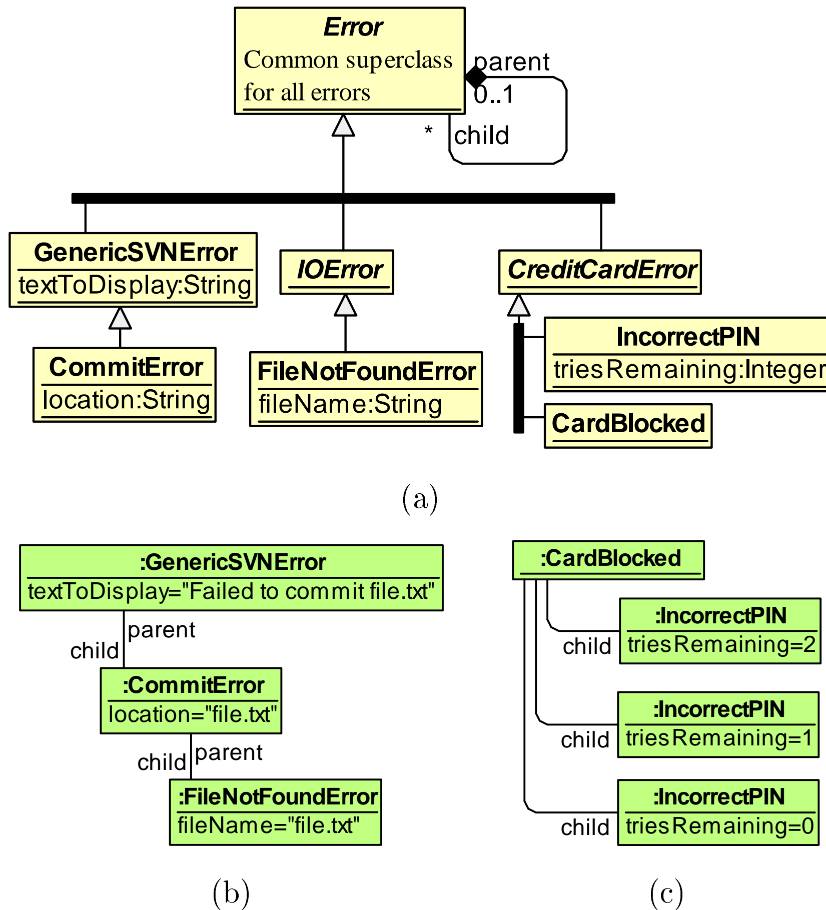


Figure 10.4: (a) A sample error metamodel (an instance of Error Meta-Metamodel). (b) Sample error instances and the *parent-child* links. (c) Sample error instances, where the *parent-child* links form a tree.

Error, which is an ordinary class at Level Ω2. The inheritance relationship between error types can be used to form error hierarchies like in Figure 10.4(a).

The *parent-child* relationship of the *Error* class is used to navigate to deeper errors, which caused the particular error object to be created, see Figure 10.4(b). The *parent-child* relationship is similar to the Java *cause* property and to the .NET *InnerException* property, but it can be used not just for chains of "inner" errors like in .NET and Java, but also for trees, since several consecutive errors, not just one, could lead to the "outer" error (see Figure 10.4(c)).

The following subsections will explain more in detail classes found in Error Meta-Metamodel.

ErrorClass

The *ErrorClass* class defines common properties for all error types.

The class-level attribute *errorMessageTemplate* is intended for storing the error message common to all errors of the given type. For example, there may be an error class called *FileNotFoundError* having the *errorMessageTemplate* value equal to:

`File not found!` .

Sometimes it is reasonable to include some details of a particular error instance, e.g., a particular file name:

`File document.txt not found!` .

Such variations could be allowed by making the *errorMessageTemplate* attribute computable. However, this is not convenient, since defining an error type would mean defining also a function to compute the value of an error message. A parametrized string could do the job as well. For example, we could write:

`File ${fileName} not found.` ,

where `${fileName}` denotes the value of the attribute *fileName* of a particular *FileNotFoundError* instance (this attribute must exist). And, if computable attributes are supported, the *fileName* value can also be calculated, when needed.

Another way of introducing variations in error messages is by means of OCL-like expressions, e.g.,

```
File ${self.fileName} not found
on the remote computer ${self.computer.name}.
```

(assume that it is possible to navigate from the error instance by the "computer" role and take the value of the *name* attribute then). For convenience, `self` pointer could be omitted, so one could write `${fileName}` instead of `${self.fileName}`. OCL-like expressions do not deny computable attributes, but provide additional flexibility in defining error messages.

Sometimes errors are also requests to the user for an action. For example, when copying several files, some file may already exist, and the user should choose between options like 'Yes', 'Yes for all', 'No', 'No for all', etc., when asked for a confirmation to overwrite

an existing file. In the proposed Error Meta-Metamodel, such options are specified as *ResolvingAction*-s. When the user selects an option, the corresponding *resolvingFunction* is invoked (this function may be a model transformation or some other function; it must take an *Error* instance as an argument). The *forAll* attribute denotes that the given resolving action has to be invoked for all errors of the same type. The *displayedName* attribute contains the name of the operation shown to the user (e.g., "Yes for all").

In certain cases it is not obligatory to wait for the user to choose the action, e.g., the system may continue copying other files. However, the error object should contain all necessary information (the context) in order the chosen action could be performed later.

Depending on whether the user interaction is required before the processing can be continued I divide all error types into two disjoin sets — waiting error types and non-waiting error types. The following two subsections describe the corresponding metaclasses *WaitingErrorClass* and *NonWaitingErrorClass*.

WaitingErrorClass

Waiting errors require user interaction before the processing can be continued. The expected behaviour[3] of Error Engine is to display a modal window with possible choices (resolving actions) how to resolve the error. The *canRememberChoice* attribute denotes whether Error Engine has to provide an option (e.g., a check box) for applying the same choice for further errors of the same type.

NonWaitingErrorClass

When a non-waiting error occurs, the processing can continue without waiting for the user interaction. The expected behaviour of Error Engine on such an error is to append it to some error list shown to the user (compiler warnings and errors are usually shown in this way). The *maximumErrorsToShow* attribute may be used to limit the number of error messages shown in the list: the older messages will automatically disappear from the screen, when the number of errors exceeds the value of *maximumErrorsToShow*.

If a non-waiting error has resolving actions, Error Engine displays the corresponding options (e.g., some buttons near the error message), but does not wait for the user to click

[3]Although I mention the expected behaviour, there may be different variations of Error Engine, and certain features may be implemented there in some other manner.

on them. When the user clicks on some option, the corresponding resolving function is placed into the queue — it will be executed after the main processing finishes.

The error list mentioned above actually can be a table. In compiler messages, file names and line numbers containing errors are usually shown in separate columns. Such columns can be instances of the *DisplayedErrorProperty* metaclass. Each such column has a *caption* and a *valueTemplate*. Each value template is a string, which describes the values for the corresponding column. It can also contain OCL-like expressions, e.g., `${this.source.fileName}` and `${this.token.lineNumber}`.

When too many similar errors occur, some cumulative error may be shown to the user (either appending, or replacing the errors that are being accumulated). For example, a compiler may summarize how many errors and warnings were there in code, or a function for copying files may show only one error message for 100 files instead of 100 messages, one message per file. For dealing with such error types, I have introduced the *AccumulatingErrorClass* class in Error Engine Meta-Metamodel, which is explained in the following subsection.

AccumulatingErrorClass

For errors that are allowed to be accumulated there is the metaclass *AccumulatingErrorClass*. The value of the *accumulateCount* attribute specifies the threshold for the number of errors, when errors are to be grouped. Until the threshold is reached, accumulating errors are shown as non-waiting errors. The *accumulateOnlySequentialErrors* value specifies whether the threshold can be reached only by sequential errors of the given kind, or errors of other kinds may intervene without restarting the counter. When the threshold has been reached, the corresponding *accumulateCount*−1 error messages are replaced or appended (depending on the *replaceAccumulatedErrors* value) by the accumulated error message specified as a value of the *accumulatedErrorMessageTemplate* attribute.

> *Example 1.* The value of *replaceAccumulatedErrors* for compiler errors may be set to `false`, and the value for *accumulatedErrorMessageTemplate* may be: “`Too many syntax errors.`”.

If several kinds of errors need to be accumulated, a common supertype may be introduced.

> *Example 2.* Having two error types *FileCopyError* and *FolderCopyError*, a common superclass *CopyError* can be introduced. The

errorMessageTemplate for *FileCopyError* and *FolderCopyError* may be, respectively, "`Could not copy the file ${fileName}.`" and "`Could not copy the folder ${folderName}.`", while the *accumulatedErrorMessageTemplate* for the superclass *CopyError* may be:

"`Could not copy ${FileCopyError.allInstances->size()} files and ${FolderCopyError.allInstances->size()} folders.`".

10.3 Handling Errors

This section explains how error instances are created by model transformations and engines, and how such error instances are handled by Error Engine.

When TDA undo/redo is invoked, a bundle of actions is being undone/redone at once (see Chapter 7). Each such bundle of actions is a single logical action from the user's point of view. When a new logical action is started, the user usually is not interested in errors from previous logical actions. Thus, I assume that only errors within one logical action have to be managed and stored in the model repository according to Error Engine Metamodel[4].

Since TDA Kernel knows when a new bundle of actions (a new logical action) starts, it performs some clean-up by deleting previous error objects from the model repository. Before each new bundle of actions TDA Kernel also re-creates a singleton instance of the *RootError* class (see Figure 10.3) and sets the current error parent pointer (a singleton instance of the *CurrentErrorParentPointer* class) to the root error object.

When transformations and engines perform their tasks and communicate by means of events and commands, errors may occur. To report an error an engine/transformation simply creates an instance of the corresponding error type and attaches it to the current parent. If the current parent is the root error object, Error Engine displays the error. Errors not attached to the root error directly are not displayed[5]. The following scenario explain the motivation behind this design choice. Assume that an engine or a transformation needs to call some subroutine, which can produce error objects. If those errors are too technical and should not be shown to the user, the engine/transformation creates a stub parent error object and moves the current error parent pointer to this stub

[4] Error Engine may still display the whole history of error messages, but other engines and transformations work only with errors within the current logical action.

[5] Actually, Error Engine may provide access to these nested errors for developers and advanced users, but for simplicity I assume that nested errors are not shown.

object. From this moment errors created by the subroutine will be attached to the stub object. Since these errors are not directly connected to the root error (e.g., *IncorrectPin* errors from Figure 10.4(c)), they will not be shown. When the subroutine finishes, the engine/transformation may replace the stub object with some more suitable error object (depending on the nested errors produced by the subroutine), keeping the nested error hierarchy, when needed (e.g., for providing additional details). Then, the current parent pointer should be moved back to the previous parent.

Each time a new error object is created, Error Engine is triggered[6]. Error Engine tracks the *CurrentErrorParentPointer* links to determine which errors have to be displayed and which not. When an error has to be displayed, Error Engine retrieves the corresponding class-level information stored according to Error Meta-Metamodel, and automatically displays the error (e.g., via a modal message box, or by appending it to the list of errors).

When displaying an error, Error Engine can use some internationalization library to display the error message in the language preferred by the user. To determine that language, Error Engine should use the *language* and *country* attributes, defined in Environment Engine Metamodel. Since Error Engine uses templates for messages, only one string for each error type has to be translated.

For certain errors it is not sufficient to simply show error messages. Some error visualization may be expected, when the user clicks on an error message. For example, if an error is related to a diagram, the corresponding graphical element may be marked in red. If the error is a syntax error in some code, the caret may go to the corresponding line.

Error Engine uses the standard TDA communication mechanism to inform transformations, when the errors should be visualized. The two event classes, namely, *VisualizeErrorEvent* and *CancelErrorVisualizationEvent*, are introduced for that (see Figure 10.3). When some error has to be visualized, a *VisualizeErrorEvent* instance is created by Error Engine and linked to the corresponding error instance. When there is no more need to have the error visualized, Error Engine creates a *CancelErrorVisualizationEvent* instance.

[6]Technically, this triggering can be implemented by introducing a virtual repository that defines classes *Error*, *RootError*, and *CurrentErrorParentPointer* with the corresponding associations. This repository then can be merged with the main repository. Each time a parent-child link is created, the virtual repository gets the control. Other solutions for implementing triggers are possible as well.

10.4 Related Work

Windows API [229] is an excellent example of detecting and displaying errors in a procedural world. Windows functions usually do not display error messages, but in case of an error some error-indicating value (like zero, `NULL`, -1, `false`, etc.) is returned. Such a value only indicates that some error has occurred. More details (e.g., the kind of the error) can be obtained by calling the *GetLastError* function, which returns the corresponding integer value. The programmer may decide, whether to format and show the error message depending on that integer value, or to handle the error in some other way, perhaps, also returning some error-indicating value. Returning and checking such values is very fast — each of these values usually occupies a single machine word. However, a single integer value may be insufficient to provide all the details about the error. Certain functions can be introduced for providing additional details, but then this approach becomes not so elegant since the programmer has to know which of these additional functions to call for each error type. In contrast, the TDA error mechanism is not so fast: instead of dealing with machine words, it deals with error objects in a model repository. However, this brings the following benefits:

- The detailed information about the error can be specified in properties of the error object. Thus, no additional functions are needed for that.
- The programmer does not have to format the message: each error class has a message template, which can be automatically transformed to a real error message by Error Engine.

In the object-oriented world, error handing is usually based on exception hierarchy. Exceptions are classes, which may have properties for encoding detailed information about errors. My solution also uses similar principles — error types correspond to classes, which form error hierarchies. Besides, the proposed Error Meta-Metamodel allows the toolsmith to specify possible actions for resolving errors (such as "Yes", "No", "Yes for all", etc.). Properties that should be displayed in a tabular form can also be defined. TDA Error Engine automatically formats and displays the corresponding error message freeing the programmer from that job. In addition, the proposed solution allows several errors of the same kind to be grouped together.

There are also some model-based solutions for managing errors. For example, the work of Brambilla et al. [230] introduces two classes for handling errors: the *ExceptionType* class

with the *name* attribute and the *ExceptionInstance* class with three specific attributes. Each exception instance is linked to exactly one exception type. Another example can be found in the presentation metamodel of the MOLA tool based on the Metaclipse platform [183]. There exists a *CompilerMessage* class with two attributes — one for the error message, and another for the error type (warning or error). There is also a transformation that is used to visualize errors graphically. Although both mentioned model-based approaches are excellent for particular purposes, they are not as universal as the proposed TDA error mechanism.

Discussion

Scientific and Practical Significance of the Research

The main principle behind TDA is the use of models and transformations at runtime. The scientific novelty of the research is refinement and formalization of this principle in the form of TDA specification. There is no known software architecture similar to TDA.

