LATVIJAS UNIVERSITĀTES
RAKSTI
733. SĒJUMS

# Datorzinātne un informācijas tehnoloģijas

SCIENTIFIC PAPERS
UNIVERSITY OF LATVIA
VOLUME 733

# Computer Science and Information Technologies

SCIENTIFIC PAPERS
UNIVERSITY OF LATVIA
VOLUME 733

# Computer Science and Information Technologies

Latvijas Universitāte

LATVIJAS UNIVERSITĀTES
RAKSTI

733. SĒJUMS

# Datorzinātne un informācijas tehnoloģijas

Latvijas Universitāte

UDK 004(082)
Da 814

ISSN 1407-2157

ISBN 987-9984-825-47-0

## Contents

# SOFTWARE DEVELOPMENT AND TESTING

# Managing Uncertainty in Globally Distributed Software Development Projects

Darja Šmite, Juris Borzovs

University of Latvia
Raiņa bulv. 19, LV-1586, Rīga, Latvia
{Darja.Smite, Juris.Borzovs}@lu.lv

**Abstract.** Global software development is not a phenomenon but a reality nowadays. However, it is still poorly explored. Lack of awareness of the particular factors inherited in the nature of globally distributed software projects makes practitioners struggle and invent new approaches to survive. It uncovers the necessity to support risk management activities. This paper describes a Knowledge Base and a Risk Barometer developed to support practitioners who lack experience in global projects. Particularities of globally distributed projects and their effect on project performance are formalized in a reusable framework for managing uncertainty. The described tools provide input for risk identification and help to evaluate risks based on the experience from former projects.

## 1 Introduction

Global Software Development (GSD; also known as Global Software Engineering (GSE), and Globally Distributed Software Development (GDSD)) has become the key trend in the area of software engineering. It is motivated by the opportunities of reaching mobility in resources, obtaining extra knowledge, speeding time-to-market, and increasing operational efficiency. And yet, GSD is accompanied by both opportunities and problems. Many specialists recognize globally distributed software development as more complex than even the most complex project managed entirely in house [8], [6]. Globally distributed software development expands the concept of traditional outsourcing and addresses transition of common in-house manner of software development to more complex software life cycle activities distributed among teams separated by various boundaries, such as contextual, organizational, cultural, temporal, geographical, and political. This type of development environment can therefore be characterized by its heterogeneity, virtualness, and inter-organizational collaboration that are impediments for effective communication and cooperation of the teams involved in completion of a joint project. New unique pressures of project management that appear to have nothing to do with the technical nature of the project and at the same time are reasons that can doom a virtual project is something even capable managers often overlook [8]. Practitioners claim that they have to experiment and quickly adjust their tactical approaches for leveraging global software development risks [2]. Researchers admit that although a body of knowledge on global software development has been crafted over time, the art and science of organizing and managing globally distributed software development is still evolving [4]. However, despite the fact that global software development is said to be different from common in-house software development projects [11], [9], [8], peculiarities of globally distributed software projects have not been explicitly formalized.

Risk analysis concepts have been applied to identify and evaluate particular negative events that might cause globally distributed software project failure. Threats that endanger globally distributed software project success are found to be quite distinctive from in-house project threats. Global risks are proved as just the part of everyday existence that cannot be avoided [11], [10], and must be confronted on a continuous basis. However, empirical results that would help evaluating the magnitude of consequences of these environmental factors and threats provide contradictious views. In addition, related studies describe [3] that as simple as it sounds, many organizations are unable to manage risks effectively. Accordingly, the research described in this paper (also published in related research papers [12] and [13]) focuses on exploring the unique threats of globally distributed projects, effect of these threats on project performance, and ways to overcome these threats before they lead to project failure.

## 2 Research Methodology

### 2.1 Grounded Theorizing

Grounded theory building methodology developed by Glaser and Strauss [5] was chosen as the basis for the study. This methodology introduces a qualitative approach that generates theory from observation [14]. Theory-creating studies are very suitable for exploratory investigations, i.e., when there is no prior knowledge of a part of reality or a phenomenon [7]. Grounded theories, because they are drawn from data, are likely to offer insight, enhance understanding, and provide a meaningful guide to action [15].

Understanding of global factors and threats evolved grounded by systematically gathered and analyzed data about the phenomenon. The data was gathered from a variety of sources, including qualitative interviews and enhanced analysis of related research literature. Data analysis was performed according to principles prescribed by grounded theory through applying open, axial, and selective coding techniques [14], also called as theoretical sampling. A Lotus Notes-based database was used for data maintenance facilitating in easy categorization.

## 2.2 Data Sources

Various data sources were used for building the theory, including qualitative interviews with experienced project managers from the investigated software house, research literature (journal articles, papers form conference proceedings), and books on global software development.

We conducted 13 interviews with 9 project managers, who represented all software development departments of the investigated software house running projects with customers from different remote locations and were appointed by the heads of departments as the most experienced ones. The interviews were held by means of semi-structured interviewing and open questions. The interviews were written down for further analysis.

We performed an extensive literature analysis using input from 33 research articles on global software development published in the related conference proceedings and journals such as IEEE Software, and Communications of ACM.

Literature analysis and interviews with experienced project managers provided a representative input regarding the phenomenon under study.

## 2.3 Data Analysis

We created a Lotus Notes based database to maintain the gathered data items and support data analysis. Sources of information and each data analysis iteration results were kept within the certain item's history for traceability opportunities.

Data analysis started with an open coding for data breaking down, examining, comparing, conceptualizing, and categorizing. While examining data sources, expressions related to particular project characteristics, different negative events, consequences or practices were identified and labelled. Data analysis resulted in total of 253 GSD related issues, which were then stored into the database. Open coding then continued with categorizing. Each issue at the beginning represented a single category, the existing labels then were analyzed in order to identify issues that are similar in meaning. Those were then grouped under more general concepts called "categories". E.g., the labels "Cultural barriers", "Cultural distance", and "Poor cultural fit" were coded under a joint category "Poor cultural fit".

This reduced the number of GSD related categories to 163.

Examination of the existing categories showed that many issues were interrelated and formed cause-effect interconnections. Axial coding was used for deriving connections between the existing categories and the risk management concepts, during which the identified GSD related issues from open coding were categorized into a hierarchy of sub-categories as follows:

- Global factors – root of global threats, that distinguish global projects;
- Global threats – items or activities that have potential for negative consequences and result from one or a combination of global factors;
- Consequences – negative outcome of a threat;
- Practices – recommendations for leveraging the risks.

Selective coding was used for systematically validating relationships and filling in categories that needed further refinement and development. Axial coding showed that some of the existing categories had to be reconsidered. For some categories identified during open coding this meant dividing into two or even more categories.

E.g., the category "E-mail communication causes time delays and misunderstandings" was divided into "E-mail communication" – a threat, and "Time delays" and "Misunderstandings" – consequences. "E-mail communication" was then united with one of the more general existing categories – "Asynchronous communication". The relations between the three categories were then produced.

Refining the dependences between the new issues and tagging the categories with the sources were performed through selective coding.

To strengthen the results of this study regarding global factors and threats, only items that appeared more than once were selected, i.e. the threats that are strongly dependent on particular environment were omitted. Due to the industrial background of the research, customer related issues were also omitted after data analysis. New versions of records were processed, saving the history and notes reflecting the decisions within the database.

### 2.4 Results

Grounded theorizing resulted in 7 global factors, 32 supplier related threats, 7 supplier related major consequences, and 32 supplier related practices (for more detail see the following chapters). As the theory has been built, we conclude that the most valuable results refer to global factors and threats. In its turn consequences and relationships between the global factors, threats, and consequences are weak and inconsistent. Therefore, we conducted a survey on 38 globally distributed projects to validate these considerations empirically and improve the theory.

## 3 Particularities of GSD Projects

The nature of global software development brings forward new areas of concern that require careful attention from project managers. Practitioners that have for a long time successfully managed in-house projects, now face new challenges that make them struggle to bring the projects to the end within the budget, time schedule, and with the satisfied customers. One may think that the influence of globalization on software development remains limited by distributing end customers from their software development suppliers and at the same time having no effect on the life cycle processes. However, the concept of globally distributed software development prescribes separated teams from different organizations and/or locations work together on a joint project execution. These organizations form supply chains of different complexity thus increasing the complexity of software process distribution.

## 3.1 Global Factors

The major distinguishing factors of globally distributed software projects identified by this research are the following [12]:

- **Multisourcing** – multiple distributed member involvement in a virtual team that develops software by joint effort, characterized by a number of collaboration partners;
- **Geographic distribution** – distance between the partners involved in the project;
- **Temporal diversity** – the level of working hours overlay, which most frequently differs from exact time zone differences;
- **Socio-cultural diversity** – level of social, ethnic, and cultural fit that can differ even between the teams from one national location;
- **Linguistic diversity** – language difference, characterized by the level of language skills of the project members;
- **Contextual diversity** – the level of organizational fit or heterogeneity, characterized by diversity in process maturity and inconsistency in work practices;
- **Political and legislative diversity** – level of legislative consistency and sources of political threat.

Inter-organizational projects involve joint inter-organizational resources and are developed by global software teams also referred to as virtual teams. Accordingly, software processes are distributed between the remote team members and are affected by organizational work practices and habits. The differences between in-house and globally distributed projects can be also illustrated as follows (see Fig.1 and Fig.2).

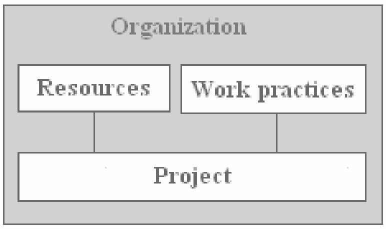


**Fig. 1.** Intra-organizational projects

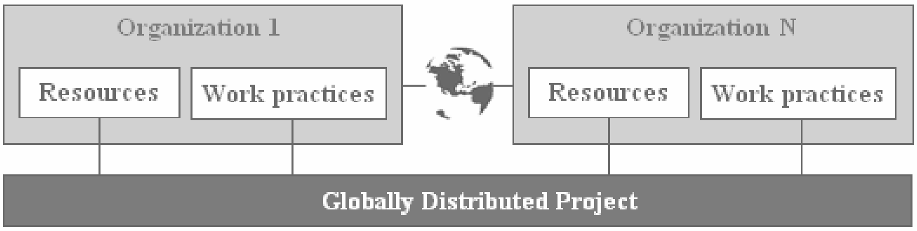


**Fig. 2.** Inter-organizational globally distributed projects

Global factors inherited in the nature of GSD projects are recognized as roots of global threats that can endanger the success of a global project. They indeed demonstrate the peculiar nature of globally distributed software development and indicate the forces that act as impediments during a project. Each of the global factors and their combination causes various threats and conditions for negative outcomes.

Therefore we emphasize the uniqueness of globally distributed environment and mark that awareness of global factors that are inherited in the nature of globally distributed project environment can help practitioners either reduce the probability or the magnitude of unexpected negative outcomes. However, if the global factors exist, they often cannot be avoided.

### 3.2 Global Threats

Global factors characterize different impediments for joint collaboration grounded in different types of diversity existent between the remote partners. These factors have considerable impact on the software life cycle processes. To limit or avoid the impact of global factors, project managers require knowledge on what to be aware of. Accordingly, in order to support project managers in timely risk management, we have collected information on global threats that endanger global projects. Global threats discovered within the research are as follows [12]:

- Customer has complex hierarchy and/or several problem escalation levels
- Supplier has complex hierarchy and/or several problem escalation levels
- Diversity in process maturity and/or inconsistency in work practices
- Lack of understanding of each other's context of decision making
- The customer believes that the work cannot be done from a far off location
- Lack of trust and commitment
- Increased cost of logistics of holding face to face meetings
- Increased level of reporting on project progress to the customer
- Increased virtualness
- Lack of language skills by supplier
- Terminology differences
- Customer's employees unwillingness to collaborate caused by threat of being fired due to switching to outsourcing mode
- Faulty effort estimates
- Increased level of complexity of project management
- Increased level of unstructured poorly-defined tasks
- Increased complexity of spreading awareness and knowledge
- Lack of common goals
- Lack of experience and expertise of the customer with outsourcing projects
- Lack of experience and expertise of the supplier with outsourcing projects
- Lack of joint risk management
- Lack of team spirit
- Poor or disadvantageous distribution of software development activities
- Relatedness with other suppliers

- Poor cultural fit
- Dominant use of asynchronous communication with the customer
- Time zone difference
- Lack of clarity about responsibility share
- Poor or complex project measurement
- Increased complexity of project, activity, human resources, and delivery planning
- Poorly defined or inconsistent SRSs
- Poorly defined or inconsistent software design and/or architecture
- Poor artefact version control

The identified global threats are not categorized according to their root factors, because a threat can be caused by a combination of global factors. These threats also tend to be general. We aimed to avoid too detailed categorization of the threats to eliminate the complexity of correlated threat hierarchy. It also relieves the process of threat identification – too long checklists with odd issues are rarely used.

Accordingly, this list of threats does not comprise all possible negative events that can endanger a global project. However, it is a useful guide to risk management that is based on previous experiences.

## 4 Outcome Predictions in Global Projects

As previously emphasized, awareness of global factors and threats is essential for global project success. However, knowing about possible threats does not mean that organizations can evaluate the extent of each factor and threat. Limited experience and expertise in globally distributed software development often drives organizations to sudden problems due to underestimation of the hidden threats. Accordingly, awareness of the negative outcome of each factor plays an important role in estimating its severity.

We therefore offer an experience-based risk-oriented approach to leverage global threats [13]. Traditional risk management concepts in this approach are introduced by components that characterize the effect of global threats on project performance. These are: probability of a threat to endanger a certain project success criteria and the magnitude of the negative outcome of a threat. We additionally calculate the probability of negative outcome for each threat based on global project survey data, which extends traditional risk analysis concepts and introduces an approach to calculate future outcome predictions.

Experience data for effect evaluation were collected through a survey of global software projects run by Latvian software houses. We have gathered data from 38 globally distributed software projects that provide a representative insight in what and how endanger global projects considering specifics of Latvia.

## 4.1 Basic Concepts

Software risk management can be defined as an attempt to formalize risk oriented correlates of development success into a readily applicable set of principles and practices [1]. However, practitioners often misuse risk terminology. Therefore the basic concepts and rules are defined as follows:

1) Term ***threat*** is used to describe possible negative events that can lead a project to its failure. E.g. Lack of experience with outsourcing projects.
2) Each threat has its probability of occurrence evaluated through the frequency of occurrence within the surveyed projects.
3) Each threat is evaluated for its negative outcome. The following criteria are used in negative outcome evaluation for this research [1]:
    - Budget overrun;
    - Unexpected management costs;
    - Customer cost escalation;
    - Time delays;
    - Late product delivery;
    - Customer dissatisfaction;
    - Supplier team's undermined morale;
    - Disputes and litigations.
4) A threat can cause different levels of negative outcome. E.g. dominant asynchronous communication may cause considerable time delays, but insignificant temporal distance only minor delays.
5) Evaluation of the level of negative outcome of the threat is called magnitude of the negative outcome. To conform to traditional risk management concepts, magnitude of the negative outcome is calculated for each pair [threat; consequence]. In other words, the threat of poor cultural fit can cause e.g. minor time delays, considerable customer dissatisfaction, disastrous undermined morale of the supplier team and none effect on other success criteria.
6) Magnitude of the negative outcome and frequency of occurrence are evaluated according to a quantitative scale with an equivalent qualitative scale for interpretation as seen in Table 1.

[1] Project compliance with budget and schedule, customer satisfaction and software product quality are the major success criteria for the project according to related literature. However, software product quality was not included in the list of indicators due to high risk of bias of the given evaluation. On the other hand, the list of project success criteria was extended due to the following reasons:

- Differentiation of causes of budget overrun;
- Time delays have been emphasized as a source of downtime, which does not obligatory drive to late product delivery;
- Supplier team's undermined morale is an important success criteria considering the industrial research background (supplier side of the project);
- Disputes and litigations are also possible negative outcomes that were additionally explored as possible causes of project cancellation.

**Table 1.** Rating scales

| Magnitude of the negative outcome | |
|---|---|
| 0 | None |
| 1 | Negligible |
| 2 | Minor |
| 3 | Moderate |
| 4 | Significant |
| 5 | Disastrous |
| **Frequency of occurrence** | **Probability** |
| 0 | Improbable |
| (0-10%] | Doubtful |
| (10-20%] | Unlikely |
| (20-40%] | Possible |
| (40-80%] | Probable |
| (80-100%] | Certain |

7) The combination of Magnitude of the negative outcome and frequency of its occurrence (for each pair [threat; consequence]) form risk exposure [1] that is widely used in traditional risk comparison and prioritization. Multiplication can be used for quantitative evaluation, and matrixes for qualitative evaluation. Accordingly it helps to identify threats that have the most severe effect on the project performance separately for budget overrun, time delays, customer dissatisfaction, etc.

## 4.2 Approach to Calculate Outcome Predictions

In order to support risk management activities for practitioners, **Probability of negative outcome** is evaluated using frequency of occurrence of the negative outcome of the threat on the certain level by computing frequencies of lower effect levels with those of higher effect levels, in other words cumulative values, according to the following equation [13]:

$$\text{Prob}\ (t,\ c_{i,j}) = \sum_{k=j}^{5} \text{Freq}\ (t,\ c_{i,k}) \qquad \textbf{(1)}$$

Variables and functions:

| | |
|---|---|
| t – | threat; |
| ci,j – | outcome, where first index indicates the certain negative outcome (budget overrun, time delays, etc.) and the second – its level (1, 2, 3, 4 or 5) |
| Freq (t, $c_{i,j}$) – | frequency of occurrence of the negative outcome of the certain level of the certain threat; |
| Prob (t, $c_{i,j}$) – | probability of the negative outcome of the certain level of a threat. |

## 4.3 Survey Overview

The previously compiled list of threats was offered to various project managers and team leads for evaluation. Representative data set was collected using a survey instrument by mailing the developed questionnaire to a selected sample of employees in the investigated company, whose job title was project manager or equivalent, e.g., development manager or development team leader. In addition, the questionnaire was made accessible in other 4 small software houses, where the project managers could participate in the survey if interested.

The complexity of lifecycle distribution in the investigated projects varied from direct subcontracting to a complex chain of 10 subcontractors involved in completion of a joint project. Respondents' experience varied from 3 to 30 years. Other characteristics considering the investigated projects under study are given in Table 2.