TDA uses models as a universal encoding for data, and it uses model transformations as a universal means to describe the business logic of a system. This refers to the practical value of the thesis: the use of models and transformations increases the abstraction level, at which the system components and business logic are described. This increases the productivity of programmers and speeds up the development process. Interface metamodels, which are used for describing engines, are important documentation artefacts that (when represented graphically) improve the readability of specifications of engines and reduce the size of the required written documentation. The IMCS-UL experience shows that even undergraduate students can work with TDA and use documentation expressed in the form of metamodels.

The fact that TDA ensures interoperability between components of different types allows system developers to choose programming and transformation languages as well as model repositories according to the actual needs. In addition, it reduces software development costs, since newly created TDA engines can be reused in other systems. Off-the-shelf third-party engines (when available) can be reused as well.

A scientific significance of the TDA undo/redo mechanism is based on the fact that it is a universal solution for all TDA-based systems. In addition, this mechanism has also a practical value: since it is automatic, developers do not have to think about undo/redo (except for specific cases).

TDA ability to work with multiple model repositories of different types is also important: it can be used to combine knowledge and tools from different technical spaces

(technologies such as XML, formal grammars, relational databases, semantic web technologies, MOF-like modelling techniques, etc. [41, 40]).

The concept of Environment Engine is an abstraction layer that allows TDA to be launched in different ways: as a PC application, as an application on a touch screen device, or as a component of another program. Thus, TDA has a potential to be used on tablet computers and smartphones.

Dialog Engine Metamodel is a new way of specifying dialog windows (usually they are described in the program code, XML files, or by means of graphical designers). Exact dimensions and positions of dialog elements may be omitted, since they can be computed automatically by a special layout algorithm proposed in the thesis. From the practical point of view, this is a convenient solution to generating graphical user interface at runtime.

Error Engine Meta-Metamodel is a new universal meta-language for describing taxonomies of error and information messages of different types.

TDA has proven its effectiveness in practical applications, including uses in state agencies SEDA[7] and SSIA[8] [3] (see Section "Applying the results in practice" on page 14 and Section 4.4).

Limitations of TDA

Probably, the most obvious TDA limitation is its performance slowdown. Since data have to be stored as models in a model repository, the repository-related overhead is involved. Programming tricks with direct memory access are not available any more, if data are accessed via RAAPI. Furthermore, TDA Kernel introduces additional slowdown via the undo/redo mechanism and indirections such as proxy references and repository adapters. Model transformations are also subject to these efficiency issues, since in TDA they are executed at runtime. Nevertheless, the performance slowdown involved in current applications is not essential: TDA works without observable delay on modern PC's. Thus, that is a reasonable price for the benefits offered by TDA. Taking into a consideration the ever increasing power of computers, it can be said that current performance issues in TDA are not issues tomorrow. On the other hand, taking into a consideration the "No Silver Bullet" conjecture by Frederick P. Brooks, the problem of raising the productivity

[7]State Education Development Agency (SEDA, Latvian - VIAA)
[8]The State Social Insurance Agency (SSIA, Latvian - VSAA)

of programmers will be the problem also tomorrow [21, 231, 232]. TDA aids here by allowing to raise the level of abstraction as well as by solving certain accidental problems (such as interoperability of different system components).

In TDA, interface metamodels are important artefacts that have to be thoroughly designed. The process of designing an interface metamodel for a TDA engine is similar to defining a cross-platform and cross-language API for a particular software library. Interface metamodels have to be created by skilled developers. TDA does not aid here. Besides, TDA does not specify how to map interface metamodels to particular functionality offered by engines. This process can be non-trivial (as in the case of Dialog Engine, which uses quadratic optimization). Still, when an engine has been developed, it can be reused in different tools (including third-party tools).

On the one hand, the requirement to use interface metamodels in TDA (as opposed to native APIs and direct access to data structures) is a limitation: all operations must be performed via a model, even the communication (by means of events and commands). On the other hand, this approach makes TDA independent on any particular implementation of engines and transformations.

The requirement to use a particular interface for accessing models (RAAPI) can be neglected by the fact that RAAPI can be mapped to different programming languages and used from different platforms. Higher-level wrappers over RAAPI can also be created. Moreover, it is possible to generate, for example, Java or C++ classes that reflect the structure of the repository and use RAAPI internally, while the user can work with the generated classes as with native Java or C++ classes.

TDA as Foundation for Other Research

Along with TDA, this thesis presented three important engines — Environment Engine, Dialog Engine, and Error Engine. These engines have a potential to be included in most TDA-based systems having graphical user interface. TDA, its kernel, and these three engines define the core of such systems. From this point of view, the thesis is a finished research.

By extending the core presented in this thesis, the whole model-driven world can be built. For instance, Artūrs Sproģis is doing the research on building domain-specific tools on the TDA foundation. Mārtiņš Zviedris and Renārs Liepiņš use the results of

A. Sproģis to build particular tools for the semantic web area. The model-driven world is indeed being built.

To provide some idea on possible types of engines that can be used within TDA, I would like to mention three engines developed at IMCS-UL[9]:

- *Graph Diagram Engine* is a complex engine for visualizing and editing graph diagrams [2, 5][10]. Currently, all domain-specific tools developed at IMCS-UL are based on graph-diagrams, thus, they use this engine. Like Dialog Engine, Graph Diagram Engine uses quadratic optimization to lay out diagrams, although in a different way [208].
- *Word Engine* is a simple engine for generating Word documents on-the-fly via the Microsoft Office automation mechanism.
- *Multiuser Engine* is an engine for managing shared graph diagrams via the subversion system.

TDA and Its Relation to the Human Brain Architecture

There is an interesting relation between TDA and the architecture of the human brain. Although we do not know how exactly our brain stores information, we know something about the structure of the brain [233]. The fact is that senses from the external world are transformed by sense organs to electrical impulses that are transferred to certain areas of the brain. That resembles how engines synchronize data between the "external" traditional code world and certain areas (represented by interface metamodels) of the model-based world. If we rearrange TDA components from Figures 3.1 on page 46 and 3.3 on page 49, we can see this more clearly (Figure 10.5).

The *thalamus* residing at the core of the brain is a "relay station" for signals coming from all the senses except smell. This resembles how TDA Kernel ensures the communication between engines and transformations by means of events and commands. The TDA analogue for smell could be a background engine that does not have its own events or commands, e.g., an autosave engine.

Transformations, with their ability to access the data from all interface metamodels and command the engines, can be associated with the *prefrontal areas* (see "thinking"

[9]The last two engines were developed by me.
[10]A part of Graph Diagram Engine Metamodel has been presented in Section 3.3.

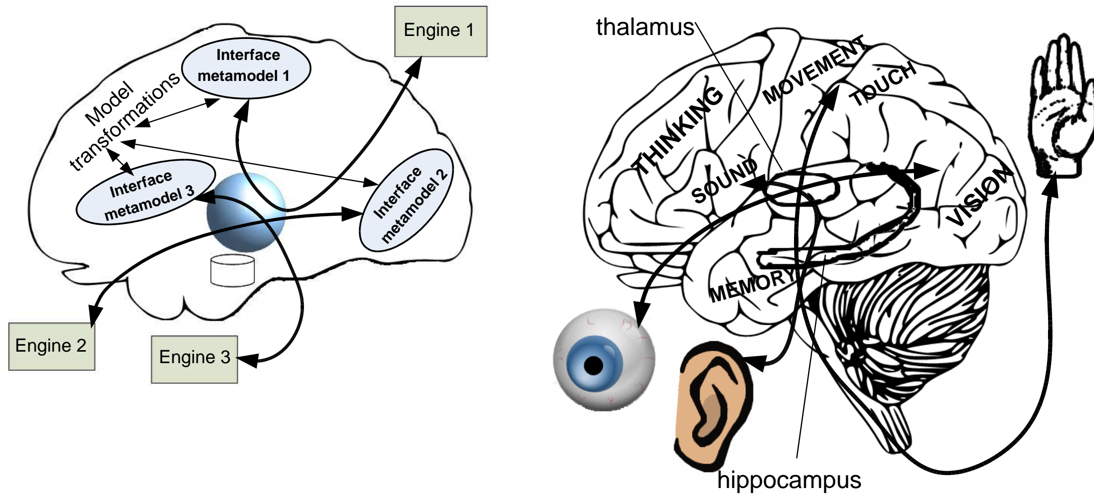


Figure 10.5: TDA and the structure of the human brain.

in Figure 10.5) that are responsible for making decisions and combining the information from different sense organs. As a result, a signal to the *motor cortex* can be given. This resembles how transformations issue commands in TDA. Like the *hippocampus*, which is responsible for forming and retrieving memory in the brain, RAAPI and the underlying repositories are responsible for memory in TDA. The *corpus callosum* (not shown in Figure 10.5), which links two cerebral hemispheres, resembles how TDA Kernel manages links between different repositories.

Conclusion

The thesis presented TDA specification and three graphical engines. Successful applications of TDA prove that system building technology, where system dynamics and business logic are described by means of model transformations working at runtime, is viable.

The following conclusions can be derived from the research:

- Formal models and their transformations are powerful artefacts that can be used to develop software systems. There are numerous technologies and standards supporting model-driven software development and model-driven engineering. On the one hand, this allows TDA to be built on the foundation of existing, well-established technologies. On the other hand, in order not to tie TDA to any particular technology, certain abstraction layers are needed. In TDA, these layers are represented by adapters for models repositories, adapters for transformations, and adapters for engines.

- By using abstraction layers as well as a universal communication mechanism, TDA can be used as a means to connect software components written in different programming languages and using different technologies. Developers are able to choose a more suitable modelling technology (technical space) and transformation language for each particular task or part of a system.

- The Model-Driven Architecture (MDA) proposed by OMG uses models and transformations at development time. The Transformation-Driven Architecture moves them to runtime. This facilitates the development of interactive systems consisting of multiple interoperating components, since models can be used to describe components of a system, while runtime transformations can be used to describe business logic.

- By hiding less important technical and platform-dependent details behind interface metamodels, TDA increases the level of abstraction, at which business logic

is described. By raising the level of abstraction and by reusing engines (some of which can be developed by third-parties), TDA facilitates more efficient software development.

- Since TDA is a common infrastructure for interactive systems, certain functionality can be implemented at the level of infrastructure for all TDA-based systems. The undo/redo mechanism and the multi-repository mechanism are examples of such functionality.

- TDA engines are a flexible means to ensure the communication between the user and TDA. They can be used to define graphical presentations (e.g., Graph Diagram Engine[11] and Dialog Engine) and provide a common way for defining and handling error messages and information messages (e.g., Error Engine). To consolidate all graphical presentations, a special engine (Environment Engine) is needed.

- The benefits offered by TDA have their price: extensive access to model repositories and the prescription to encode data in models involves certain performance slowdown. In practical applications, however, this slowdown is non-observable by a human eye.

- TDA has proven its effectiveness in developing domain specific tools. Some of them are being used in production environment in Latvia. The OWLGrEd tool is being used worldwide. The comparison of TDA with the architecture of the human brain suggests that TDA is suitable also for developing much more complex systems.

Obviously, the thesis could not address all aspects of interactive systems. That is why I concentrated on the core of such systems. The proposed core consists of TDA, TDA Kernel (implementing TDA-level functionality), and the three engines that have a potential to be included in most TDA-based systems. Having such a core, numerous interactive systems as well as meta-tools (like TDA-based meta-tool GRAF) can be built upon TDA. Since models and transformations have a potentially unlimited number of applications, there is hope that TDA could find its way to a wide range of areas. As was mentioned in the Discussion chapter, M. Zviedris, R. Liepiņš, and A. Sproģis base their research on TDA. Certain work is being done at IMCS-UL to apply TDA-based tools in medicine [234]. Like a microprocessor architecture with its instructions set and technical design is a *physical*

[11] This engine is not a contribution of this thesis, but it is described in two papers (I am a co-author) [2, 5].

foundation for software, TDA with its abstraction layers ($\sim$ instructions sets) and its design based on the notions of models, transformations, events, and commands, can be viewed as a *logical* foundation for model-driven software. When George Boole introduced his algebra (now called Boolean algebra) in 1854, he was not able to realize the impact his algebra would have on modern computers [235]. Perhaps, the potential of models and model transformations will also be realized later, when more information about our thinking processes and cognitive abilities becomes available.

Bibliography

[1] J. Barzdins, S. Kozlovics, and E. Rencis, "The Transformation-Driven Architecture," in *Proceedings of DSM'08 Workshop of OOPSLA 2008*, Nashville, Tennessee, USA, 2008, pp. 60–63.

[2] J. Barzdins, K. Cerans, S. Kozlovics, E. Rencis, and A. Zarins, "A Graph Diagram Engine for the Transformation-Driven Architecture," in *Proceedings of MDDAUI 2009 Workshop of International Conference on Intelligent User Interfaces 2009*, Sanibel Island, Florida, USA, 2009, pp. 29–32.

[3] 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," in *Proceedings of DSM'09 Workshop of OOPSLA 2009*, Florida, USA, 2009, pp. 34–40.

[4] J. Barzdins, K. Cerans, S. Kozlovics, L. Lace, R. Liepins, E. Rencis, A. Sprogis, and A. Zarins, "MDE-based Graphical Tool Building Framework," in *Scientific Papers, University of Latvia*, vol. 756, 2010, pp. 121–138.

[5] J. Barzdins, K. Cerans, S. Kozlovics, E. Rencis, and A. Zarins, "A Graph Diagram Engine for the Transformation-Driven Architecture," in *Scientific Papers, University of Latvia*, vol. 756, 2010, pp. 139–149.

[6] S. Kozlovics, "A Dialog Engine Metamodel for the Transformation-Driven Architecture," in *Scientific Papers, University of Latvia*, vol. 756, 2010, pp. 151–170.

[7] A. Sprogis, R. Liepins, J. Barzdins, K. Cerans, S. Kozlovics, L. Lace, E. Rencis, and A. Zarins, "GRAF: a graphical tool building framework," in *Proceedings of the Tools and Consultancy Track of ECMFA 2010*, S. Gerard, Ed. CEA LIST, 2010.

[8] S. Kozlovics, E. Rencis, S. Rikacovs, and K. Cerans, "Universal UNDO mechanism for the Transformation-Driven Architecture," in *Proceeding of the Ninth Interna-*

tional Baltic Conference, DB&IS 2010. Riga, Latvia: University of Latvia Press, 2010, pp. 325–340.

[9] S. Kozlovics, E. Rencis, S. Rikacovs, and K. Cerans, "A kernel-level UNDO/REDO mechanism for the Transformation-Driven Architecture," in *Proceedings of the 2011 conference on Databases and Information Systems VI: Selected Papers from the Ninth International Baltic Conference, DB&IS 2010*, ser. Frontiers in Artificial Intelligence and Applications, vol. 224. Amsterdam, The Netherlands: IOS Press, 2011, pp. 80–93.

[10] S. Kozlovics, "A universal model-based solution for describing and handling errors," in *Perspectives in Business Informatics Research*, ser. Lecture Notes in Business Information Processing, vol. 90. Springer Berlin Heidelberg, 2011, pp. 190–203.

[11] E. Rencis, J. Barzdins, and S. Kozlovics, "Towards open graphical tool-building framework," in *Scientific Journal of Riga Technical University (Special issue for the 10th International Conference on Perspectives in Business Informatics Research)*, ser. Computer Science: Applied Computer Systems, vol. 46, no. 5. Riga, Latvia: RTU Press, 2011, pp. 80–87.

[12] S. Kozlovics, "Calculating The Layout For Dialog Windows Specified As Models," in *Scientific Papers, University of Latvia*, vol. 787, 2012, pp. 106–124.

[13] S. Kozlovics, "The orchestra of multiple model repositories," in *SOFSEM 2013: Theory and Practice of Computer Science*, ser. Lecture Notes in Computer Science, vol. 7741. Springer Berlin Heidelberg, 2013, pp. 503–514.

[14] С. Козлович и Я. Барздиньш, "Управляемая трансформациями архитектура для интерактивных систем," *Автоматика и вычислительная техника*, vol. 47, no. 1/2013, pp. 39–52, 2013.

[15] S. Kozlovics and J. Barzdins, "The Transformation-Driven Architecture for interactive systems," *Automatic Control and Computer Sciences*, vol. 47, no. 1/2013, pp. 28–37, 2013, Allerton Press, Inc.

[16] G. J. Alred, C. T. Brusaw, and W. E. Oliu, *Handbook of Technical Writing, Tenth Edition.* St. Martin's Press, 2011.

[17] "Capers Jones' Software Productivity Research (SPR)," 2006.

[18] S. Kelly and J.-P. Tolvanen, *Domain-Specific Modeling: Enabling Full Code Generation.* Hoboken, New Jersey: John Wiley & Sons, Inc., 2008.

[19] P. A. Ng and R. T. Yeh, Eds., *Modern software engineering, foundations and current perspectives.* New York, NY, USA: Van Nostrand Reinhold Co., 1990.

[20] В. Д. Паронджанов, *Как улучшить работу ума: Алгоритмы без программистов — это очень просто.* М.: Дело, 2001.

[21] F. P. Brooks, Jr., “No silver bullet — essence and accidents of software engineering,” *Computer*, vol. 20, no. 4, pp. 10–19, Apr. 1987. [Online]. Available: http://dx.doi.org/10.1109/MC.1987.1663532

[22] P. Russell, *Brain Book: Know Your Own Mind and How to Use it.* Routledge & Kegan Paul PLC, 1979.

[23] J. R. Cordy, “Eating our own dog food: DSLs for generative and transformational engineering,” Joint Keynote Address at GPCE 2009, ACM 8th International Conference on Generative Programming and Component Engineering, and SLE 2009, 2nd International Conference on Software Language Engineering, Denver, Colorado, Berlin, Heidelberg, pp. 1–1, October 2009, slides are available at http://planet-sl.org/sle-conference/sle-organization/7z36rz47aezerz6er3434h/organization/sle2009/jim_cordy_sle09_keynote.pdf.

[24] “Matthew 13:31,” in *New World Translation of the Holy Scriptures.* Watch Tower Bible and Tract Society of Pennsylvania, 1984.

[25] “Psalm 139:16,” in *New World Translation of the Holy Scriptures.* Watch Tower Bible and Tract Society of Pennsylvania, 1984.

[26] E. Kalnina, “Model transformation development using MOLA mappings and Template MOLA,” Ph.D. dissertation, University of Latvia, 2011.

[27] Object Management Group. Model Driven Architecture. http://www.omg.org/mda/.

[28] J. Miller and J. Mukerji, *MDA Guide, Version 1.0.1*, Object Management Group Std. omg/03-06-01, 2003.

[29] A. G. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[30] J. Bézivin, "On the unification power of models," *Software & Systems Modeling*, vol. 4, pp. 171–188, 2005.

[31] J.-J. Dubray, "Why did MDE miss the boat? (a summary)," InfoQ news, October 27, 2011, http://www.infoq.com/news/2011/10/mde-missed-the-boat.

[32] J. Bézivin, "Why did MDE miss the boat?" in *First International Workshop on Combined Object-Oriented Modeling and Programming (COOMP 2011)*, 2011.

[33] The GradeTwo tool. http://gradetwo.lumii.lv/.

[34] J. Barzdins, G. Barzdins, K. Cerans, R. Liepins, and A. Sprogis, "OWLGrEd: a UML style graphical notation and editor for OWL 2," in *Proceedings of OWLED 2010*, 2010.

[35] G. Barzdins, E. Liepins, M. Veilande, and M. Zviedris, "Ontology enabled graphical database query tool for end-users," in *Databases and Information Systems V - Selected Papers from the Eighth International Baltic Conference, DB&IS 2008*, 2008.

[36] J. Ludewig, "Models in software engineering – an introduction," *Software and Systems Modeling*, vol. 2, pp. 5–14, 2003.

[37] J. Daintith and E. Wright, Eds., *A Dictionary of Computing.* Oxford University Press, USA, 2010.

[38] *Encyclopædia Britannica. Encyclopædia Britannica Online Academic Edition.* Encyclopædia Britannica Inc., 2012, ch. «**abstraction**», retrieved 13 June, 2012, from http://www.britannica.com/EBchecked/topic/1994/abstraction.

[39] *Information technology - Syntactic metalanguage - Extended BNF*, ISO/IEC Std. 14 977:1996(E).

[40] J. Bézivin and I. Kurtev, "Model-based technology integration with the technical space concept," in *Proceedings of the Metainformatics Symposium*, 2005.

[41] I. Kurtev, J. Bézivin, and M. Aksit, "Technological spaces: An initial appraisal," in *CoopIS, DOA 2002 Federated Conferences, Industrial track*, 2002.

[42] OMG, "OMG Unified Modeling Language (OMG UML), Infrastructure, Version 2.4.1," OMG Document Number: formal/2011-08-05, http://www.omg.org/spec/UML.

[43] *Extensible Markup Language (XML) 1.1 (Second Edition)*, http://www.w3.org/TR/xml11/, W3C Recommendation 16 August 2006, edited in place 29 September 2006.

[44] R. G. Flatscher. An overview of the architecture of EIA's CASE Data Interchange Format (CDIF). http://wi.wu-wien.ac.at/rgf/9606mobi.html.

[45] Information technology – Information Resource Dictionary System (IRDS) Services Interface (iso/iec 10728:1993 standard). http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=18821.

[46] Object Management Group, *OMG Meta Object Facility (MOF) Core Specification Version 2.4.1*, Object Management Group Std. formal/2011-08-07, 2011.

[47] Object Management Group, *MOF Support For Semantic Structures (SMOF)*, Object Management Group Std. ptc/2012-08-18, 2012, http://www.omg.org/spec/SMOF/.

[48] Eclipse Modeling Framework (EMF, Eclipse Modeling subproject). http://www.eclipse.org/emf.

[49] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework, 2nd Edition*, E. Gamma, L. Nackman, and J. Wiegand, Eds. Addison-Wesley, 2008.

[50] "Enhanced Model Repository," http://modelbased.net/aif/solutions/singular_solutions/solution_mof_repository.html.

[51] M. Matula, "NetBeans Metadata Repository," https://netbeans-uml-extender-plugin.googlecode.com/files/MDR-whitepaper.pdf.

[52] "A Meta-Modelling Technology for CMOF-based Models," http://www.webgambas.com/metabubble/amof.html.

[53] "The InfoLibrarian Universal MetaMart Metadata Repository," http://infolibcorp.com/Metadata%20Repository.html.

[54] "The CDO model repository," http://www.eclipse.org/cdo/.

[55] M. Opmanis and K. Čerāns, "Multilevel data repository for ontological and meta-modeling," in *Databases and Information Systems VI - Selected Papers from the Ninth International Baltic Conference, DB&IS 2010*, 2011.

[56] *XML Schema Definition Language (XSD) 1.1 Part 1: Structures*, http://www.w3.org/TR/xmlschema11-1/, W3C Recommendation 5 April 2012.

[57] *XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes*, http://www.w3.org/TR/xmlschema11-2/, W3C Recommendation 5 April 2012.

[58] S. Cook, G. Jones, S. Kent, and A. Wills, *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley, 2007.

[59] MetaEdit+. http://www.metacase.com.

[60] S. Kelly, "The model repository: More than just XML under version control," OOPSLA DSM Workshop 2008 keynote, 2008.

[61] *Resource Description Framework*, http://www.w3.org/RDF/, W3C (a suite of recommendations) 2004-02-10.

[62] *OWL Web Ontology Language Reference*, http://www.w3.org/TR/owl-ref/, W3C Recommendation 10 February 2004.

[63] *OWL 2 Web Ontology Language Document Overview (Second Edition)*, http://www.w3.org/TR/owl2-overview/, W3C Recommendation 11 December 2012.

[64] *OWL 2 Web Ontology Language Profiles (Second Edition)*, http://www.w3.org/TR/owl2-profiles/, W3C Std. 11 December 2012.

[65] "Sesame home page," http://www.openrdf.org/.

[66] "Virtuoso open-source edition," http://virtuoso.openlinksw.com/dataspace/dav/wiki/Main/.

[67] OWLIM semantic repository. http://www.ontotext.com/owlim/.

[68] The OWL API. http://owlapi.sourceforge.net/.

[69] Apache Jena. http://jena.apache.org/.

[70] Allegrograph RDFStore Web 3.0's database. http://www.franz.com/agraph/allegrograph/.

[71] LINQ (Language-Integrated Query). http://msdn.microsoft.com/en-us/library/vstudio/bb397926.aspx.

[72] K. Anuja, "Object relational mapping," Ph.D. dissertation, Cochin University of Science and Technology, 2007.

[73] S. Ambler. Mapping objects to relational databases: O/R mapping in detail. http://www.agiledata.org/essays/mappingObjects.html.

[74] The Java Graph Laboratory (JGraLab) project. http://www.ohloh.net/p/jgralab.

[75] Object Management Group, *OMG MOF 2 XMI Mapping Specification, Version 2.4.1*, Object Management Group Std. formal/2011-08-09, 2011.

[76] F. Jouault and J. Bezivin, "KM3: a DSL for metamodel specification," in *Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems*, 2006.

[77] The MOST project. http://www.most-project.eu.

[78] F. S. Parreiras, S. Staab, and A. Winter, "TwoUse: Integrating UML models and OWL ontologies," Universitat Koblenz-Landau, Tech. Rep. 16/2007, 2007, technical Report.

[79] Relational persistence for Java and .NET. http://www.hibernate.org/.

[80] D. Hofstadter, *Gödel, Escher, Bach: an eternal golden braid.* Harvester Press Ltd., 1979.

[81] C. Atkinson and T. Kühne, "Model-Driven Development: A metamodeling foundation," *IEEE Software*, vol. 20, no. 5, pp. 36–41, Sep. 2003.

[82] T. Kühne, "Matters of (meta-) modeling," *Software and Systems Modeling*, vol. 5, pp. 369–385, 2006.

[83] T. Kühne, "Clarifying matters of (meta-) modeling: an author's reply," *Software and Systems Modeling*, vol. 5, pp. 395–401, 2006.

[84] D. Gašević, N. Kaviani, and M. Hatala, "On metamodeling in megamodels," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, G. Engels, B. Opdyke, D. Schmidt, and F. Weil, Eds. Springer Berlin / Heidelberg, 2007, vol. 4735, pp. 91–105.

[85] C. Atkinson and T. Kühne, "Meta-level independent modelling," in *International Workshop on Model Engineering at 14th European Conference on Object-Oriented Programming*, 2000.

[86] J. Bézivin, "About model transformations," A post at Models Everywhere, November 20, 2010, http://modelseverywhere.wordpress.com/2010/11/20/about-model-transformations/.

[87] K. Czarnecki and S. Helsen, "Classification of model transformation approaches," in *Proceeding of OOPSLA03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[88] T. Mens and P. Van Gorp, "A taxonomy of model transformation," *Electron. Notes Theor. Comput. Sci.*, vol. 152, pp. 125–142, Mar. 2006. [Online]. Available: http://dx.doi.org/10.1016/j.entcs.2005.10.021

[89] T. Mens, K. Czarnecki, and P. V. Gorp, "A taxonomy of model transformations," in *Proc. Dagstuhl Seminar on "Language Engineering for Model-Driven Software Development". Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl*. Electronic, 2005.

[90] Object Management Group, *MOF 2.0 Query / Views / Transformations RFP*, Object Management Group Request for Proposal ad/2002-04-10, 2002.

[91] Object Management Group, *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.0*, Object Management Group Std. formal/2008-04-03, 2008.

[92] Object Management Group, *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1*, Object Management Group Std. formal/2011-01-01, 2011.

[93] "The Model To Model (M2M) project, a subproject of Eclipse Modeling Project," http://www.eclipse.org/m2m/.

[94] "The *smartqvt* project," http://sourceforge.net/projects/smartqvt/.

[95] IKV++ Technologies AG, "The *medini qvt* project," http://projects.ikv.de/qvt/.

[96] F. Allilaire and T. Idrissi, "ADT: Eclipse development tools for ATL," in *Proceedings of Second European Workshop on Model Driven Architecture*, 2004.

[97] F. Jouault and I. Kurtev, "Transforming models with ATL," in *Proceedings of the 2005 international conference on Satellite Events at the MoDELS*, ser. MoDELS'05. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 128–138.

[98] "ATL — a model transformation technology, an Eclipse Model-to-Model Transformation project," http://www.eclipse.org/atl/.

[99] D. Kolovos, L. Rose, and R. Paige, "The Epsilon Book," http://www.eclipse.org/epsilon/doc/book/.

[100] "Epsilon, an Eclipse Model-to-Model Transformation project," http://www.eclipse.org/epsilon/.

[101] A. Kalnins, J. Barzdins, and E. Celms, "Model transformation language MOLA," in *Model Driven Architecture*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, vol. 3599, pp. 62–76.

[102] Research Laboratory of Modeling and Software Technologies of Institute of Mathematics and Computer Science, University of Latvia, "The MOLA project," http://mola.mii.lu.lv/.

[103] J. Barzdins, A. Kalnins, E. Rencis, and S. Rikacovs, "Model transformation languages and their implementation by bootstrapping method," in *Pillars of computer science*, A. Avron, N. Dershowitz, and A. Rabinovich, Eds. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 130–145. [Online]. Available: http://dl.acm.org/citation.cfm?id=1805839.1805847

[104] I. of Mathematics and U. o. L. Computer Science, "The Lx transformation language set home page," http://lx.mii.lu.lv/.