**Table 2.** Characteristics of the Surveyed Projects

| Characteristics | Survey results | | |
|---|---|---|---|
| Collaboration type<br><br>*Describes entities involved in the joint project, e.g. customer→ supplier (1→1), or customer → multiple suppliers (1→N)* | 1→1→1 | | 13 projects |
| | 1→1 | | 10 projects |
| | 1→N | | 7 projects |
| | 1→1→N | | 6 projects |
| | 1→N→N | | 2 projects |
| Number of partners | 2 | | 11 projects |
| | 3 | | 16 projects |
| | 4 | | 3 projects |
| | more than 4 | | 5 projects |
| Project success<br><br>*Subjective evaluation given by the project managers considering budget and calendar compliance, and customer satisfaction, using the scale 1-10.* | Successful: 15,8% | 10 | 4 projects |
| | | 9 | 2 projects |
| | Somewhat successful: 50,0% | 8 | 7 projects |
| | | 7 | 12 projects |
| | Unsuccessful: 34,2% | 6 | 5 projects |
| | | 5 | 4 projects |
| | | 4 | 2 projects |
| | | 3 | 1 project |
| | | 1 | 1 project |

The following data were gathered during the survey:

- Project characteristics (collaboration model, project activity distribution, location of partners, project type, project status, success evaluation, etc.);
- Report of frequency of occurrence of the listed threats in the projects;
- Evaluation of the impact of each experienced threat on the project results.

## 4.4 Survey Data Analysis

Survey data were kept in and analyzed using SPSS ® 14.0 tool[2], which provided a broad range of capabilities for the entire analytical process, including easy data search and categorization, powerful statistics, tabular and graphical representation of the results. Data was recorded within 316 variables.

Quantitative analysis of 38 globally distributed project survey data was performed to evaluate the effect of global factors and threats on GSD project performance. Survey responses have been statistically analyzed to compute the following values for each threat:

- Frequency of occurrence;
- Average outcome;
- Probability of certain level of the certain negative outcome.

**Frequency of occurrence** is based on the historic information from the survey.

**Average outcomes** of a threat are minimum conditions that practitioners have to take into account while collaborating in the globally distributed project environment. Survey data contain evaluation of the magnitude of the negative outcome of each threat. Magnitude of the negative outcome of each threat is evaluated using a linear scale: [0, 1, 2, 3, 4, 5] or its equivalent [None, Negligible, Minor, Moderate, Significant, Disastrous] as described earlier.

**Probability of certain level of the negative outcome** or **negative outcome predictions** are evaluated as cumulative values using frequency of occurrence of each threat to cause negative outcome of a certain level by computing frequencies of lower levels of impact with those of higher according to the definition given above.

## 4.5 Risk Barometer

Considering the length of the list of global threats and complexity of risk analysis, we developed a tool that computerizes project outcome predictions correspondingly labelled as "Risk Barometer" [13].

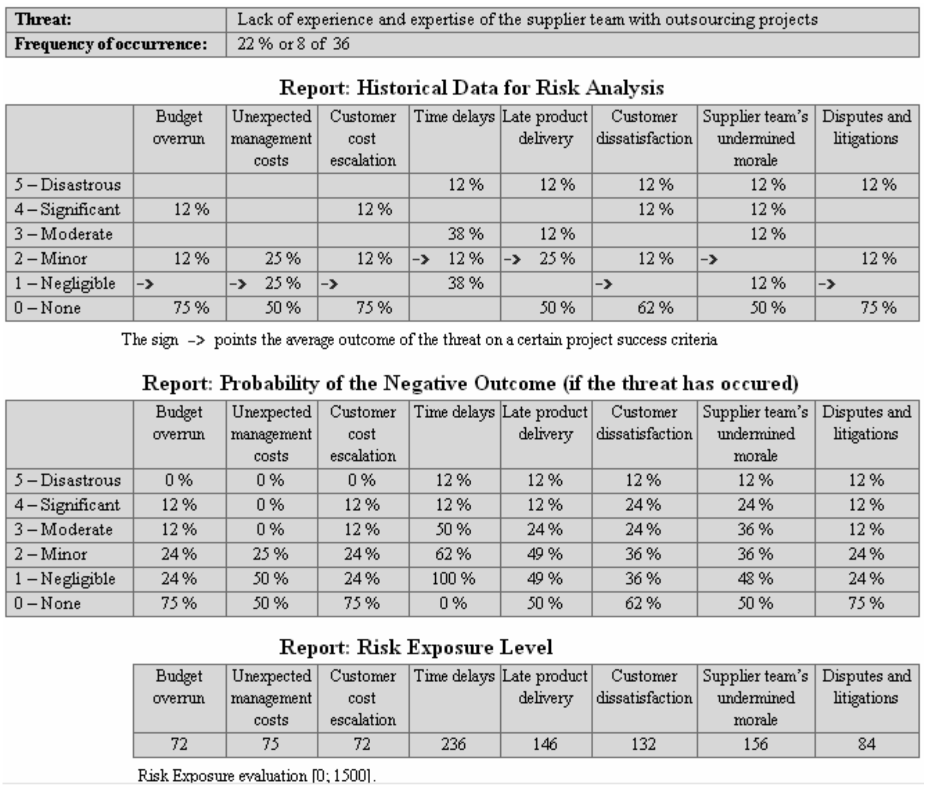
Risk Barometer is developed as a Lotus Notes based function aiming to support outcome predictions in global projects especially for project managers who lack awareness of possible negative events and their consequences in globally distributed environment. Risk Barometer performs its predictions on the basis of historical data from post-project risk evaluation reports. Risk Barometer and historical data is integrated in the Knowledge Base that serves as a central repository for organizational learning support. The survey provided the first input for outcome predictions from anonymous survey data gathered during the research and kept within the Knowledge Base. New project experiences can be added to continuously support Risk Barometer prediction improvements.

Risk Barometer is intended for project managers to evaluate global project threats, considering the probability of occurrence and possible negative impact that can be compared with historical data from other projects. We foresee that hidden threats and their outcomes, such as hidden costs, unobvious sources of time delays and customer

[2] SPSS Software Solutions Online – http://www.spss.com/

dissatisfaction, will help inexperienced project managers prepare against impediments inherited in the nature of globally distributed projects.

An example of Risk Barometer predictions for a threat of lacking experience and expertise in outsourcing projects can be seen in Fig.3.



| | |
|---|---|
| **Threat:** | Lack of experience and expertise of the supplier team with outsourcing projects |
| **Frequency of occurrence:** | 22 % or 8 of 36 |

**Report: Historical Data for Risk Analysis**

| | Budget overrun | Unexpected management costs | Customer cost escalation | Time delays | Late product delivery | Customer dissatisfaction | Supplier team's undermined morale | Disputes and litigations |
|---|---|---|---|---|---|---|---|---|
| 5 – Disastrous | | | | 12 % | 12 % | 12 % | 12 % | 12 % |
| 4 – Significant | 12 % | | 12 % | | | 12 % | 12 % | |
| 3 – Moderate | | | | 38 % | 12 % | | 12 % | |
| 2 – Minor | 12 % | 25 % | 12 % | -> 12 % | -> 25 % | 12 % | -> | 12 % |
| 1 – Negligible | -> | -> 25 % | -> | 38 % | | -> | 12 % | -> |
| 0 – None | 75 % | 50 % | 75 % | | 50 % | 62 % | 50 % | 75 % |

The sign -> points the average outcome of the threat on a certain project success criteria

**Report: Probability of the Negative Outcome (if the threat has occured)**

| | Budget overrun | Unexpected management costs | Customer cost escalation | Time delays | Late product delivery | Customer dissatisfaction | Supplier team's undermined morale | Disputes and litigations |
|---|---|---|---|---|---|---|---|---|
| 5 – Disastrous | 0 % | 0 % | 0 % | 12 % | 12 % | 12 % | 12 % | 12 % |
| 4 – Significant | 12 % | 0 % | 12 % | 12 % | 12 % | 24 % | 24 % | 12 % |
| 3 – Moderate | 12 % | 0 % | 12 % | 50 % | 24 % | 24 % | 36 % | 12 % |
| 2 – Minor | 24 % | 25 % | 24 % | 62 % | 49 % | 36 % | 36 % | 24 % |
| 1 – Negligible | 24 % | 50 % | 24 % | 100 % | 49 % | 36 % | 48 % | 24 % |
| 0 – None | 75 % | 50 % | 75 % | 0 % | 50 % | 62 % | 50 % | 75 % |

**Report: Risk Exposure Level**

| Budget overrun | Unexpected management costs | Customer cost escalation | Time delays | Late product delivery | Customer dissatisfaction | Supplier team's undermined morale | Disputes and litigations |
|---|---|---|---|---|---|---|---|
| 72 | 75 | 72 | 236 | 146 | 132 | 156 | 84 |

Risk Exposure evaluation [0; 1500].

**Fig. 3.** Example of predictions by the Risk Barometer: for a threat

## 5 Discussion

### 5.1 Particularities of Global Projects

The derived lists of global factors and threats make the peculiarity and complexity of globally distributed software development obvious. Global software development puts new demands on the software processes stressed by an increased complexity of project coordination (through temporal and geographical distances), communication (lacking proximity and cultural diversity), cooperation (lacking trust and commitment), infrastructure management (uniting heterogeneous contexts) and other aspects of distributed software development. The global factors characterize the distinguishing nature of globally distributed software development projects by emphasizing unavoidable elements that are inherited in this kind of work environment and shall be analyzed throughout the project.

The list of global threats provides guidance for effective risk identification and demonstrates the various ways that global factors may act. This knowledge is especially demanded by practitioners that lack previous experiences in developing software with globally distributed partners.