[105] R. Liepiņš, “Library for model querying: IQuery,” in *Proceedings of the 12th Workshop on OCL and Textual Modelling*, ser. OCL ’12. New York, NY, USA: ACM, 2012, pp. 31–36.

[106] S. Demathieu, C. Griffin, and S. Sendall, “Model transformation with the IBM Model Transformation Framework,” An article at IBM developerWorks, May, 3 2005, http://www.ibm.com/developerworks/rational/library/05/503_sebas/.

[107] A. Agrawal, G. Karsai, and F. Shi, “Graph transformations on domain-specific models,” Institute for Software Integrated Systems, Vanderbilt University, Tech. Rep., 2003, http://w3.isis.vanderbilt.edu/publications/archive/agrawal_a_11_0_2003_graph_tran.pdf.

[108] Institute for Software Integrated Systems, Vanderbilt University, “The graph rewrite and transformation (GReAT) tool suite,” http://repo.isis.vanderbilt.edu/tools/get_tool?GReAT.

[109] T. Horn and J. Ebert, “The GReTL transformation language,” in *Proceedings of the 4th international conference on Theory and practice of model transformations*, ser. ICMT’11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 183–197.

[110] M. Lawley and J. Steel, “Practical declarative model transformation with Tefkat,” in *Proceedings of the 2005 international conference on Satellite Events at the MoDELS*, ser. MoDELS’05. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 139–150.

[111] “Tefkat: The EMF Transformation Engine,” http://tefkat.sourceforge.net/.

[112] D. Varró and A. Balogh, “The model transformation language of the VIATRA2 framework,” *Sci. Comput. Program.*, vol. 68, no. 3, pp. 187–207, Oct. 2007. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2007.05.004

[113] VIATRA2 (VIsual Automated model TRAnsformations) framework, an Eclipse project. http://eclipse.org/viatra2/.

[114] J. M. Almendros-Jiménez and L. Iribarne, “A model transformation language based on logic programming,” in *SOFSEM 2013: Theory and Practice of Computer Science*, ser. Lecture Notes in Computer Science, P. Emde Boas, F. Groen, G. Italiano, J. Nawrocki, and H. Sack, Eds. Springer Berlin Heidelberg, 2013, vol. 7741, pp. 382–394.

[115] P. Braun and F. Marschall, "BOTL: the bidirectional object oriented transformation language," Institut für Informatik der Technischen Universität Munchen, Tech. Rep., 2003.

[116] "The BOTL (Bidirectional Object-oriented Transformation Language) project," http://botl.sourceforge.net/.

[117] J. Cuadrado, J. Molina, and M. Tortosa, "RubyTL: A practical, extensible transformation language," in *Model Driven Architecture – Foundations and Applications*, ser. Lecture Notes in Computer Science, A. Rensink and J. Warmer, Eds. Springer Berlin / Heidelberg, 2006, vol. 4066, pp. 158–172.

[118] "The Agile Generative Environment (AGE) project," http://gts.inf.um.es/trac/age.

[119] M. Herrmannsdoerfer, S. Benz, and E. Juergens, "COPE – automating coupled evolution of metamodels and models," in *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, ser. Genoa. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 52–76.

[120] L. M. Rose, D. S. Kolovos, R. F. Paige, and F. A. C. Polack, "Model migration with Epsilon Flock," in *Proceedings of the Third international conference on Theory and practice of model transformations*, ser. ICMT'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 184–198. [Online]. Available: http://dl.acm.org/citation.cfm?id=1875847.1875862

[121] G. Taentzer, "AGG: a tool environment for algebraic graph transformation," in *Proceedings of the International Workshop on Applications of Graph Transformations with Industrial Relevance*, ser. AGTIVE '99. London, UK, UK: Springer-Verlag, 2000, pp. 481–488. [Online]. Available: http://dl.acm.org/citation.cfm?id=646676.702002

[122] "The attributed graph grammar system: A development environment for attributed graph transformation systems," http://user.cs.tu-berlin.de/\protect\unhbox\voidb@x\penalty\@M\gragra/agg/.

[123] A. Schürr, A. J. Winter, and A. Zündorf, "The PROGRES approach: language and environment," in *Handbook of graph grammars and computing by graph transformation*, H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, Eds.

River Edge, NJ, USA: World Scientific Publishing Co., Inc., 1999, ch. The PROGRES approach: language and environment, pp. 487–550. [Online]. Available: http://dl.acm.org/citation.cfm?id=328523.328617

[124] A. Schürr, "Specification of graph translators with triple graph grammars," in *WG*, ser. Lecture Notes in Computer Science, E. W. Mayr, G. Schmidt, and G. Tinhofer, Eds., vol. 903. Springer, 1994, pp. 151–163.

[125] T. Fischer, J. Niere, L. Torunski, and A. Zündorf, "Story diagrams: A new graph rewrite language based on the Unified Modeling Language and Java," in *Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations*, ser. TAGT'98. London, UK, UK: Springer-Verlag, 2000, pp. 296–309. [Online]. Available: http://dl.acm.org/citation.cfm?id=645872.668867

[126] Fujaba Development Group - University of Paderborn, "Incremental Model Transformation and Synchronization with Triple Graph Grammars," 2006, http://www.fujaba.de/projects/triple-graph-grammars.html.

[127] 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 *Proceedings of the 17th IEEE international conference on Automated software engineering*, ser. ASE '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 267–. [Online]. Available: http://dl.acm.org/citation.cfm?id=786769.787011

[128] E. Jakumeit, S. Buchwald, and M. Kroll, "Grgen.net," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 12, no. 3, pp. 263–271, Jul. 2010.

[129] The GrGen.NET project. http://www.info.uni-karlsruhe.de/software/grgen/.

[130] *XSL Transformations (XSLT), Version 1.0*, http://www.w3.org/TR/xslt, W3C Recommendation 16 November 1999.

[131] *XSL Transformations (XSLT), Version 3.0 (a working draft)*, http://www.w3.org/TR/xslt-30/, W3C Working Draft 10 July 2012.

[132] C. Bizer and A. Seaborne, "D2RQ – treating non-RDF databases as virtual RDF graphs," in *Proceedings of the 3rd International Semantic Web Conference (ISWC 2004)*, 2004.

[133] "D2RQ: Accessing relational databases as virtual RDF graphs," FU Berlin, DERI, UCB, JP Morgan Chase, AGFA Healthcare, HP Labs, Johannes Kepler Universität Linz, http://d2rq.org/.

[134] W3C, "R2RML: RDB to RDF mapping language (W3C working draft)," Souripriya Das, Seema Sundara, Richard Cyganiak, eds., 29 May, 2012, http://www.w3.org/TR/r2rml/.

[135] C. Bizer, "D2R MAP - database to RDF mapping language and processor," Institut für Produktion, Wirtschaftsinformatik und OR, http://www4.wiwiss.fu-berlin.de/bizer/d2rmap/d2rmap.htm.

[136] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin, "On the use of higher-order model transformations," in *Model Driven Architecture – Foundations and Applications*, ser. Lecture Notes in Computer Science, R. Paige, A. Hartman, and A. Rensink, Eds. Springer Berlin / Heidelberg, 2009, vol. 5562, pp. 18–33.

[137] *XML Path Language (XPath), Version 1.0*, http://www.w3.org/TR/xpath/, W3C Recommendation 16 November 1999.

[138] Object Management Group, *Object Constraint Language (OCL), Version 2.3.1*, http://www.omg.org/spec/OCL/, Object Management Group Std., 2012, oMG Document Number: formal/2012-01-01.

[139] "Epsilon Validation Language," http://www.eclipse.org/epsilon/doc/evl/.

[140] F. Corno and L. Farinetti, "Logic and reasoning in the semantic web (part II – OWL)," Materials for the «01LHVIU - Semantic Web: Technologies, Tools, Applications» course at *Politecnico di Torino, Dipartimento di Automatica e Informatica*, 2012, http://elite.polito.it/files/courses/01LHV/2012/7-OWLreasoning.pdf.

[141] S. Mazzocchi, "Closed world vs. open world: the first semantic web battle," A blog at Stefano's Linotype, June 16, 2005, http://www.betaversion.org/\protect\unhbox\voidb@x\penalty\@M\stefano/linotype/news/91/.

[142] M. Zakharyaschev, "Reasoning with OWL. Open vs closed worlds. Constructors." Materials for the «Semantic Web» course, http://www.dcs.bbk.ac.uk/\protect\unhbox\voidb@x\penalty\@M\michael/sw/slides/Sew11-6.pdf.

[143] J. Cabot, "Clarifying concepts: MBE vs MDE vs MDD vs MDA," Post at MOdeling LAnguages, http://modeling-languages.com/clarifying-concepts-mbe-vs-mde-vs-mdd-vs-mda/.

[144] D. Ameller, "Considering non-functional requirements in model-driven engineering," Master's thesis, Universitat Politècnica de Catalunya, 2009.

[145] J. Bézivin, "What is MDE?" A post at Models Everywhere, October 31, 2010, http://modelseverywhere.wordpress.com/2010/10/31/what-is-mde/.

[146] J. Bézivin, "Broadening application area," A post at Models Everywhere, November 5, 2010, http://modelseverywhere.wordpress.com/2010/11/05/broadening-application-area/.

[147] M. Völter, "MD*/DSL Best Practices, Version 2.0," April 2011, http://www.voelter.de/data/pub/DSLBestPractices-2011Update.pdf.

[148] J. Siegel, "Making the case: OMG's Model Driven Architecture," http://www.sdtimes.com/content/article.aspx?ArticleID=26807&page=1, October 15 2002, SD Times.

[149] OMG ADM Task Force, "Why do we need standards for the modernization of existing systems? (adm whitepaper)," http://adm.omg.org/legacy/ADM_whitepaper.pdf.

[150] G. Barbier, H. Bruneliere, F. Jouault, Y. Lennon, and F. Madiot, *Information Systems Transformation: Architecture-Driven Modernization Case Studies.* The Morgan Kaufmann/OMG Press, 2010, ch. MoDisco, a Model-Driven Platform to Support Real Legacy Modernization Use Cases, pp. 365–400.

[151] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot, "Modisco: a generic and extensible framework for model driven reverse engineering," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 173–174. [Online]. Available: http://doi.acm.org/10.1145/1858996.1859032

[152] "The MoDisco project," http://www.eclipse.org/MoDisco/.

[153] J. Osis, E. Asnina, and A. Grave, "Computation Independent Representation of the Problem Domain in MDA," *e-Informatica Software Engineering Journal*, vol. 2, no. 1, pp. 29–46, 2008.

[154] J. Osis and E. Asnina, Eds., *Model-Driven Domain Analysis and Software Development: Architectures and Functions*. IGI Global, 2011.

[155] Object Management Group, "MDA success stories," http://www.omg.org/mda/products_success.htm.

[156] A. Sostaks and A. Kalnins, "The implementation of MOLA to L3 compiler," *Scientific Papers University of Latvia*, vol. 733, pp. 140–178, 2008.

[157] S. W. Ambler, "Are you ready for the MDA?" http://www.agilemodeling.com/essays/readyForMDA.htm.

[158] D. Thomas, "MDA: revenge of the modelers or UML utopia?" *Software, IEEE*, vol. 21, no. 3, pp. 15–17, 2004.

[159] M. Fowler, "ModelDrivenArchitecture," http://martinfowler.com/bliki/ModelDrivenArchitecture.html, 2 February 2004.

[160] D. Haywood, "MDA: Nice idea, shame about the...," http://www.theserverside.com/news/1365166/MDA-Nice-idea-shame-about-the, May 1 2004.

[161] A. reviewer at the International Conference on Model Transformation 2011, "Paper 47, review 2 (provided by Elina Kalnina)."

[162] S. W. Amber, "Agile Modeling (AM) home page: Effective practices for modeling and documentation," http://www.agilemodeling.com/.

[163] S. Ambler, "Agile model driven development is good enough," *Software, IEEE*, vol. 20, no. 5, pp. 71–73, 2003.

[164] S. Cook, "Domain-Specific Modeling and Model Driven Architecture," *MDA Journal*, pp. 2–10, 2004.

[165] B. Muglia, "Microsoft focuses on bringing modeling mainstream, improves IT delivery of business strategies," http://www.microsoft.com/en-us/news/press/2008/sep08/09-10OMGModelingPR.aspx, September 10 2008, microsoft News Center.

[166] I. Jacobson and S. Cook, "The road ahead for UML," http://www.drdobbs.com/architecture-and-design/224701702?pgno=1, May 12 2010.

[167] W. W. Eckerson, "Three Tier Client/Server Architecture: Achieving Scalability, Performance, and Efficiency in Client Server Applications," in *Open Information Systems 10*, Jan. 1995.

[168] J. Bentley, "Programming pearls: little languages," *Commun. ACM*, vol. 29, pp. 711–721, August 1986. [Online]. Available: http://doi.acm.org/10.1145/6424.315691

[169] *SPARQL Query Language for RDF*, http://www.w3.org/TR/rdf-sparql-query/, W3C Recommendation 15 January 2008.

[170] S. C. Johnson, "Yacc: Yet Another Compiler-Compiler," AT&T Bell Laboratories, Tech. Rep., 1975.

[171] Object Management Group, *Business Process Model and Notation (BPMN), Version 2.0*, http://www.bpmn.org/, Object Management Group Std. formal/2011-01-03, 2011.

[172] Eclipse Graphical Modeling Project (GMP). http://www.eclipse.org/gmf.

[173] A. Shatalin and A. Tikhomirov, "Graphical Modeling Framework architecture overview," in *Eclipse Modeling Symposium*, 2006.

[174] Graphical Editor Framework (GEF, Eclipse Tools subproject). http://www.eclipse.org/gef.

[175] J. Ebert, R. Suttenbach, and I. Uhe, "Meta-CASE in practice: a case for KOGGE," in *Proceedings of the 9th International Conference, CAiSE'97*, 1997.

[176] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi, "The generic modeling environment," in *Workshop on Intelligent Signal Processing*, 2001.

[177] "GME: Generic Modeling Environment," http://www.isis.vanderbilt.edu/projects/gme/.

[178] DiaGen/DiaMeta. http://www.unibw.de/inf2/DiaGen/.

[179] N. Zhu, J. Grundy, J. Hosking, N. Liu, S. Cao, and A. Mehra, "Pounamu: A meta-tool for exploratory domain-specific visual language tool development," *Journal of Systems and Software*, vol. 80, no. 8, pp. 1390–1407, 2007.

[180] DoME (Domain Modeling Environment). http://dome-continued.sourceforge.net/.

[181] C. Ermel, K. Ehrig, G. Taentzer, and E. Weiss, "Object oriented and rule-based design of visual languages using Tiger," in *Proceedings of 3rd International Workshop on Graph Based Tool*, vol. 1, 2006.