Furthermore, practices applied for global risk mitigation shall act as a counterforce against global threats and reduce the effect of global factors and threats on project results (see Fig.4).

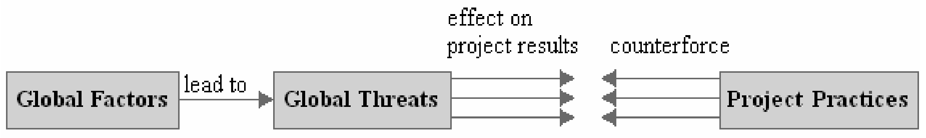


**Fig. 4.** Project practices as a counterforce for global threats

The extent of the effect of global factors and threats on project results shows that they indeed may drive projects to failure if not managed on time. Global threats may lead to a considerable negative outcome on project budget, cause calendar deviations and customer dissatisfaction.

## 5.2 Application of Risk Barometer

Risk Barometer provides a general overview of the outcome of each threat and probability of its occurrence. Since global projects are so different and the extent of global factors may influence occurrence of global threats in particular circumstances, project managers may not ground their risk predictions only on personal experience. It is therefore recommended to use an experience-based approach to analyse global risks and monitor them on a regular basis among different projects in an organization.

Application of Risk Barometer is feasible in any global project despite its size and complexity. Global factors and threats inherited in the nature of globally distributed environment will not vanish if the project will last only a month or consider a well-known task that shall be performed by well-trained developers. Project managers shall use outcome predictions to see what kind of effect they may cause and report on real situation after the end of the project.

Risk Barometer can also be used to evaluate the sources of necessary investments in globally distributed projects by analyzing sources of budget overrun, unexpected management costs, and customer cost escalation. It can be also useful to point out sources of time delays for better effort estimation.

## Conclusions

The results of the research reported in this paper support conclusion that globally distributed software development significantly differs from in-house software development [12]. Global factors and threats provide a valuable ground for effective risk identification supplemented by project outcome predictions that support further risk analysis for practitioners. In contradiction to many studies conducted from the customer perspective, this study investigates and includes global project problems from supplier perspective thus providing a useful support for Latvian and other software houses that operate as outsourcing service providers.

GSD project case studies range from announcements of tremendous success to total failure. No research so far has provided a clear vision of the true amount of investments necessary to make global software projects work. Risk Barometer forms a ground for an experience-based risk-oriented approach for GSD project outcome evaluation [13]. The results of Risk Barometer performance include observations of budget, schedule, and customer satisfaction threats – their significance and historical frequency of occurrence. Risk Barometer extends the traditional risk analysis approach and provides automatic prediction calculations on the basis of previous project data.

**Acknowledgments**: The authors appreciate valuable research input received from the project managers within the investigated software house. This research is partly supported by the European Social Fund and the Latvian Council of Science project No. 02.2002 "Latvian Informatics Production Unit Support Program in the Area of Engineering, Computer Networks, and Signal Processing".

## References

[1] Boehm B.W., "Software Risk Management: Principles and Practices", IEEE Software, Vol. 8, No.1, 1991, pp. 32-41
[2] Carmel, E., Agarwal, R. "Tactical Approaches for Alleviating Distance in Global Software Development", IEEE Software, Vol. 18, No. 2, 2001, pp.22-29
[3] Carr M.J., "Counterpoint: Risk Management May Not Be for Everyone", IEEE Software, Vol.30 No.5, 1997, pp. 21-24
[4] Damian D. and Moitra D., "Global Software Development: How Far Have We Come?", IEEE Software, Vol. 23, No.5 2006, pp.17-19
[5] Glaser B., Strauss A. "The discovery of grounded theory: Strategies of qualitative research", Wiedenfeld and Nicholson, London, 1967
[6] Iesalnieks J., Gulbe B. "Old and New Europe as IT Partners", Baltic IT&T Review, Nr.2 (33), 2004, pp.38-40
[7] Jarvinen P. "On Research Methods", Opinpajan Kirja, Tampere, 2001
[8] Karolak D.W., "Global Software Development: Managing Virtual Teams and Environments", IEEE Computer Society, 1998
[9] Merriam-Webster Online Dictionary. Available online www.m-w.com

[10] Pavlovs S., “Рига интересна, ибо мультикультурна” Коммерсант Baltic daily / 25.04.2005, Nr.78, pp.5

[11] Sahay S., Nicholson B., Krishna S., “Global IT Outsourcing: Software Development across Borders”. Cambridge University Press, 2003

[12] Smite D., Borzovs J., „A Framework for Overcoming Supplier Related Threats in Global Projects”, In Proceedings of the Int. Conf. on European Software Process Improvement (EuroSPI), published in LNCS by Springer Verlag, October 2006, Finland, pp. 49-60

[13] Smite D., “Project Outcome Predictions: Risk Barometer Based on Historical Data”, In proceedings of the Int. Conf. on Global Software Engineering (ICGSE) by IEEE Computer Society, August 2007, Germany, pp.103-112

[14] Strauss A., Corbin J. “Basics of Qualitative Research – Grounded Theory Procedures and Techniques”, Sage Publications, Newbury Park Ca, 1990

[15] Strauss, A., Corbin, J., “Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory”. Thousand Oaks, CA: Sage Publications, 1998

LATVIJAS UNIVERSITĀTES RAKSTI. 2008, 733. sēj.:
DATORZINĀTNE UN INFORMĀCIJAS TEHNOLOĢIJAS 24.–37. lpp.

# Application of Smart Technologies in Software Development: Automated Version Updating

Zane Bičevska, Jānis Bičevskis

Datorikas Institūts DIVI, A.Kalniņa str. 2-7, Rīga, Latvia

University of Latvia, Raiņa blvd. 19, Rīga, Latvia
Zane.Bicevska@di.lv, Janis.Bicevskis@lu.lv

**Abstract**. This paper proposes software version update solutions in compliance with smart technologies [1] - architectural designs and software components which using metainformation on the system and its operation requirements are able to solve efficiently the problems of software maintenance.

In order to ensure automated version update the authors propose several mutually independent mechanisms such as environment testing, software version update, automated data migration to the latest versions as well as automated self-testing of the installed version on internal consistency.

Based on experience in software development, distribution and introduction, the authors identify and describe particular framework principles for integration of automated function of software version update into produced software.

**Keywords:** Software engineering, Maintenance, Testing, Smart Technologies.

## 1 Introduction

The IT industry is characterized by multiplicity of infrastructure, applications and executable environments. Also, significant resources are required to adapt customized software for usage in multiple environments. The fierce competition in IT industry dictates fast appearance of high quality and innovative products on the market. Thus, the software developers encounter serious challenge – under significant time pressure to develop and deliver a software usable in multiple environment.

There are two approaches used in the praxis to ensure usability of developed and delivered software. In the first case requirements for software usability are defined, e.g. minimal requirements for hardware and version of operating system. However, this approach may cause problems such as mutually conflicting software versions. Additional difficulties arise if users are not ready or do not understand how to satisfy the defined requirements.

The other approach foresees that developers ensure usage of software in multiple environments and their combinations. The platform independence that foresees usage of software in various operating systems, on different data base management systems and using various browsers might require significant additional resources. In fact, all efforts to develop "universally applicable" software result in necessity to develop technical solutions for each of the environments.

Usage of universal algorithms and metadata only partially can unburden development of "universally applicable" software, for in result the executable code will remain platform dependent. Therefore, users of software are forced to encounter difficulties arising from software installing and updating – considerable time consumption and efforts are spent analyzing collision and problem causes, for consultations and for improving software to ensure its adequacy to the specific environment.

Consequently, the need arises for certain software development principles to ensure collision free (or reduced to minimum) software installation and download of software version updates. The following parts of the paper deal with smart technologies principle in essence – software should comprise features that support installing the latest versions without user assistance and ensuring the following:

1) automated analysis of the software compliance with environmental requirements
2) automated download and installation of the latest version of the software modules
3) automated personalization and individual data migration to the latest version
4) self-testing of the latest version, checking operating correctness of the information system on critical functionality
5) generation of back-up files and system renewal in case of failure

Even partial application of these principles (we are not aware of the cases when such principles were applied in full extent) has proven to be very useful.

## 2 Software Life Cycle Models and Smart Technologies

Over the years a number of discussions have been devoted to software development life cycle models [2] and analysis on strengths and weaknesses of linear and incremental models have been performed. A new approach (e.g. principles agile software development process) has been added to lengthy discussions. Nevertheless, the main attention in software life cycle models traditionally is being paid to software development, including requirement gathering (specification), design, implementation and testing. Less research is devoted to the system maintenance and operation despite the fact that these aspects take up the main part of the duration of a successful system.

In real life development of software is just the very first phase of software life cycle. The most time consuming phase of software life cycle is software maintenance including user support, software upgrade, and functionality extension services (e.g. MS Windows is being under development for more than 15 years!)

Every successful software solution has been used and improved for significantly longer time period that it was created for. A successfully developed system might be used for many years simultaneously modified, improved with new functions permanently satisfying occurring client needs. Thus, every time changes occur in software or operational environment, the issue of testing software's correctness becomes crucial.

Therefore, to ensure software reliability in long-term, the system already in its early development phases should comprise not only client defined functionality but also additional mechanisms to support usage, maintenance, and further development of software.