[182] J. de Lara, H. Vangheluwe, and M. Alfonseca, "Meta-modelling and graph grammars for multi-paradigm modelling in ATOM3," *Software and System Modeling*, vol. 3, pp. 194–209, 2004.

[183] A. Kalnins, O. Vilitis, E. Celms, E. Kalnina, A. Sostaks, and J. Barzdins, "Building tools by model transformations in Eclipse," in *Proceedings of DSM'07 Workshop of OOPSLA 2007.* Montreal, Canada: Jyvaskyla University Printing House, 2007, pp. 194–207.

[184] C. Amelunxen, A. Konigs, T. Rotschke, and A. Schurr, "MOFLON: A standard-compliant metamodeling framework with graph transformations," in *Model Driven Architecture - Foundations and Applications: Second European Conference, volume 4066 of Lecture Notes in Computer Science (LNCS.* Springer Verlag, Springer Verlag, 2006, pp. 361–375.

[185] J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez, "The AMMA platform support for modeling in the large and modeling in the small," LINA, Universite de Nantes, Tech. Rep., 2005.

[186] A. Haase, M. Volter, S. Efftinge, and B. Kolb, "Introduction to openArchitectureWare 4.1.2," in *Model-Driven Development Tool Implementers Forum (co-located with TOOLS 2007)*, 2007.

[187] J. Bézivin, F. Jouault, and P. Valduriez, "On the Need for Megamodels," in *Proceedings of Workshop on Best Practices for Model-Driven Software Development at the 19th OOPSLA*, Vancouver, British Columbia, Canada, 2004.

[188] Я.Я. Бичевский, А.К. Зариньш, and А.А. Калниньш, "Инструкция по работе с системой макрокоманд обработки данных (СМОД). Часть 1. Основные операторы." Ассоциация пользователей ЭВМ типа "Минск", Москва, 1973.

[189] Я.Я. Бичевский and М.В. Витиньш, "Инструкция по работе с системой макрокоманд обработки данных (СМОД). Часть 2. Дополнительные команды." Ассоциация пользователей ЭВМ типа "Минск", Москва, 1973.

[190] "History of SDL," http://www.sdl-forum.org/sdl2000present/tsld002.htm.

[191] "GRADE History," http://www.infologistik.com/papers/grade_history.shtml.

[192] J. Barzdins, A. Zarins, K. Cerans, A. Kalnins, E. Rencis, L. Lace, R. Liepins, and A. Sprogis, "GrTP: Transformation based graphical tool building platform," in *Proceedings of MDDAUI 2007 Workshop of MODELS 2007*, Nashville, Tennessee, USA, 2007.

[193] J. Barzdins, G. Barzdins, R. Balodis, K. Cerans, A. Kalnins, M. Opmanis, and K. Podnieks, "Towards Semantic Latvia," in *Proceedings of Seventh International Baltic Conference on Databases and Information Systems, Communications, Vilnius, Lithuania*, O. Vasileckas, J. Eder, and A. Caplinskas, Eds., 2006, pp. 203–218.

[194] P. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems.* Addison Wesley Publishing Company, 1987. [Online]. Available: http://www.amazon.com/Concurrency-Control-Recovery-Database-Systems/dp/0201107155%3FSubscriptionId%3D0JYN1NVW651KCA56C102%26tag%3Dtechkie-20%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D0201107155

[195] RFC 2396 - Uniform Resource Identifiers (URI): Generic Syntax. http://www.ietf.org/rfc/rfc2396.txt. IETF.

[196] C. Hein, T. Ritter, and M. Wagner, "Model-driven tool integration with ModelBus," in *Proceedings of Future Trends of Model-Driven Development Workshop*, 2009.

[197] "ModelBus homepage," http://www.modelbus.org/modelbus/.

[198] "Object Database Management Systems," http://www.odbms.org/.

[199] I. Ráth, G. Bergmann, A. Ökrös, and D. Varró, "Live model transformations driven by incremental pattern matching," in *Proceedings of the 1st international conference on Theory and Practice of Model Transformations*, ser. ICMT '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 107–121.

[200] A. Warth, Y. Ohshima, T. Kaehler, and A. Kay, "Worlds: Controlling the scope of side effects," Viewpoints Research Institute, Tech. Rep., 2010.

[201] E. Rencis, "On views on metamodels," in *Databases and Information Systems VI - Selected Papers from the Ninth International Baltic Conference, DB&IS 2010*, 2011.

[202] "Eclipse – The Eclipse Foundation open source community website," http://www.eclipse.org/.

[203] D. English, "To undo or not to undo," http://blog.jodoro.com/2009/01/to-undo-or-not-to-undo.html, January 20, 2009.

[204] S. Rikacovs, "Towards a seed transformation language and its implementation," in *Doctoral Symposium at MODELS 2008*, 2008.

[205] T. Hartmann and D. A. Sadilek, "Undoing operational steps of domain-specific modeling languages," in *Proceedings of DSM'08 Workshop of OOPSLA 2008*, 2008.

[206] I. Groher and A. Egyed, "Selective and consistent undoing of model changes," in *Proceedings of the 13th international conference on Model driven engineering languages and systems: Part II*, ser. MODELS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 123–137. [Online]. Available: http://dl.acm.org/citation.cfm?id=1929101.1929115

[207] S. Liang, *Java Native Interface: Programmer's Guide and Specification*, L. Friendly, Ed. Addison-Wesley, 1999.

[208] K. Freivalds and P. Ķikusts, "Optimum layout adjustment supporting ordering constraints in graph-like diagram drawing," in *Proceedings of the Latvian Academy of Sciences, Section B*, vol. 55, no. 1, 2001, pp. 43–51.

[209] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, 2nd ed. MIT Press, Cambridge, MA, U.S.A., 2001.

[210] R. Tarjan, "Shortest paths," AT&T Bell Laboratories, Murray Hill, NJ, Tech. Rep., 1981.

[211] "Glade — a user interface designer," http://glade.gnome.org.

[212] M. Abrams and E. Helms, J., *User Interface Markup Language (UIML) specification. Working draft 3.1.*, http://www.oasis-open.org/committees/download.php/5937/uiml-core-3.1-draft-01-20040311.pdf, OASIS Open Std.

[213] "UsiXML: USer Interface eXtensible Markup Language," http://www.usixml.org.

[214] T. Pavek, "Get to know GroupLayout," in *NetBeans Magazine*, no. 1, pp. 58–66, http://netbeans.org/download/magazine/01/nb01_group_layout.pdf.

[215] "Laying out components within a container," http://docs.oracle.com/javase/tutorial/uiswing/layout/.

[216] "Windows Presentation Foundation," http://msdn.microsoft.com/en-us/library/ms754130.aspx.

[217] C. Lutteroth and G. Weber, "User interface layout with ordinal and linear constraints," in *AUIC*, ser. CRPIT, W. Piekarski, Ed., vol. 50. Australian Computer Society, 2006, pp. 53–60.

[218] "Ajax: A new approach to web applications," http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications, February 18, 2005.

[219] "Google Web Toolkit homepage," https://developers.google.com/web-toolkit/.

[220] "The official Microsoft Silverlight site," http://www.silverlight.net/.

[221] "Adobe Flash," http://www.adobe.com/products/flash.

[222] "W3C: The Forms Working Group," http://www.w3.org/MarkUp/Forms/.

[223] G. J. Badros, A. Borning, and P. J. Stuckey, "The cassowary linear arithmetic constraint solving algorithm," *ACM Trans. Comput.-Hum. Interact.*, vol. 8, no. 4, pp. 267–306, Dec. 2001. [Online]. Available: http://doi.acm.org/10.1145/504704.504705

[224] A. Borning, K. Marriott, P. Stuckey, and Y. Xiao, "Solving linear arithmetic constraints for user interface applications," in *Proceedings of the 10th annual ACM symposium on User interface software and technology*, ser. UIST '97. New York, NY, USA: ACM, 1997, pp. 87–96. [Online]. Available: http://doi.acm.org/10.1145/263407.263518

[225] H. Hosobe, "A modular geometric constraint solver for user interface applications," in *Proceedings of the 14th annual ACM symposium on User interface software and technology*, ser. UIST '01. New York, NY, USA: ACM, 2001, pp. 91–100. [Online]. Available: http://doi.acm.org/10.1145/502348.502362

[226] "QT Developer Network," http://qt-project.org/.

[227] Java Platform, Standard Edition 7 API Specification. http://docs.oracle.com/javase/7/docs/api/.

[228] .NET Framework Developer Center. http://msdn.microsoft.com/netframework. http://msdn.microsoft.com/netframework.

[229] J. Richter, *Programming Applications for Microsoft Windows*, 4th ed. Microsoft Press, 1999.

[230] M. Brambilla, S. Ceri, S. Comai, and C. Tziviskou, "Exception handling in workflow-driven web applications," in *Proceedings of International Conference on World Wide Web*. ACM Press, 2005, pp. 170–179.

[231] S. D. Fraser, F. P. Brooks, Jr., M. Fowler, R. Lopez, A. Namioka, L. Northrop, D. L. Parnas, and D. Thomas, ""no silver bullet" reloaded: retrospective on "essence and accidents of software engineering"," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, ser. OOPSLA '07. New York, NY, USA: ACM, 2007, pp. 1026–1030.

[232] F. Marinescu and A. Avram, "No Silver Bullet Reloaded Retrospective OOPSLA Panel Summary," InfoQ article, October 21, 2008, http://www.infoq.com/articles/No-Silver-Bullet-Summary.

[233] R. Carter, S. Aldridge, M. Page, and S. Parker, *The Human Brain Book*, S. Hirani, K. John, and R. Warren, Eds. Dorling Kindersley Ltd., 2009.

[234] J. Barzdins, J. Barzdins, E. Rencis, and A. Sostaks, "Modeling and query language for hospitals," in *Health Information Science*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, vol. 7798, pp. 113–124.

[235] G. Boole, *An Investigation of the Laws of Thought: On Which Are Founded the Mathematical Theories of Logic and Probabilities (Classic Reprint).* Forgotten Books, 2012 [1854].

Appendix A

Repository Access Application Programming Interface (RAAPI) Documentation

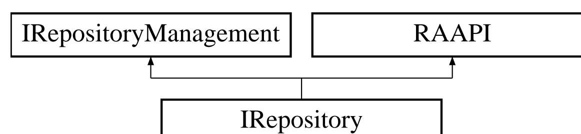
The documentation below has been generated by Doxygen from the RAAPI source code developed by me (S. Kozlovičs). The source code license is as follows:


```
Copyright (C) 2010-2012 by University of Latvia
Copyright (C) 2012-2013 by the Institute of Mathematics and Computer Science,
                           University of Latvia

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to
deal in the Software without restriction, including without limitation the
rights to use, copy, modify, merge, publish, distribute, sublicense, and/or
sell copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:
The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS
IN THE SOFTWARE.
```


The up-to-date version of RAAPI can be found at `tda.lumii.lv/raapi.html`.

A.1 IRepository Interface Reference

Inheritance diagram for IRepository:

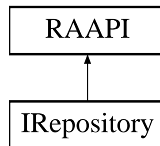


Detailed Description

The `IRepository` interface is a union of `RAAPI` containing operations on model elements and `IRepositoryManagement` containing technical operations on repositories such as operations for opening, closing, saving, etc.

A.2 RAAPI Interface Reference

Inheritance diagram for RAAPI:



Public Member Functions

Operations on primitive data types

- Reference findPrimitiveDataType (in UnicodeString name)
- UnicodeString getPrimitiveDataTypeName (in Reference rDataType)
- boolean isPrimitiveDataType (in Reference r)

Operations on classes

- Reference createClass (in UnicodeString name)
- Reference findClass (in UnicodeString name)
- UnicodeString getClassName (in Reference rClass)
- boolean deleteClass (in Reference rClass)
- Iterator getIteratorForClasses ()
- boolean createGeneralization (in Reference rSubClass, in Reference rSuperClass)
- boolean deleteGeneralization (in Reference rSubClass, in Reference rSuperClass)
- Iterator getIteratorForDirectSuperClasses (in Reference rSubClass)
- Iterator getIteratorForDirectSubClasses (in Reference rSuperClass)
- boolean isClass (in Reference r)
- boolean isDirectSubClass (in Reference rSubClass, in Reference rSuperClass)
- boolean isDerivedClass (in Reference rDirectlyOrIndirectlyDerivedClass, in Reference rSuperClass)

Operations on objects

- Reference createObject (in Reference rClass)
- boolean deleteObject (in Reference rObject)
- boolean includeObjectInClass (in Reference rObject, in Reference rClass)
- boolean excludeObjectFromClass (in Reference rObject, in Reference rClass)
- boolean moveObject (in Reference rObject, in Reference rToClass)
- Iterator getIteratorForAllClassObjects (in Reference rClassOrAdvancedAssociation)
- Iterator getIteratorForDirectClassObjects (in Reference rClassOrAdvancedAssociation)
- Iterator getIteratorForDirectObjectClasses (in Reference rObjectOrAdvancedLink)

- boolean isTypeOf (in Reference rObject, in Reference rClass)
- boolean isKindOf (in Reference rObject, in Reference rClass)

Operations on attributes

- Reference createAttribute (in Reference rClass, in UnicodeString name, in Reference rPrimitiveType)
- Reference findAttribute (in Reference rClass, in UnicodeString name)
- boolean deleteAttribute (in Reference rAttribute)
- Iterator getIteratorForAllAttributes (in Reference rClass)
- Iterator getIteratorForDirectAttributes (in Reference rClass)
- UnicodeString getAttributeName (in Reference rAttribute)
- Reference getAttributeDomain (in Reference rAttribute)
- Reference getAttributeType (in Reference rAttribute)
- boolean isAttribute (in Reference r)

Operations on attribute values

- boolean setAttributeValue (in Reference rObject, in Reference rAttribute, in UnicodeString value)
- UnicodeString getAttributeValue (in Reference rObject, in Reference rAttribute)
- boolean deleteAttributeValue (in Reference rObject, in Reference rAttribute)
- Iterator getIteratorForObjectsByAttributeValue (in Reference rAttribute, in UnicodeString value)

Operations on associations

- Reference createAssociation (in Reference rSourceClass, in Reference rTargetClass, in UnicodeString sourceRoleName, in UnicodeString targetRoleName, in boolean isComposition)
- Reference createDirectedAssociation (in Reference rSourceClass, in Reference r-TargetClass, in UnicodeString targetRoleName, in boolean isComposition)
- Reference createAdvancedAssociation (in UnicodeString name, in boolean nAry, in boolean associationClass)
- Reference findAssociationEnd (in Reference rSourceClass, in UnicodeString targetRoleName)
- boolean deleteAssociation (in Reference rAssociationEndOrAdvancedAssociation)
- Iterator getIteratorForAllOutgoingAssociationEnds (in Reference rClass)
- Iterator getIteratorForDirectOutgoingAssociationEnds (in Reference rClass)
- Iterator getIteratorForAllIngoingAssociationEnds (in Reference rClass)
- Iterator getIteratorForDirectIngoingAssociationEnds (in Reference rClass)
- Reference getInverseAssociationEnd (in Reference rAssociationEnd)
- Reference getSourceClass (in Reference rTargetAssociationEnd)
- Reference getTargetClass (in Reference rTargetAssociationEnd)
- UnicodeString getRoleName (in Reference rAssociationEnd)
- boolean isComposition (in Reference rTargetAssociationEnd)
- boolean isAdvancedAssociation (in Reference r)
- boolean isAssociationEnd (in Reference r)

Operations on links

- boolean createLink (in Reference rSourceObject, in Reference rTargetObject, in Reference rAssociationEnd)
- boolean createOrderedLink (in Reference rSourceObject, in Reference rTargetObject, in Reference rAssociationEnd, in long targetPosition)

- boolean deleteLink (in Reference rSourceObject, in Reference rTargetObject, in Reference rAssociationEnd)
- boolean linkExists (in Reference rSourceObject, in Reference rTargetObject, in Reference rAssociationEnd)
- Iterator getIteratorForLinkedObjects (in Reference rObject, in Reference r-AssociationEnd)
- long getLinkedObjectPosition (in Reference rSourceObject, in Reference rTarget-Object, in Reference rAssociationEnd)

Operations with iterators

- Reference resolveIteratorFirst (in Iterator it)
- Reference resolveIteratorNext (in Iterator it)
- long getIteratorLength (in Iterator it)
- Reference resolveIterator (in Iterator it, in long position)
- void freeIterator (in Iterator it)

Operations on references

- void freeReference (in Reference r)
- UnicodeString serializeReference (in Reference r)
- Reference deserializeReference (in UnicodeString r)

Operations with quasi-linguistic meta-metamodel at Level M3

- Iterator getIteratorForLinguisticClasses ()
- Iterator getIteratorForDirectLinguisticInstances (in Reference rClass)
- Iterator getIteratorForAllLinguisticInstances (in Reference rClass)
- Reference getLinguisticClassFor (in Reference r)
- boolean isLinguistic (in Reference r)

Calling repository-specific operations

- UnicodeString callSpecificOperation (in UnicodeString operationName, in UnicodeString arguments)

Detailed Description

`RAAPI` is a common abstraction layer for accessing models stored in different repositories associated with different technical spaces.

Member Function Documentation

Reference **RAAPI::findPrimitiveDataType (** in UnicodeString ***name*** **)**

Obtains a reference to a primitive data type with the given name.

Parameters

name	the type name. Each repository must support at least four standard primitive data types: "Integer", "Real", "Boolean", and "String". Certain repositories may introduce additional primitive types. To denote a repository-specific additional primitive data type, prepend the mount point of that repository, e.g., `MountPoint::PeculiarDataType`.

Returns

a reference to a primitive data type with the given name, or 0 on error.
Note (TDA Kernel): TDA Kernel returns a proxy reference, which is usable even when there are multiple repositories mounted or non-standard primitive data types are used.

UnicodeString **RAAPI::getPrimitiveDataTypeName (** in Reference ***rDataType*** **)**

Returns the name of the given primitive data type.

Parameters

rDataType	a reference to a primitive data type, for which the name has to be obtained

Returns

the name of the given primitive data type, or `null` on error.

See Also

RAAPI::findPrimitiveDataType

boolean RAAPI::isPrimitiveDataType (in Reference ***r*** **)**

Checks whether the given reference is associated with a primitive data type.

Parameters

r	a reference in question

Returns

whether the given reference is associated with a primitive data type. On error, `false` is returned.

See Also

RAAPI::findPrimitiveDataType
RAAPI::getPrimitiveDataTypeName

Reference **RAAPI::createClass (** in UnicodeString ***name*** **)**

Creates a class with the given fully qualified name.

Parameters

name	the fully qualified name of the class (packages are delimited by double colon "::"); this fully qualified name must be unique

Returns

a reference to the class just created, or 0 on error.

Reference **RAAPI::findClass (** in UnicodeString ***name*** **)**

Obtains a reference to an existing class with the given fully qualified name.
Note (M3): If the underlying repository provides access to its quasi-linguistic meta-metamodel, quasi-linguistic classes can be accessed by appending the package name "M3" to the mount point of that repository, e.g., `SomePath::MountPoint::M3::SomeMetaType`
Note (adapters): This function is optional for repository adapters. If not implemented in an adapter, TDA Kernel implements it through `getIteratorForClasses` and `getIteratorForLinguisticClasses`.

Parameters

name	the fully qualified name of the class (for quasi-linguistic classes, append package "M3" to the mount point of the corresponding repository)

Returns

a reference to a (quasi-ontological or quasi-linguistic) class with the given fully qualified name.

UnicodeString **RAAPI::getClassName (in** Reference ***rClass*** **)**

Returns the fully qualified name of the given class.
Note (M3): If the reference points to a quasi-linguistic class, then the package name "M3" is also included in the return value, e.g., `MountPointForTheCorresponding-Repository::M3::ClassName`.

Parameters

rClass	a reference to the class, for which the class name has to be obtained

Returns

the fully qualified name of the given class, or `null` on error.

boolean RAAPI::deleteClass (in Reference ***rClass*** **)**

Deletes the class and frees the reference.

Parameters

rClass	a reference to the class to be deleted

Returns

whether the operation succeeded.

Iterator **RAAPI::getIteratorForClasses ( )**

Obtains an iterator for all classes (all quasi-ontological classes at all quasi-ontological meta-levels).
Note (M3): Linguistic classes are not traversed by this iterator. Use `getIteratorFor-LinguisticClasses` instead.

Returns

an iterator for all classes, or 0 on error.

See Also

RAAPI::getIteratorForLinguisticClasses

boolean RAAPI::createGeneralization (in Reference ***rSubClass,*** **in** Reference ***rSuperClass*** **)**

Creates a generalization between the two given classes.
The given subclass can be a derived class of the given superclass, but the direct generalization between them must not exist.
The generalization relation being created must not introduce inheritance loops.

Parameters

rSubClass	a class that becomes a subclass
rSuperClass	a class that becomes a superclass

Returns
whether the operation succeeded.

boolean RAAPI::deleteGeneralization (in Reference *rSubClass,* in Reference *rSuperClass*)

Deletes the generalization between the given two classes.

Parameters

rSubClass	a class that was a subclass
rSuperClass	a class that was a superclass

Returns
whether the operation succeeded.

Iterator RAAPI::getIteratorForDirectSuperClasses (in Reference *rSubClass*)

Obtains an iterator for all direct superclasses of the given subclass.
Note (M3): If the given subclass is a quasi-linguistic class, then an iterator for its direct quasi-linguistic superclasses is returned.

Parameters

rSubClass	a subclass for which to obtain direct superclasses

Returns
an iterator for all direct superclasses of the given subclass, or 0 on error.

Iterator RAAPI::getIteratorForDirectSubClasses (in Reference *rSuperClass*)

Obtains an iterator for all direct subclasses of the given superclass.
Note (M3): If the given superclass is a quasi-linguistic class, then an iterator for direct quasi-linguistic subclasses is returned.

Parameters

rSuperClass	a superclass for which to obtain direct subclasses

Returns
an iterator for all direct subclasses of the given superclass, or 0 on error.

boolean RAAPI::isClass (in Reference *r*)

Checks whether the given reference is associated with a class.
Note (M3): A reference at Level M3 can also be passed.

Parameters

r	a reference in question

Returns
whether the given reference is associated with a class. On error, `false` is returned.

boolean RAAPI::isDirectSubClass (in Reference *rSubClass,* in Reference *rSuperClass*)

Checks whether the generalization relation between the two given classes holds.
Note (M3): Both classes may be either quasi-ontological, or quasi-linguistic.

Parameters

rSubClass	a reference to a potential subclass
rSuperClass	a reference to a potential superclass

Returns
whether the generalization relation holds. On error, `false` is returned.

boolean RAAPI::isDerivedClass (in Reference ***rDirectlyOrIndirectlyDerivedClass,* in** Reference ***rSuperClass*)**

Checks whether one class is a direct or indirect subclass of another.
Note (M3): Both classes may be either quasi-ontological, or quasi-linguistic.

Parameters

rDirectlyOr-Indirectly-DerivedClass	a reference to a potential subclass or derived class
rSuperClass	a reference to a potential (direct or indirect) superclass

Returns
whether the first class derives from the second. On error, `false` is returned.

Reference **RAAPI::createObject (in** Reference ***rClass*)**

Creates an instance of the given class.
Note (M3): If the given class is a quasi-linguistic class, then its quasi-linguistic instance at Level MΩ is being created.

Parameters

rClass	a reference to a class (either quasi-ontological, or quasi-linguistic)

Returns
whether the operation succeeded.

boolean RAAPI::deleteObject (in Reference ***rObject*)**

Creates the given object.

Parameters

rObject	a reference to the object to be deleted

Returns
whether the operation succeeded.

boolean RAAPI::includeObjectInClass (in Reference ***rObject,* in** Reference ***rClass*)**

Adds the given object to the given (quasi-ontological) class. The function works, if the underlying repository supports multiple classification and dynamic reclassification.
Note (M3): It is assumed that an element from a quasi-ontological level can be associated with only one quasi-linguistic type (quasi-linguistic class), thus, `includeObjectInClass` is meaningless in this case.

Parameters

rObject	a reference to the object to be included in the given class
rClass	a reference to the class, where to put the object (in addition to classes, where the object already belongs)

Returns
whether the operation succeeded.

boolean RAAPI::excludeObjectFromClass (in Reference *rObject,* in Reference *rClass*)

Takes out the given object from the given (quasi-ontological) class.
The function works, if the underlying repository supports multiple classification and dynamic reclassification. If the object currently is only in one class, then the operation fails (it is assumed that each object must be at least in one class).
Note (M3): It is assumed that an element from a quasi-ontological level can be associated with only one quasi-linguistic type (quasi-linguistic class), thus, `excludeObjectInClass` as well as `includeObjectInClass` are meaningless in this case.

Parameters

rObject	a reference to the object to be excluded from the given class
rClass	a reference to the class, which to exclude from the classifiers of the given object

Returns
whether the operation succeeded.

boolean RAAPI::moveObject (in Reference *rObject,* in Reference *rToClass*)

Moves (reclassifies) the given object into the given (quasi-ontological) class, removing it from its current class (classes).
The function is similar to calling
`includeObjectInClass(rObject, rToClass);` followed by calling
`excludeObjectInClass(rObject, c)` for all other current classifiers *c* of the given object.
The distinction is that it may be possible to implement this function even when multiple classification is not supported.
Note (adapters): This function is optional for repository adapters. If not implemented in an adapter, TDA Kernel implements it by recreating the object (with the new type), while also recreating all attributes and links.
Note (M3): It is assumed that an element from a quasi-ontological level cannot dynamically change its quasi-linguistic type (quasi-linguistic class), thus, `moveObject` is meaningless in this case.

Parameters

rObject	a reference to the object to be reclassified
rToClass	a reference to the class, to which the object will belong

Returns
whether the operation succeeded.

Iterator RAAPI::getIteratorForAllClassObjects (in Reference *rClassOrAdvancedAssociation*)

Obtains an iterator for all quasi-ontological instances of the given class or advanced association.
Note (adapters): A repository adapter may implement only one of the functions `getIteratorForAllClassObjects` and `getIteratorForDirectClassObjects`. The unimplemented function will be implemented via another by TDA Kernel.
Note (M3): If the given class or advanced association is quasi-linguistic, then an iterator for the quasi-linguistic elements it describes is returned, e.g., for the EMOF class "Class",

an iterator for all classes found in EMOF is returned; for the EMOF class "Property", an iterator for all properties found in EMOF is returned, etc.

Parameters

rClassOrAdvancedAssociation	a reference to a class or an advanced association

Returns

an iterator for all quasi-ontological instances (objects) of the given class or advanced association. On error, 0 is returned.

See Also

RAAPI::getIteratorForDirectClassObjects

Iterator **RAAPI::getIteratorForDirectClassObjects** (**in** Reference ***rClassOrAdvancedAssociation***)

Obtains an iterator for direct quasi-ontological instances of the given class or advanced association.
Note (adapters): A repository adapter may implement only one of the functions `getIteratorForAllClassObjects` and `getIteratorForDirectClassObjects`. The unimplemented function will be implemented via another by TDA Kernel.
Note (M3): If the given class or advanced association is quasi-linguistic, then an iterator for the quasi-linguistic elements it describes is returned, e.g., for the EMOF class "Class", an iterator for all classes found in EMOF is returned; for the EMOF class "Property", an iterator for all properties found in EMOF is returned, etc.

Parameters

rClassOrAdvancedAssociation	a reference to a class or an advanced association

Returns

an iterator for direct quasi-ontological instances (objects) of the given class or advanced association. On error, 0 is returned.

See Also

RAAPI::getIteratorForAllClassObjects

Iterator **RAAPI::getIteratorForDirectObjectClasses** (**in** Reference ***rObjectOrAdvancedLink***)

Obtains an iterator for direct quasi-ontological classes of the given object or advanced link.
Note (M3): The function works also if the given object or advanced link is quasi-linguistic and the underlying repository provides access to quasi-linguistic elements. To get the quasi-linguistic class for the given element at some quasi-ontological level, use `getLinguisticClassFor`.

Parameters

rObjectOrAdvancedLink	a reference to an object or advanced link

Returns
an iterator for direct quasi-ontological classes of the given object or advanced link. On error, 0 is returned.

See Also
RAAPI::getLinguisticClassFor

boolean RAAPI::isTypeOf (in Reference *rObject*, in Reference *rClass*)

Checks whether the given object is a direct (quasi-ontological or quasi-linguistic) instance of the given class.
Note (M3): The function works also when one or both of `rObject` and `rClass` is/are quasi-linguistic. If the object is at a quasi-ontological meta-level, but the class is quasi-linguistic, then the function checks whether the object is a direct quasi-linguistic instance of the given class.

Parameters

rObject	a reference to an object
rClass	a reference to a class

Returns
whether the given object is a direct instance of the given class. On error, `false` is returned.

See Also
RAAPI::isKindOf

boolean RAAPI::isKindOf (in Reference *rObject*, in Reference *rClass*)

Checks whether the given object is a direct or indirect, quasi-ontological or quasi-linguistic, instance of the given class.
Note (M3): The function works also when one or both of `rObject` and `rClass` is/are quasi-linguistic. If the object is at a quasi-ontological meta-level, but the class is quasi-linguistic, then the function checks whether the object is a quasi-linguistic instance of the given class or one of its subclasses.

Parameters

rObject	a reference to an object
rClass	a reference to a class

Returns
whether the given object is a (direct or indirect) instance of the given class. On error, `false` is returned.