The smart technology is based on the idea about "smart" software that like living beings is able to "self-management". It means the software should be able to handle unpredictable events in unknown environments. A smart technology conform software should be able to deal with internal (related to internal structure and functioning of the system) and external (originated outside of the system) events adequately.

The targets set in the self-adaptive software [3, 4] partially overlap with the principles of the smart technologies. Self-adaptive software researchers are focusing on software ability to adapt itself to implementation environment. This report sets different targets: troubleshooting SW exploitation failures by applying automatic indication of possible failures and reporting them to staff. Implementation of this approach is more convenient to use in practice.

Smart technology oriented approach is illustrated in the succeeding figure (Fig.1). It demonstrates that the core functionality of software solution should be enhanced with several additional features supporting the usage, maintenance, and further development of software. These additional features called smart technologies are created in the process of software design and implementing similar to scaffolding in the construction process of a building. However unlike the building process the "scaffolding" of an information system is never taken down; it stays in the information system for its whole life time.

Relatively, these features are divided into two main groups:

1) External stability – ability to analyse environment consistency to its performance conditions

2) Internal stability – ability to check and maintain internal consistence

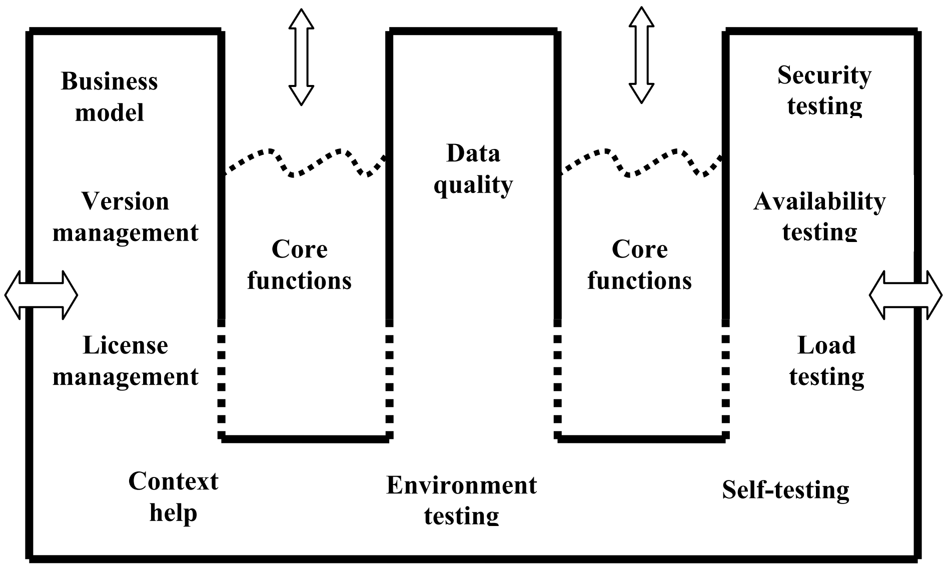


**Fig. 1.** Components of Smart Technology

Software fits to the principles of smart technology if it provides the following features:

1) Business model is incorporated into software
2) Version management – automatic updating of versions from the central server, including the adequate conversion of data structures and data
3) License management – automatic control of licence conditions, updating of standard and specific software according to them
4) Context help – integrated functionality of context sensitive help system
5) Environment testing - ability to analyse the external environment (for example, options of operating system and data base management system) and to adapt itself to the specific environment
6) Self-testing - ability to check the internal integrity by automatic execution of test cases in the productive environment and to inform users and developers about detected inconsistencies
7) Load testing - ability to provide monitoring of performance and load balancing
8) Availability testing – monitoring of system availability using agent technologies; ability to inform remote about the status of the software and additional components needed for a correct functioning
9) Security testing - monitoring of system security using agent technologies

10) Data quality monitoring - ability to check the completeness and integrity of data accumulated in the database

Detailed analysis of the features of smart technologies can serve as a subject of a separate research paper. This paper discusses only one of the smart technology features – automated version updating.

## 3 Automated Version Updating

It is a common practice in IT industry to pass (accept) the latest software version in the test environment before it is introduced in the production environment. However, in cases when system is used on many workstations and/or on several scattered servers, no warranty can be provided that this version will function problem-free for the following reasons:

1) each workstation and/or server contains individual data including e.g. personalization values, user rights, and even sophisticated definitions of processes
2) each workstation and/or server may be equipped with a different environment - different operating system, different versions of system database administration, file layout, regional settings, and other external systems operating parallely to the new software version

The first of proposed criteria is related to the field of delivering and installing (distributing) software. Information system equipped with smart technology is able to analyze the environment which it is put into from viewpoint of standard and specific parameters.

As standard parameters are supposed to be, for example, the operational system, the data base management system, browsers etc. used on the concrete server or workstation.

The specific parameters include checking of the evidence of previous (possibly damaged) software versions as well as evidence of other specific solutions on the workstation.

Automated software version updating includes both, external and internal stability criteria, since the environment analysis is performed before software version update and functionality self-testing after installation of version update.

Full cycle of automated version updating includes sequenced actuation of the following mechanisms in compliance with smart technology principles:

1) environment test
2) system back-up administration
3) downloading and installation of software version
4) data migration

5) self-testing
6) system renewal in cases of failure

### 3.1 Environment Test

The idea is taken from the real world which shows the ability of living beings to adjust themselves to specific conditions. Environment test includes the analysis of specific parameters by benchmarking their values and adapting functions to pre-defined decision-making flow.

In a similar way the software equipped with the smart technologies should be able to analyze versions and other configurations of the operating and data base management systems according to the requirements of the software, and react adequately to inconsistencies by generating necessary messages to users.

Although at the first glance, the task seems trivial, in fact the environment analysis (its compliance with the requirements) and subsequent decision-making (choice of the optimal way to adapt) are the most crucial tasks. In effect this analysis is much more complicated, because the version number and some of the configuration only partly determine the ability of the given application to fulfill their functions. An appropriate reaction depends on external factors, e.g. availability of internet connection to the necessary resources etc.

One of the solutions [5] aimed to simplify the analysis of an external environment is introduction of software application requirement passport that is created during the software development process in which the requirements against the environment are fixed: for operating system and other of its components, detailed on the level of object classes, of DLL's and of others - .ini-files, registry entries, location of files and folders, regional and language settings, workstation settings etc. The creation of the passport is an obligation for the developer; it happens by generation of the passport from the development environment in which the application is created and is able to work. The created passport is integrated into the software.

After implementation of the application in the production environment a module prepared by the developer compares the production environment parameters with the passport parameters (analysis of environment).

Differences between target environment parameters and values in passport may cause various reactions depending on adaptation mechanism. The most trivial reaction is when the user is notified about the differences and further steps necessary to correct environment configuration („Please, change regional settings").

High developed solutions are equipped with mechanisms aimed at collecting the missing components, e.g. automated component update is done from producer resources. These mechanisms are also able to reconfigure the environment according to the requirements [6].

Such automated mechanisms involved in environment adaptation also may seriously endanger operations of external software solutions. Therefore, it is reasonable to include into solution a compliance passport, a type of passport similar to the requirement passport that is delivered together with the software. This compliance passport fixes and indicates compulsory requirements of the external systems and parameters eventually conflicting with the software. The first application of smart technologies is related to the field of delivering and installing (distributing) of software. Information system equipped with smart technology is able to analyse the environment which it is put into from viewpoint of standard and specific parameters. As standard parameters are supposed to be for example the operational system, the data base management system, browsers etc. used on the concrete server or workstation.

Specific parameters are checked for evidence of previous (possibly damaged) software versions as well as evidence of other specific solutions on the workstation. A typical example of a dangerous software - “neighbour” are antivirus solutions which can classify smart technology software as a virus and even block it up.

Similarly, as for installation of a new version SW users should have a reverse link to SW developer. Due to this reverse link developers can receive complete information on performance, including failure reports and statistics on activities that would allow developers to improve SW quality. As a rule including of above mentioned features into SW requires components that are functioning throughout the whole life cycle of software and are considered as the extension of core functionality that sometimes client remains unaware.

## 3.2 Installation of Software Version

The solution and installation of the new software version highly depends on architectural design.

Traditionally, web applications are the most uncomplicated in terms of installation, for their operation they use standardized Internet browsers for data illustration. In fact, at least three problems refer to updating process of web applications. They are as follows:

1) **Partial conformity of browsers**. Development of Internet and other browsers is a natural process that cannot be influenced by system developers. Therefore, in practice it is impossible to ensure identical functioning (at least display of information) of all available browsers.

2) **Decentralized data storing**. Companies and public institutions usually do not store their data in centralized system but on individual servers; therefore software and its modules have to be developed and distributed on many mutually independent servers.

3) **Different settings on local workstations**. Personalization as a solution for increased user comfort becomes more and more popular, however conflicts

with the data centralization idea. Additional difficulties arise from different regional settings, that are necessary in exploiting different systems, e.g. decimal separator, date format).

Furthermore, the paper deals with client-server software [7, 8] version distribution due to importance and sophistication of the proposed solutions over centralized web applications.

The installation of the latest software version is done automatically through the Internet and includes the following steps:

**1) Installation of the latest version on the server**

The latest software version together with the parameter file containing numbers of the latest version of system components and parameters necessary for system operation are placed on centralized server.

In order to enhance version downloading the system is dividend into components, which are relatively independent parts of the system and can be updated independently. For each system component the number of the latest version as well as implementation parameters are recorded in the parameter file.

If the latest version of one of the components is dependent on the latest version of other component then numbers of for both components are increased in the parameter file.

**2) Identification of parameters of the latest version**

At the moment when a user from the local workstations connects to the system, a built-in mechanism turns to central database and downloads the parameter file of the latest software version.

**3) Comparison of software versions**

The numbers of the latest version of system components are compared with the numbers of component versions on local work station. For those components whose workstation numbers are smaller than the number on central server, downloading of the latest versions should proceed.

**4) Downloading and installing new components**

The system downloads the necessary batch of the latest software versions and installs them onto local workstations. During the installation, conformity of the operating system with the parameters indicated in file is controlled. The successfully finished installation has the fixed parameter file on the local workstation by equalizing the version of installed components with the number of installed version. If for some reason downloading or installing  of the latest version has failed, version numbers of components in parameter file are not modified, respectively updating of components will be repeated next time when system is turned on.

## 3.3 Data Migration

Software modifications are often caused by changes in database structure, settings, exchange formats etc.

Data migration as an integrated part of automated version updating mechanism can be analyzed from two aspects: first, matching data structures to the new software version, and second, periodical data synchronization among local workstations (sometimes might work off-line) and central data pool.

Adaptation of data structures to the new software version is ensured by special scripts that are distributed together with the latest software version and activated immediately after software installation. Frequently, complementing data structures is related to complementing historical data with new data, e.g. default, or data migration (structural changes). Though this is relatively simple task, data synchronization is a much more complicated task.

Data synchronization represents a much more complicated task. Data synchronization comprises the main problem when software has client-server architecture and work is ensured in both off-line (on workstations) and online (connecting to the central server). Within data synchronization a threefold task should be solved: first, problem of unambiguous object identification, second, rollback functionality, third, automated synchronization of conflicting and inconsistent data.

One of the most commonly used solutions is extraction of identifier segment, where objects are identified by belonging to a certain instance. If solution is run online, all objects are stored centralized. If, for some reasons running of system online is not available or not used, the system automatically switches to working offline and ensures data storing on local data base instance. After online connection is established, software automatically performs data synchronization by referring to object identifiers and object creation/modification timestamp. During synchronization, instance of data creation (local working station) is identified by using object interval. Then values stored in central database are compared with the actual values taken form local databases and if necessary outdated values in central database are substituted with the actual values.

The specific data migration is needed in those cases when personalization for each software user is provided. If personalization is kept on each individual workstation, then installation of the latest software version initiates migration of the personalization parameters to the new version. If personalization is centralized, the migration should be done on server level. A very characteristic example of personalization is user and rights administration, which is an integral part of any system.

In case of smart technologies migration of the personalized data (user and rights control) is automatically done by internal component, thus relieving user form working with incomprehensible scripts.

## 3.4 Self-Testing

A very important feature of smart technologies is self-testing. Like living beings who can control themselves and are able to understand the limits of their possibilities, a smart software must be able to check itself before it is run [9,10]. In the hardware industry it is a usual feature that the equipment tests itself as soon as it is switched on. In the case of smart technology software the self-testing means the ability of software to run predefined set of test cases in the production's environment automaticly to check its status.

Functionality of self-testing contains two basic components:

1) Test cases of critical functionality (data component)
2) Built-in mechanism (software component) providing automatic running of test cases

The critical functionality is a set of system functions which are substantial for using of the system. It is impossible to use the system valuably if these features do not function. The critical functionality covers all basic functional requirements of a system – for instance, calculation algorithms, workflows etc. - but does not include non-functional requirements like performance, navigation, and others. To implement self-testing functionality the developers should prepare tests that are appropriate to the selected testing parameter [11] (e.g. full set of test cases [12,13,14]) covering all critical system's functions and incorporating them into the software and the database/ file system.

In addition to that an automatic test running and comparing of results with benchmark values should be implemented [15]. The most important feature of self-testing is an ability to test the software in the productive environment using real data in the read-only mode without changing data entries in production database.

The key feature that differs self-testing from conventional testing [16] is running of self-testing in the productive environment, including an access to the real database of system. The self-testing should be executable in nearly every moment of the system's functioning without disturbing users.

Moreover, each self-testing session includes regressive testing accordingly to the accrued test data. Along with changes in the system a test set and benchmark values are modified/improved accordingly.

System self-testing differs significantly from the traditional testing processes passed during the development phase and performed by the group of independent experts. Self-testing involves developer, thus ensuring that developer's work is not only the written source code but also the result of source code and module tests that ensure self-testing function. Regarding the accrued tests the developer has to prove conformity of his/her work results with the requirements. The need for this procedure appears especially when problems occurred in production environment can not be transferred to the test environment.

Role of self-testing in software development life cycle differs from the role of testing in traditional life-cycle, as for example in model „V”. The self-testing in software development life cycle has double application - first, it is a tool for developer to check system operation before testing by the group of independent experts. Second, it indicates readiness of the system tested in production environment.

Though self-testing suggests an increased possibility that the delivered software is a high quality and reliable, it should be emphasized again that implementation of self-testing function requires additional development efforts as well as designing of specific architecture.

## 3.5 Back-up Administration

Automated version updating modifies application set, environmental settings as well as database structure and even data. Therefore back-up administration should be integral part of the automated version updating mechanism.

The above mentioned system components (application, database, environmental settings) should be provided with two following groups of functions:

1) Back-up generation

2) Roll-back functionality to ensure recovery of prior state in case of failure

Back-up generation is a relatively simple task – before downloading the latest version into special directories software files are copied and databases back-up copy is saved. Also a file with environmental settings is filled (conform to a software requirements passport described in previous sections).

Roll-back functionality is a much more sophisticated task, since, first, due to limited hardware ability duration of historical records is finite, second, recovery of environment to the prior state is not always possible just by modifying standard settings.

Important part of roll-back functionality is detection of failures, respectively, set of unambiguous indications that signal failure in installation of the latest software version and necessity to recover to the prior state from back-up files. Moreover, due to the limited ability of keeping historical records it is absolutely crucial that generated and available back-up files are created from feasible software version not from the previously failed installation efforts! The analysis of failures, especially, the set of indications signaling capability or incapability of the software is a subject of separate research paper. In most simplistic solutions all components are checked and main application window is opened without getting to the error handling routine.

In our proposed solution the roll-back functionality ensures repeated downloading and installing of the last successfully installed version including reconstruction of the appropriate database structure. Unfortunately, in this case all the data accrued after the last successful installation are endangered; respectively they are saved in modernized data structure though by incomplete software version.

## 4. Use Cases

The above described principles of smart technologies are implemented and approbated in software development and implementation projects [8,18,19]. However, in neither of software products smart technologies were implemented to the full extent. These software products support only several features; furthermore, functionality of most of these features is narrowed.

Henceforth, the paper deals with the results of particular project, referring to automated version updating.

The task of the project was to develop software that would ensure structured gathering of financial information on several hierarchal levels. The system was supposed to run in 600 public offices located throughout the territory of Latvia ensuring regular gathering of information by different time periods. It was required that the system should be able to run not only in offices provided with Internet connection but also in offices with irregular and instable Internet connection or even in offices without Internet connection. The requirements suggested remote software installation and very limited financial and human resources were allocated for maintenance (user support) services. Furthermore, the specific requirements dictated that in certain periods most users will use the system simultaneously.

An application that provided all required functions has already been developed; therefore project task was to adapt application according to the requirements of automated version updating. Additional human resources for the system adaptation accounted for c.a. 10% of the initial system developing resources. The most part of these 10% were invested into extensive testing and code review. In order to ensure reliability of the developed mechanisms, software testing with different infrastructure configurations was performed on virtual machines.

The achievements and results of this work have been used successfully in all local governments and many public institutions on different levels of hierarchy in Latvia since 2005. Automated version updating has significantly improved system maintenance and reduced the need for user consulting resources. We believe that the idea has proven to the successful and we have continued implementing it in several other projects.

## 5. Indications for further research:

1) Implementation of smart technology principle in software takes fewer resources than full-range configuration support. At the same time, smart technology places fewer constraints on the acceptable means of expression.

2) Smart technologies allow reducing the efforts for software testing and setting up, thus increasing the client service level significantly.

3) Smart technologies assist to provide software performance in a changing environment and environment containing heterogeneous platforms and infrastructure. Nevertheless mechanisms of smart technologies need regular adaptation to the environment changes, especially in case of standardized software. It is very important to provide in-depht reporting mechanism to inform the developers about indicated problems in time.
4) Adding smart technologies to the software after the development is useful though requires more resources than including smart technology already in the software architecture design phase.
5) Although the clients approve opportunities provided by the smart technologies, usually, they are not willing to provide additional financial means to ensure them. The smart technologies are certainly costly – effective and should pay-off in long-term business projects and long-term cooperation with client when the number of users exceeds 10 or if workstations are configured differently. Opportunities provided by the smart technologies pay-off and enhance software distribution and user support even if a company has only two geographically separate subdivisions.

## Acknowledgements

The research is supported by the European Regional Development Fund (ERDF).