See Also
RAAPI::isTypeOf

Reference RAAPI::createAttribute (in Reference *rClass*, in UnicodeString *name*, in Reference *rPrimitiveType*)

Creates (defines) a new attribute for the given class. The default cardinality is the widest cardinality supported by the repository (e.g., "0..*", if multi-valued attributes are supported; or "0..1", otherwise). The cardinality can be looked up and changed by using the quasi-linguistic meta-metalevel.

Parameters

rClass	a reference to an existing class, for which to define the attribute
name	the name of the attribute being created; it must be unique within all the attributes defined for this class, including derived ones
rPrimitive-Type	a reference to a primitive data type for attribute values

Returns

a reference to the attribute just created, or 0 on error.

Reference **RAAPI::findAttribute (** in Reference ***rClass,*** in UnicodeString ***name*** **)**

Obtains a reference to an existing attribute with the given name of the given class.
Note (M3): The class reference may point also to a quasi-linguistic class.

Parameters

rClass	a reference to a class, where the attribute in question belongs; `rClass` may be also one of its subclasses, since the attribute is available for subclasses, too
name	the name of the attribute

Returns

a reference to the desired attribute, or 0 on error; the reference returned is the same reference for the class, for which the attribute was defined, as well as for derived classes.

boolean RAAPI::deleteAttribute (in Reference ***rAttribute*** **)**

Deletes the given attribute.

Parameters

rAttribute	a reference to the attribute to be deleted

Returns

whether the operation succeeded.

Iterator **RAAPI::getIteratorForAllAttributes (** in Reference ***rClass*** **)**

Obtains an iterator for all (including inherited) attributes of the given class.
Note (adapters): A repository adapter may implement only one of the functions `getIteratorForAllAttributes` and `getIteratorForDirectAttributes`. The unimplemented function will be implemented via another by TDA Kernel.
Note (M3): The function works also for quasi-linguistic classes.

Parameters

rClass	a reference to a class, whose attributes we are interested in

Returns

an iterator for all attributes (including inherited) of the given class. On error, 0 is returned.

See Also

RAAPI::getIteratorForDirectAttributes

Iterator **RAAPI::getIteratorForDirectAttributes (** in Reference ***rClass*** **)**

Obtains an iterator for direct (without inherited) attributes of the given class.

Note (adapters): A repository adapter may implement only one of the functions `getIteratorForAllAttributes` and `getIteratorForDirectAttributes`. The unimplemented function will be implemented via another by TDA Kernel.
Note (M3): The function works also for quasi-linguistic classes.

Parameters

rClass	a reference to a class, whose attributes we are interested in

Returns
an iterator for direct attributes of the given class. On error, 0 is returned.

See Also
RAAPI::getIteratorForAllAttributes

UnicodeString **RAAPI::getAttributeName (** **in** Reference ***rAttribute*** **)**

Returns the name of the given attribute.
Note (M3): The function works also for attributes of quasi-linguistic classes.

Parameters

rAttribute	a reference to the attribute in question

Returns
the name of the given attribute, or `null` on error.

Reference **RAAPI::getAttributeDomain (** **in** Reference ***rAttribute*** **)**

Obtains a class, for which the given attribute was defined.
Note (M3): The function works also for attributes of quasi-linguistic classes.

Parameters

rAttribute	a reference to the attribute in question

Returns
a reference to a class, for which the given attribute belongs, or 0 on error.

Reference **RAAPI::getAttributeType (** **in** Reference ***rAttribute*** **)**

Returns the (primitive) type for values of the given attribute.
Note (M3): The function works also for attributes of quasi-linguistic classes.

Parameters

rAttribute	a reference to the attribute in question

Returns
a reference to a primitive data type for values of the given attribute.

boolean RAAPI::isAttribute (**in** Reference ***r*** **)**

Checks whether the given reference is associated with an attribute.
Note (adapters): If a repository adapter does not implement this function, TDA Kernel will implement it by means of `getAttributeDomain`, `getAttributeName` and `findAttribute`.
Note (M3): A reference at Level M3 can also be passed.

Parameters

r	a reference in question

Returns

whether the given reference is associated with an attribute. On error, `false` is returned.

boolean RAAPI::setAttributeValue (in Reference *rObject,* in Reference *rAttribute,* in UnicodeString *value*)

Sets the value or the ordered collection of values (encoded as a string) of the given attribute for the given object.
Note (adapters): Repository adapters may assume that the value is not `null` and not a string encoding `null`, since for those cases TDA Kernel forwards the call to `delete-AttributeValue`.
Note (M3): The attribute reference can be a reference at the M3 level. In this case the object can be any element at the MΩ level.

Parameters

rObject	the object, for which to set the attribute value (values)
rAttribute	the attribute, for which to set the value; this attribute must be associated either with a quasi-ontological class or the quasi-linguistic class of the given object
value	the attribute value (values) encoded as a string

Returns

whether the value(s) has (have) been set. On error, `false` is returned.

UnicodeString RAAPI::getAttributeValue (in Reference *rObject,* in Reference *rAttribute*)

Gets the value or the ordered collection of values (encoded as a string) of the given attribute for the given object.
Note (M3): The attribute reference can be a reference at the M3 level.

Parameters

rObject	the object, for which to get the attribute value (values)
rAttribute	the attribute, for which to obtain the value; this attribute must be associated either with a quasi-ontological class or the quasi-linguistic class of the given object

Returns

the attribute value (values) encoded as a string, or `null` on error.

boolean RAAPI::deleteAttributeValue (in Reference *rObject,* in Reference *rAttribute*)

Deletes the value (all the values) of the given attribute for the given object.
Note (M3): The attribute reference can be a reference at the M3 level. In this case the object can be any element at the MΩ level.

Parameters

rObject	the object, for which to get the attribute value (values)
rAttribute	the attribute, for which to obtain the value; this attribute must be associated either with a quasi-ontological class or the quasi-linguistic class of the given object

Returns
whether the operation succeeded.

Iterator **RAAPI::getIteratorForObjectsByAttributeValue (in** Reference ***rAttribute*, in** UnicodeString ***value*)**

Obtains an iterator for objects, for whose the value of the given attribute equals to the given value. The value has to be encoded as a string (it may encode an ordered collection of multiple values).
Note (M3): The attribute reference can be a reference at the M3 level. In this case the objects traversed by the returned iterator are elements at the MΩ level.

Parameters

rAttribute	the attribute to check
value	the value to check

Returns
the iterator for objects with the given attribute value, or 0 on error.

Reference **RAAPI::createAssociation (in** Reference ***rSourceClass*, in** Reference ***rTargetClass*, in** UnicodeString ***sourceRoleName*, in** UnicodeString ***targetRoleName*, in boolean *isComposition*)**

Creates a bidirectional association (or two directed associations, where each is an inverse of the other). The default value for the source and target cardinalities should be "*".
Note (M3): The M3 level can be used to get/set the cardinality, if the repository supports constraints and the M3 level operations. Cardinality constraints must be accessible via M3 for that.

Parameters

rSourceClass	the class, where the association starts
rTargetClass	the class, where the association ends
sourceRoleName	the name of the association end near the source class
targetRoleName	the name of the association end near the target class
isComposition	whether the association is a composition, i.e., the source class objects are containers for the target class objects

Returns
a reference for the target association end of the association just created, or 0 on error.

Reference **RAAPI::createDirectedAssociation (in** Reference ***rSourceClass*, in** Reference ***rTargetClass*, in** UnicodeString ***targetRoleName*, in boolean *isComposition*)**

Creates a directed association. The default value for the source and target cardinalities should be "*".
Note (adapters): If a repository adapter does not implement this function, TDA kernel will simulate it by means of `createAssociation` (a stub inverse role will be generated).
Note (M3): The M3 level can be used to get/set the cardinality, if the repository supports constraints and the M3 level operations. Cardinality constraints must be accessible via M3 for that.

Parameters

rSourceClass	the class, where the association starts
rTargetClass	the class, where the association ends
targetRoleName	the name of the association end near the target class
isComposition	whether the association is a composition, i.e., the source class objects are containers for the target class objects

Returns

a reference for the target association end of the association just created, or 0 on error.

Reference **RAAPI::createAdvancedAssociation (** **in** UnicodeString ***name,*** **in boolean** ***nAry,*** **in boolean** ***associationClass*** **)**

Creates an n-ary association, an association class, or an n-ary association class.
An advanced association behaves likes a class (although it might not be a class internally) with n bidirectional associations attached to it. To specify all n association ends, call `createAssociation` n times, where a reference to the n-ary association has to be passed instead of one of the class references. N-ary association links can be created by means of `createObject`, and n-ary link ends can be created by calling `createLink` n times and passing a reference to the n-ary link instead of one of the object references.
Note (adapters): The underlying repository is allowed to create an n-ary association class, even when nAry or associationClass is `false`.
Note (adapters): If a repository adapter does not implement this function, TDA kernel will implement this function by introducing an additional class.
Note (M3): The M3 level can be used to get/set the cardinality, if the repository supports constraints and the M3 level operations. Cardinality constraints must be accessible via M3 for that.

Parameters

name	the name of the advanced association (the class name in case of an association class)
nAry	whether the association is an n-ary association
associationClass	whether the association is an association class

Returns

a reference to the n-ary association just created (not the association end, since no association ends are created yet), or 0 on error.

Reference **RAAPI::findAssociationEnd (** **in** Reference ***rSourceClass,*** **in** UnicodeString ***targetRoleName*** **)**

Obtains a reference to an association end (by its role name) starting at the given class.
Note (adapters): If not implemented in the adapter, TDA kernel will implement it by means of `getIteratorForAllOutgoingAssociationEnds`.
Note (M3): The function works also, when searching for association ends at Level M3.

Parameters

rSourceClass	a class that is a source class for the association, or one of its subclasses
targetRoleName	a role name associated with the target association end

Returns
a reference to an association end corresponding to the given target role name.

boolean RAAPI::deleteAssociation (in Reference *rAssociationEndOrAdvancedAssociation*)

Deletes the given association. Directed and bidirectional associations are specified by (one of) their ends. Advanced associations have their own references. If the association is bidirectional, the inverse association end is deleted as well. For advanced associations, all association parts are deleted.

Parameters

rAssociationEndOrAdvancedAssociation	a reference to an association end (if the association is directed or bidirectional) or a reference to an advanced association

Returns
whether the operation succeeded.

Iterator RAAPI::getIteratorForAllOutgoingAssociationEnds (in Reference *rClass*)

Obtains an iterator for all (including inherited) outgoing association ends of the given class.
Note (adapters): A repository adapter may implement only one of the functions `getIteratorForAllOutgoingAssociationEnds` and `getIteratorForDirectOutgoingAssociationEnds`. The unimplemented function will be implemented via another by TDA Kernel.
Note (M3): The function works also for associations at Level M3.

Parameters

rClass	a reference to a class, whose outgoing associations (including inherited) have to be traversed

Returns
an iterator for all (including inherited) outgoing association ends of the given class. On error, 0 is returned.

See Also
RAAPI::getIteratorForDirectOutgoingAssociationEnds

Iterator RAAPI::getIteratorForDirectOutgoingAssociationEnds (in Reference *rClass*)

Obtains an iterator for direct (without inherited) outgoing association ends of the given class.
Note (adapters): A repository adapter may implement only one of the functions `getIteratorForAllOutgoingAssociationEnds` and `getIteratorForDirectOutgoingAssociationEnds`. The unimplemented function will be implemented via another by TDA Kernel.
Note (M3): The function works also for associations at Level M3.

Parameters

rClass	a reference to a class, whose direct outgoing associations have to be traversed

Returns

an iterator for direct (without inherited) outgoing association ends of the given class. On error, 0 is returned.

See Also

RAAPI::getIteratorForAllOutgoingAssociationEnds

Iterator **RAAPI::getIteratorForAllIngoingAssociationEnds (** in Reference ***rClass*** **)**

Obtains an iterator for all (including inherited) ingoing association ends of the given class.
Note (adapters): A repository adapter may implement only one of the functions `getIteratorForAllIngoingAssociationEnds` and `getIteratorForDirectIngoingAssociationEnds`. The unimplemented function will be implemented via another by TDA Kernel.
Note (M3): The function works also for associations at Level M3.

Parameters

rClass	a reference to a class, whose ingoing associations (including inherited) have to be traversed

Returns

an iterator for all (including inherited) ingoing association ends of the given class. On error, 0 is returned.

See Also

RAAPI::getIteratorForDirectIngoingAssociationEnds

Iterator **RAAPI::getIteratorForDirectIngoingAssociationEnds (** in Reference ***rClass*** **)**

Obtains an iterator for direct (without inherited) ingoing association ends of the given class.
Note (adapters): A repository adapter may implement only one of the functions `getIteratorForAllIngoingAssociationEnds` and `getIteratorForDirectIngoingAssociationEnds`. The unimplemented function will be implemented via another by TDA Kernel.
Note (M3): The function works also for associations at Level M3.

Parameters

rClass	a reference to a class, whose direct ingoing associations have to be traversed

Returns

an iterator for direct (without inherited) ingoing association ends of the given class. On error, 0 is returned.

See Also

RAAPI::getIteratorForAllIngoingAssociationEnds

Reference **RAAPI::getInverseAssociationEnd (** in Reference ***rAssociationEnd*** **)**

Obtains a reference to the inverse association end of the given association end (if association is bidirectional or a bidirectional part of an advanced association).
Note (M3): The function works also for association ends at Level M3.

Parameters

rAssociationEnd	a reference to a known association end, for which the inverse end has to be obtained

Returns
a reference to the inverse association end. On error or if the association end does not have the inverse, 0 is returned.

Reference **RAAPI::getSourceClass (** in Reference ***rTargetAssociationEnd*** **)**

Obtains a reference to the source class of the given directed or bidirectional association (or part of an advanced association) specified by its target end. Any of the association ends can be considered a target end, when calling this function.

Parameters

rTarget-Association-End	an association end of some association; this association end will be considered a target end

Returns
a reference to the source class of the given association specified by its target end.

Reference **RAAPI::getTargetClass (** in Reference ***rTargetAssociationEnd*** **)**

Obtains a reference to the class corresponding to the given association end of some directed, bidirectional, or advanced association. For bidirectional and advanced associations, any of the two association ends can be considered a target end, when calling this function.

Parameters

rTarget-Association-End	an association end of some association; this association end will be considered a target end

Returns
a reference to the class corresponding to the given association end.

UnicodeString **RAAPI::getRoleName (** in Reference ***rAssociationEnd*** **)**

Returns the role name of the given association end.

Parameters

rAssociation-End	an association end of some directed, bidirectional, or advanced association

Returns
the role name of the given association end, or `null` on error.

boolean RAAPI::isComposition (in Reference ***rTargetAssociationEnd*** **)**

Returns, whether the directed or bidirectional association (or a part of an advanced association) specified by its target association end is a composition (i.e., whether the source class objects are containers for the target class objects).
Note (M3): A reference at Level M3 can also be passed.