## References

[1] Bicevska Z., Bicevskis J.: Smart Technologies in Software Life Cycle. In: Münch J., Abrahamsson P. (eds.): Product-Focused Software Process Improvement. Lecture Notes in Computer Science, Vol. 4589. Springer-Verlag, Berlin Heidelberg (2007): 262-272

[2] Roger S.Pressman, Software Engineering. A Practitioner's Approach. Sixth Edition. McGrawHill. 2005

[3] Roger Laddaga, Paul Robertson *Self Adaptive Software: A Position Paper*, International workshop on Self Adaptive Software Properties in Complex Information Systems, 2004, Bertinoro, Italy

[4] Qianxiang Wang Towards a Rule Model for Self-adaptive Software ACM SIGSOFT Software Engineering Notes, January 2005, Vol. 30, N. 1

[5] Rauhvargers K. and Bicevskis J., Towards a semantic execution environment testing model (this volume)

[6] Beydeda, S.: Research in Testing COTS Components - Built-in Testing Approaches. In: ACS/IEEE 2005 International Conference on Computer Systems and Applications. Bonn, Germany (2005)

[7] Andzans A., Mikelsons J.,Medvedis I. And others. *ICT in Latvian Educational System - LIIS* Approach, Proceedings of The 3rd International Conference on Education and Information Systems: Technologies and Applications, July 14 - 17, 2005, Orlando, Florida, USA

[8] Latvian Education Informatization System – LIIS [on-line]. Available on the internet: http:/www.liis.lv

[9] Sami Beydeda: Self-Metamorphic-Testing Components. COMPSAC (2) 2006: 265-272

[10] Beydeda, S.: Research in Testing COTS Components - Built-in Testing Approaches. In: ACS/IEEE 2005 International Conference on Computer Systems and Applications. Bonn, Germany (2005)

[11] Boris Beizer. Black-Box Testing Techniques for Functional Testing of Software and Systems. John Wiley & Sons, Inc, USA, 1995

[12] Bicevskis J., Borzovs J., Straujums U., Zarins A., Miller E.F. SMOTL - A System to Construct Samples for Data processing Program Debugging. IEEE Trans. Software Engineering, 1979, SE-5, No.1

[13] Bicevskis J., Bicevska Z., Borzovs J. Regression Testing of Software System Specifications and Computer Programs. Conf. Proc. Quality Week, San Francisco, 1995, Software Reserch Institute.

[14] Bicevskis J. The Effictiveness of Testing Models. Proc. of 3d Intern. Baltic Workshop "Databases and Information Systems", Rīga, 1998

[15] Hans Buwalda Essentials of Testing and Test Automation. Proc. of 15th Quality Week 2002, San Francisko, 2002

[16] Surya Kumar Role of Test Tools in Product Testing and Automation. Proc. of 6th ICSTEST Conf, Dusseldorf, 2005

[17] A.Auzins, J.Barzdins, J.Bicevskis, K.Cerans, A.Kalnins. Automatic construction of test sets: theoretical approach. Lecture Notes in Computer Science. Vol. 502, Springer - Verlag, 1991.

[18] [on-line]. Available on the internet: http://www.numuri.lv

[19] [on-line]. Available on the internet: http://www.liaa.gov.lv

LATVIJAS UNIVERSITĀTES RAKSTI. 2008, 733. sēj.:
DATORZINĀTNE UN INFORMĀCIJAS TEHNOLOĢIJAS 38.–52. lpp.

# Towards a Semantic Execution Environment Testing Model

Krišs Rauhvargers, Jānis Bičevskis

University of Latvia, Raiņa blvd. 19, Rīga, Latvia
Kriss.Rauhvargers@bank.lv, Janis.Bicevskis@lu.lv

**Abstract.** The paper analyzes one component of "smart technologies" – a model for program execution environment verification that employs software meta-data descriptions of quality requirements to ensure the conformity of characteristics of the surrounding environment to those necessary. The study is based on practical software deployment and maintenance experience in areas where the production environment is inadequate and defies normal software operation. The solution is to develop a "profile" for each software item which would contain information about software requirements regarding its execution, for instance, OS version, configuration file and registry entries, regional settings, etc. The profile document is added to software deliverables together with a set of tools capable of verifying the adequacy of the execution environment according to the document. The profile document can be used in both the installation and operational phases of software.

**Keywords.** Maintenance, Testing, Smart Technologies, Self-healing systems.

## 1 Introduction

As new computing paradigms, such as distributed computing, service-oriented architecture, and business process support are emerging, software becomes more complex and more difficult to maintain. Quite much of the software being developed today is aimed to serve a single business and the customers are willing to invest as little funds as possible. Therefore, the software is built for a specific environmental platform and may have strong bonds to it.

One of the core principles of developing "Smart Technology Compatible Software" [1] is to create software that is able to analyze the external environment and, like a cognitive being, adapt to it or at least to state that the environment is not suitable for normal existence. If this kind of software were feasible, the installation process and maintenance would be greatly simplified.

The present paper analyzes the requirement stated in [1] and provides a methodology that, when properly implemented, solves the problem. Methodology discussed in the paper is based on describing knowledge about software dependencies on external environment outside the executable code, and creating a human and computer-readable document – a software profile document, based on the collected data. Using the profile document, environment may then be checked during initial software deployment and later in the software life-cycle.

The paper is further organized as follows: the next chapter describes the background of the research and states the research questions. Chapter 3 discusses work in related directions of software engineering. In chapter 4 we present our methodology and aspects to consider when implementing the methodology. Chapter 5 is a short report on first practical experiences already achieved.

## 2 Background

When a company reaches certain size or functional business area coverage, it may need custom-tailored software applications to be able to handle its business functions. Independently from the chosen software platform, there is a selection to be made – whether to outsource the development from other company or to develop the applications in-house. The former allows to "keep out of IT" by not requiring much involvement of company's staff in the development work, while the latter allows to be more flexible in specifying requirements and maintenance since the software code is available and can be modified upon request.

The historical basis of this paper is formed by an in-house solutions development and maintenance model where security and safety are highly significant. Because of such priorities in this model, the software is being "pushed" through testing and acceptance-testing environments before it may be installed in "live" environment. To install software at a particular operational environment (that is, any environment other than development), a special code compilation is carried out for the particular environment and then it is installed manually according to instructions given by the developers.

The manual installation model is reasonable in the particular case, since the software is designed to be installed easily and the software applications are not distributed to many users. To be more precise, in the client-server architecture, the client components are developed somewhat like "portable applications", i.e. applications that do not need client-side installation at all and are fetched from the server for execution. However, the server side of the applications has to be configured and installed manually.

All the instructions for setup and installation are prepared in textual form by the developers and are executed by people authorized to do so (systems administrators). The instructions typically include tasks such as compilation, copying executable files and libraries to servers, registering and configuring components, altering operating system configuration.

The instructions also contain information that can be treated as requirements regarding the execution of the particular piece of software – needs for other specific software items to be installed, configuration settings, file locations, specific tasks to be performed before the installation – to be summarized as "execution requirements".

The execution requirements hold through the lifetime of the particular software application, i.e., they must be satisfied whenever the software application is executed.

Upon software migration to another physical or logical environment – for instance, moving to a newer server – a series of questions arises. For instance, what are the current execution requirements of system? What else directories should we re-create in the new environment? Which configuration file does this system use? Did we have to open any specific ports in the network firewall for this system?

## 2.1 Systems Interaction

In realistic environments, different software applications may be installed in the same computer. The execution requirements may differ from system to system and some of them may be contradictory. For instance, system A may require date format setting as specified by ISO 8601:2001 (yyyy-mm-dd) and, at the same time, system B may rely on American English date format (mm/dd/yyyy). Of course, work-arounds exist for the situation described, if only the problem is known during the installation of whichever system is installed later.

Software systems' integration may be necessary to avoid data and functionality duplication in systems. For instance, the organization's customer data may be shared between the CRM (Customer Relationship Management) software system as the "host" of the data and other "guest" systems such as inventory system, accountancy system etc. The integration may be carried out by using public interfaces of the hosting system – an API (Advanced Programming Interface), if such exists or by using internal mechanisms of the host system. The latter solution is technically possible only in the in-house development model and may sometimes be used.

Of course, the interface that the host system provides and the guest systems are dependent of may change by time. It is not a very likely scenario in the case of integration using public API, but quite likely when private interfaces are exploited. This leads to unpredictable effects at maintaining the guest systems when updates for the host system are deployed – in-depth regressive integration tests are required.

## 2.2 The Research Question

In many systems, typically those designed to support unique business process instances rather than process outcomes (documents); it is not possible to perform a "test run" of the system to verify that the system still works. Such tests could have unpredictable effect on the business process. Hence, if business processes occur rarely but are significant to their owners, there is a need for methods to verify the software system configuration and to check if it is up-to-date without running the business support applications.

Automated software installers are another option for software deployment. However, neither the maintenance model described here, nor the automated installers provide a way for checking if execution requirements of different systems are not contradictory and if all requirements are satisfied. The easiest known way is performing manual checks of all requirements. Of course, in the real world it may appear too time-consuming.

Installation package wizards such as Macrovision InstallShield or NullSoft NSIS offer features for checking pre-installation requirements. Also, they may provide a "repair" feature if the system is known not to be malfunctioning. However, this does not allow performing a preventive checking to find out if the configuration is still acceptable.

The present paper is a study on formalization of system execution requirements taking into account the semantic nature of every requirement.

The main questions of research are:

- Is it possible to automate the verification of execution requirements?
- If the process is automated, can the requirements still be human-readable?
- Can the verification process be unified throughout the enterprise?

## 3 Related Work

The field of automated software testing has been studied extensively; however mainly research concentrates on testing the software internals, trying to verify that the software is built according to the specification. Formal verification frameworks such as Context UNIT and Mobile UNITY [2] have been described as well as practical implementations such as Java PathExplorer [3] are present.