Parameters

rTarget-Association-End	an association end of some association; this association end will be considered a target end

Returns

whether the directed or bidirectional association (or a part of an advanced association) is a composition.

boolean RAAPI::isAdvancedAssociation (in Reference *r*)

Checks, whether the given reference corresponds to an advanced association.
Note (M3): A reference at Level M3 can also be passed.

Parameters

r	a reference in question

Returns

whether the given reference corresponds to an advanced association. On error, `false` is returned.

boolean RAAPI::isAssociationEnd (in Reference *r*)

Checks, whether the given reference corresponds to an association end.
Note (adapters): If not implemented in a repository adapter, TDA Kernel will implement it by means of `getSourceClass`, `getRoleName` and `findAssociationEnd`.
Note (M3): A reference at Level M3 can also be passed.

Parameters

r	a reference in question

Returns

whether the given reference corresponds to an association end. On error, `false` is returned.

boolean RAAPI::createLink (in Reference *rSourceObject,* in Reference *rTargetObject,* in Reference *rAssociationEnd*)

Creates a link of the given type (specified by `rAssociationEnd`) between two objects.
Note (M3): An association end at Level M3 can also be passed. In this case, at least one of the source and target objects must be an element at the MΩ level. The semantics of such link then depends on a particular quasi-linguistic metamodel at Level M3.

Parameters

rSourceObject	a start object of the link; this object must be an instance of the source class for the given association end
rTargetObject	an end object of the link; this object must be an instance of the target class for the given association end
rAssociationEnd	a target association end that specifies the link type

Returns

whether the operation succeeded.

boolean RAAPI::createOrderedLink (in Reference *rSourceObject,* in Reference *rTargetObject,* in Reference *rAssociationEnd,* in long *targetPosition*)

Creates a link of the given type (specified by `rAssociationEnd`) between two objects at the given position. The target position normally should be from 0 to n, where n is the

number of currently linked objects at positions from 0 to n-1. If the target position is outside [0..n], then the link is appended to the end.
Note (M3): An association end at Level M3 can also be passed. In this case, at least one of the source and target objects must be an element at the MΩ level. The semantics of such link then depends on a particular quasi-linguistic metamodel at Level M3.

Parameters

rSourceObject	a start object of the link; this object must be an instance of the source class for the given association end
rTargetObject	an end object of the link; this object must be an instance of the target class for the given association end
rAssociationEnd	a target association end that specifies the link type
targetPosition	the position (starting from 0) of the target object in the list of linked objects of the source object;

Returns
whether the operation succeeded.

boolean RAAPI::deleteLink (in Reference ***rSourceObject,*** **in** Reference ***rTargetObject,*** **in** Reference ***rAssociationEnd*** **)**

Deletes a link of the given type (specified by `rTargetAssociationEnd`) between the given two objects.
Note (M3): An association end at Level M3 can also be passed. In this case, at least one of the source and target objects must be an element at the MΩ level. The semantics of such link then depends on a particular quasi-linguistic metamodel at Level M3.

Parameters

rSourceObject	a start object of the link; this object must be an instance of the source class for the given association end
rTargetObject	an end object of the link; this object must be an instance of the target class for the given association end
rAssociationEnd	a target association end that specifies the link type

Returns
whether the operation succeeded.

boolean RAAPI::linkExists (in Reference ***rSourceObject,*** **in** Reference ***rTargetObject,*** **in** Reference ***rAssociationEnd*** **)**

Checks whether the link of the given type (specified by `rTargetAssociationEnd`) between the given two objects exists.
Note (adapters): If not implemented in a repository adapter, TDA Kernel will implement this function through `getIteratorForLinkedObjects`.
Note (M3): An association end at Level M3 can also be passed. In this case, at least one of the source and target objects must be an element at the MΩ level. The semantics of such link then depends on a particular quasi-linguistic metamodel at Level M3.

Parameters

rSourceObject	a start object of the link; this object must be an instance of the source class for the given association end
rTargetObject	an end object of the link; this object must be an instance of the target class for the given association end
rAssociationEnd	a target association end that specifies the link type

Returns

whether the link exists. On error, `false` is returned.

Iterator **RAAPI::getIteratorForLinkedObjects (in** Reference ***rObject*, in** Reference ***rAssociationEnd*)**

Returns an iterator for objects linked to the given start object by links of the given type.
Note (M3): The type of links may also be an association end at Level M3.

Parameters

rObject	a start object, for which the iterable objects are linked this object must be an instance of the source class for the given association end
rAssociationEnd	a target association end that specifies the type of links

Returns

an iterator for objects, linked to the given object by links of the given type, or 0 on error.

long RAAPI::getLinkedObjectPosition (in Reference ***rSourceObject*, in** Reference ***rTargetObject*, in** Reference ***rAssociationEnd*)**

Returns the index (numeration starts from 0) of the target object in the list of objects linked to the source object by links of the given type.
Note (M3): The type of links may also be an association end at Level M3.

Parameters

rSourceObject	a source object; this object must be an instance of the source class for the given association end
rTargetObject	a target object; this object must be an instance of the target class for the given association end
rAssociationEnd	a target association end that specifies the type of links

Returns

the index (numeration starts from 0) of the given target object in the list of objects linked to the source object. On error or when the source and target objects are not linked by the given association, -1 is returned.

Reference **RAAPI::resolveIteratorFirst (in** Iterator ***it*)**

Places the iterator to the position 0 and gets the element there.

Parameters

it	an iterator reference

Returns

the element at position 0 in the iterable list. If there are no elements or if an error occurred, 0 is returned.

See Also

RAAPI::resolveIteratorNext
RAAPI::freeIterator

Reference **RAAPI::resolveIteratorNext (** in Iterator ***it*** **)**

Moves the iterator forward and gets the element at that position.

Parameters

it	an iterator reference

Returns

the element the iterator points to, after the iterator has been moved one step forward. If there are no elements or if an error occurred, 0 is returned.

See Also

RAAPI::resolveIteratorFirst
RAAPI::freeIterator

long **RAAPI::getIteratorLength (** in Iterator ***it*** **)**

Places the iterator to the position 0 and returns the total number of elements to iterate through. Call `resolveIteratorFirst` or `resolveIterator` to move the iterator.
Note (adapters): If not implemented in a repository adapter, TDA Kernel traverses all the elements and stores them in a temporary list. Thus, the first call will take the linear execution time, while all subsequent calls will take the constant time. The same refers to the `resolveIterator` function. If both `getIteratorLength` and `resolveIterator` are used, the temporary list is created only once.

Parameters

it	an iterator reference

Returns

the total number of elements to iterate through. On error returns 0 (thus, the return value still represents the number of iterations, which can be performed with this iterator).

See Also

RAAPI::resolveIterator
RAAPI::freeIterator

Reference **RAAPI::resolveIterator (** in Iterator ***it,*** in long ***position*** **)**

Returns a reference to the element at the given `position` (numeration starts from 0) and forwards the iterator to `position+1`.
Note (adapters): If not implemented in a repository adapter, TDA Kernel traverses all the elements and stores them in a temporary list. Thus, the first call will take the linear execution time, while all subsequent calls will take the constant time. The same refers to the `getIteratorLength` function. If both `getIteratorLength` and `resolveIterator` are used, the temporary list is created only once.

Parameters

it	an iterator reference
position	the position in the iterable list, where the interested element is located

Returns
a reference to the element at the given position, or 0 if the position is out of bounds, or if an error occurred.

See Also
RAAPI::getIteratorLength
RAAPI::freeIterator

void RAAPI::freeIterator (in Iterator *it*)

Frees the memory associated with the given iterator reference.

Parameters

it	an iterator reference

void RAAPI::freeReference (in Reference *r*)

Decrements the counter of the given reference. When the counter reaches 0, frees the memory associated with the given reference (if necessary).

UnicodeString RAAPI::serializeReference (in Reference *r*)

Creates a string representation of the given reference, which survives the current session. For the next session, TDA kernel will use this string to get another reference to the same element by means of `deserializeReference`. This is essential for storing inter-repository relations.

Parameters

r	the reference to serialize

Returns
a string representation of the given reference, which survives the current session, or `null` on error.

See Also
RAAPI::deserializeReference

Reference RAAPI::deserializeReference (in UnicodeString *r*)

Obtains a reference to a serialized element from the given serialization. This is essential for loading inter-repository relations.

Parameters

r	the serialization of an element, for which to obtain a reference

Returns
a reference corresponding for the given serialization, or 0 on error.

See Also
RAAPI::serializeReference

Iterator RAAPI::getIteratorForLinguisticClasses ( )

Returns an iterator for all quasi-linguistic classes at Level M3.
Note (M3): This function works only at Level M3.

Returns
an iterator for all quasi-linguistic classes at Level M3.

Iterator **RAAPI::getIteratorForDirectLinguisticInstances (** **in** Reference ***rClass*** **)**

Returns an iterator for direct quasi-linguistic instances (not including instances of subclasses) at Level MΩ of the given class at Level M3.
Note (M3): This function takes a class at Level M3 and returns an iterator for elements at Level MΩ.

Parameters

rClass	a Level M3 class

Returns

an iterator for direct quasi-linguistic instances at Level MΩ of the given class at Level M3, or 0 on error.

Iterator **RAAPI::getIteratorForAllLinguisticInstances (** **in** Reference ***rClass*** **)**

Returns an iterator for all quasi-linguistic instances (including instances of subclasses) at Level MΩ of the given class at Level M3.
Note (M3): This function takes a class at Level M3 and returns an iterator for elements at Level MΩ.

Parameters

rClass	a Level M3 class

Returns

an iterator for direct quasi-linguistic instances at Level MΩ of the given class (and its subclasses) at Level M3, or 0 on error.

Reference **RAAPI::getLinguisticClassFor (** **in** Reference ***r*** **)**

Returns a reference to the Level M3 class of the given quasi-ontological (Level MΩ) element. It is assumed that there may be at most one quasi-linguistic class at M3 for each quasi-ontological element at MΩ.

Parameters

r	a quasi-ontological (Level MΩ) element

Returns

a reference to the Level M3 class of the given quasi-ontological (Level MΩ) element. On error or if M3 is not supported by the underlying repository, 0 is returned.

boolean RAAPI::isLinguistic (**in** Reference ***r*** **)**

Checks, whether the given reference is associated with a Level M3 element. Can be used together with `isClass`, `isAssociationEnd`, etc. to get more details about the element.

Parameters

r	a reference in question

Returns

whether the given reference is associated with a Level M3 element. On error, `false` is returned.

UnicodeString **RAAPI::callSpecificOperation (** **in** UnicodeString ***operationName,*** **in** UnicodeString ***arguments*** **)**

Calls a repository-specific operation (e.g., or MOF/ECore-like operation, an SQL statement, or a SPARQL statement). Arguments (if any) are encoded as a string delimited by means of the Unicode character U+001E (INFORMATION SEPARATOR TWO). For no-argument methods `arguments` must be `null`.
For instance, a repository may accept the following calls:

```
callSpecificOperation("SQL", "SELECT * FROM MY TABLE");
callSpecificOperation("myMethod", "<object-reference>\u001-
E<argument1>...");
callSpecificOperation("", null);
```

For static MOF/ECore-like operations, the first argument should point to a class. For non-static operations the first argument should point to an object (that resembles `this` pointer in Java).

Parameters

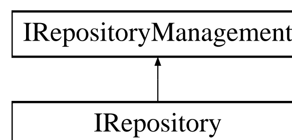
operationName	a repository-specific operation name
arguments	operation-specific arguments encoded as a string

Returns

the return value of the call encoded as a string, or `null` on error.

A.3 IRepositoryManagement Interface Reference

Inheritance diagram for IRepositoryManagement:



Public Member Functions

- boolean exists (in UnicodeString location)
- boolean open (in UnicodeString location)
- void close ()
- boolean startSave ()
- boolean finishSave ()
- boolean cancelSave ()
- boolean drop (in UnicodeString location)

Detailed Description

The `IRepositoryManagement` interface contains technical operations on repositories such as operations for opening, closing, saving, etc. This interface is a complement to `RAAPI`.

Member Function Documentation

boolean IRepositoryManagement::exists (in UnicodeString *location*)

Checks whether the given location is already occupied by some repository of the same type. This can be used to ask for the user confirmation to drop an existing repository, when creating a new one at the same location.

Parameters

location	a string denoting the location to check. The location string is is specific to the type of the repository, e.g., for ECore this is the .xmi file name, for JR this is the folder name, etc. TDA Kernel requires a URI, containing the repository name followed by a colon followed by a repository-specific location, e.g., "jr:/path/to/repository".

Returns
whether the given location is already occupied by some repository of the same type.

boolean IRepositoryManagement::open (in UnicodeString *location*)

Opens or creates (if the repository does not exist yet) the repository at the given location. This can be used to ask for the user confirmation to drop an existing repository, when creating a new one at the same location.

Parameters

location	a string denoting the location of the repository. The location string is is specific to the type of the repository, e.g., for ECore this is the .xmi file name, for JR this is the folder name, etc. TDA Kernel requires a URI, containing the repository name followed by a colon followed by a repository-specific location, e.g., "jr:/path/to/repository".

Returns
whether the repository has been opened or created.

void IRepositoryManagement::close ( )

Closes the repository without save.

See Also
IRepositoryManagement::startSave
IRepositoryManagement::finishSave
IRepositoryManagement::cancelSave

boolean IRepositoryManagement::startSave ( )

Starts the two-phase save process of the repository. The save process can be rolled back by calling `cancelSave` or commited by calling `finishSave`.

Returns
whether the operation succeeded. If `false` is returned, neither `cancelSave`, nor `finishSave` must be called.

See Also
IRepositoryManagement::finishSave
IRepositoryManagement::cancelSave

boolean IRepositoryManagement::finishSave ( )

Finishes the two-phase save process of the repository. After finishing, the save process cannot be rolled back anymore.

Returns
whether the operation succeeded.

See Also
IRepositoryManagement::startSave
IRepositoryManagement::cancelSave

boolean IRepositoryManagement::cancelSave ( )

Rolls back the started save process. The repository content on the disk (or other media) is returned to the previous state. The repository content currently loaded in memory is not changed.

Returns
whether the operation succeeded.

See Also
IRepositoryManagement::startSave
IRepositoryManagement::finishSave

boolean IRepositoryManagement::drop (in UnicodeString *location*)

Deletes the repository at the given location. The repository must be closed.

Parameters

location	a string denoting the location of the repository. The location string is specific to the type of the repository, e.g., for ECore this is the .xmi file name, for JR this is the folder name, etc. TDA Kernel requires a URI, containing the repository name followed by a colon followed by a repository-specific location, e.g., "jr:/path/to/repository".

Returns
whether the operation succeeded.