The concept of an execution requirement indicates that the system under test is not aware of the particular circumstance – it will work fine if the conditions are met. The software system may have a built-in mechanism of self-protection, but it is not intended to adapt the situation.

The present research concentrates on aspects of the execution environment that the system is not aware of.

### 3.1 Self-Healing Systems and Built-In Tests

The topic of self-healing systems can be considered relatively close to the research topic of this paper. Both research topics share the same goal – a system working at an operating environment that is known not to be perfect.

A comprehensive study in the field of self-healing systems by D. Tosi [4] identifies and analyzes the key elements of a self-healing system.

The concept of self-healing software is defined as a system that monitors the surrounding environment at run-time, detects failures and, knowing its “normal” way of operation, can “heal” itself. Such a system can then be called a “fully autonomic system”, which is, according to classification of [5], the highest (5th) level AS (Autonomic System). Such systems are far beyond the scope of current research which concentrates on practical support for transition between 2nd level (Managed Level[1]) and 3rd level (Predictive Level[2]) of autonomic systems.

---

[1] Managed Level: At this level, system management technologies can be used to collect information from different systems. It helps administrators to collect and analyze information. Most analysis is done by IT professionals. This is the starting point of automation of IT tasks.

[2] Predictive Level: At this level, individual components monitor themselves, analyze changes, and offer advice. Therefore, dependency on persons is reduced and decision making is improved.

In D. Tosi's paper [4], the self-management objective is decomposed into a number of dimensions: requirements (security, performance), monitoring/detection (includes self-adaptation, self-optimization, and self-healing), and repair. According to this decomposition, the current paper is a research on monitoring and detection, specifically on system's reaction to changes in the runtime environment.

The author of [4] also mentions the need for a knowledge base that describes the "normal" environment of the system.

In [6] Wang et al. present a concept of software component with built-in tests. A built-in test (BIT) is a set of code functions that perform verifications, if the component is working as predicted. Authors of [6] suggest that a component should contain one or more BITs that can be executed; and implement some specific interface to be called when the system is run in maintenance mode. Here, the authors define a concept of "maintenance mode" [6] for a software system, which is proposed as a special execution mode of the system when built-in tests can be activated but the business functionality of the system is not touched.

The authors of [7] propose a framework named Component+ that mainly focuses on benefiting from the use of BITs in software components. Authors note that every component implements its fixed interface which other components may rely on. This can be called a „contract" between components. Authors of the paper suggest that BITs are used for testing, whether the component is capable of serving the defined interfaces, i.e. if contracts between components are not violated. It is remarkable that the authors mention the importance of „Quality of Service" (QoS) testing for verification of the operating environment, but their proposed model concentrates on testing the contracts between the components, hence, QoS testing is a „by-product".

## 3.2 Unit Testing and Test-Driven Development

At a first glance, the test-driven development [8] seems to be a good solution for the research problem. Though unit testing technique has a great effect at improving software quality and results at integration testing, trying to use the unit tests for validating the surrounding environment would be a misuse of the particular technology. This is because unit testing is designed to focus on testing a single unit. According to the guidelines, a unit test shouldn't pass the boundaries of the unit to be tested. To overcome the need for "outside world" reactions during tests, the unit testing model proposes the use of mock objects and fake objects [9]. Both kinds of objects are used to simulate the interface of external units required by the unit under test. The use of such substitute code indicates that unit testing may not be useful for integration between components, and, moreover, integration between the system and the surrounding environment.

## 3.3 Similar Ideas in Hardware Appliances

A similar topic has been investigated quite profoundly in hardware engineering. Runtime checks of the surrounding environment seem to be natural and have been studied extensively in hardware world. Research is being carried out in different areas – both on self-tests of internal state of the item, for instance, for a CPU or a

RAM module, during the stand-by time [10], and on overall inspections of the surrounding environment, i.e. parts of the "hosting" system. A system embedded in a car is a good example. Upon initialization, the central processor performs a comprehensive examination of all the distributed sensors and sub-processor units; if any of them fails, it shows a warning sign to the driver or even does not allow starting the engine.

Similarly to software systems, nowadays the hardware systems have also become component-based. For instance, a digital photo camera relies upon a specific type of interface for its memory chip (e.g. Compact Flash type I cards supported only). Upon initialization, the camera checks whether the card is currently available, the type and the capacity of the flash card. The user may be prompted "Error, wrong card inserted!", "CF full". This can be compared to the "contract testing" described in [7].

The idea of the current paper is similar to the methodology described here - verification if the external components the system under test relies on, fulfills their duties.

## 4 The Proposed Model

### 4.1 Model Summary

Methodology proposed by this paper is based on describing knowledge about the software system outside the executable code (as some sort of meta-data about the system), and using the collected data for creation of a computer document – a software „profile“. Such document is later used for execution requirement testing.

A software requirements profile document contains listings of both the internal links between software components and dependencies on external services, facilities and interfaces. Using appropriate tools, the profile document can be employed for different purposes throughout the life-time of the system. The present paper describes the usage of software profile document as the core of execution environment verification by external verification modules – small pieces of software aimed to perform verification routines (further in text – EVM), but other applications may exist.

The following model is offered:

1. bonds of the program code with the environment are registered as software meta-data during development;
2. upon finishing development of a particular software item, a software profile – a document summarizing execution requirements for the particular item is generated from the code meta-data;
3. the software item is delivered into other execution environment together with its profile document;
4. conformance of the execution environment is verified using data from the profile document.

A summary of the model is shown in Figure 1

The described situation applies to internally stable [1] systems. That is, the system under test has already passed the integration tests in the development environment and is known to be working under certain circumstances.

There are two areas of software life-cycle where the software profile methodology is useful – the initial verification of execution requirements during software installation into an environment where the software has not yet been used, and the routine re-validation of run-time requirements every time the software application is executed. Re-validations take place in the same way as initial validation; the only difference is that the user is not prompted for initiation data.

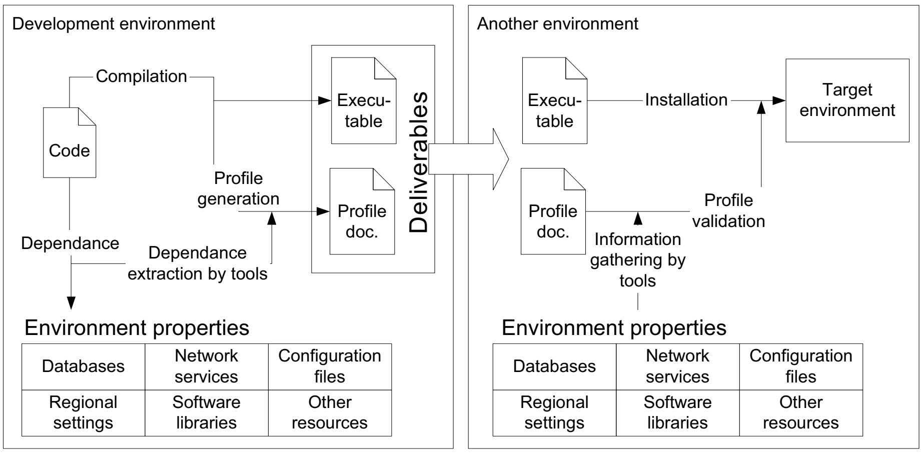


**Fig. 1.** The Tool-Driven Process of Software Profile Generation it the Development Environment and Use in Other Environments.

## 4.2 Motivation for the Proposed Model

The model proposed in the paper helps overcome several shortcomings of other related technologies that could possibly be used to achieve the same goals.

When compared to built-in tests, the software profile methodology has the advantage of test descriptions not being encoded in the system itself. As a result, the knowledge is not hidden from software maintainers and the list of known requirements may be supplemented without recompiling the software.

The contents of software profile document should rather be treated as descriptions of software properties than test descriptions, and hence the tests can be more flexible.

It is possible to execute different tests per single requirement to verify different aspects of the requirement. It is also possible to replace the test algorithm if and when needed. The scenario is quite likely since the properties of environment may change. For instance, the software may require that "Outgoing TCP traffic on 80 shall be allowed on the computer". Verification of the requirement in a programmatic manner relies heavily on the kind of firewall software installed on the particular computer and hence, upon changing the firewall software, the EVM should also be replaced with an updated version.

## 4.3 General Architecture of Runtime Validation Framework

As recommended by [6, 7], runtime validation should not interrupt the regular work of the system under test. Our model conforms with the thesis completely and is designed to separate not only the regular and maintenance modes, but also the executable code.

The software profile approach is different from a typical BIT architecture in a way that the tests are actually not embedded in the system. To enable a component for self-testing of the execution environment, only functionality for loading the software profile validation runtime („loaders") have to be encoded into the system.

When software is run in maintenance mode, the loader functions are used for loading the verification runtime core and handing the execution control to it. As the core is loaded, it looks for the software profile document of the system under test and analyzes it. Knowing the EVMs currently available for testing specific requirements, the core parses the software profile document and invokes an appropriate test routine using a specific EVM (or multiple modules) on each requirement listed in the document.

Hence, the model relies strongly on the dynamic loading feature of the execution environment. The feature is first employed to load the verification runtime core module and later, to load the specific verification tools. The required effect can be achieved easily in today's execution frameworks - using the reflection and dynamic load feature of .Net framework, using the ClassLoader interface of Java, even easier in scripting languages such as PHP or Python.

## 4.4 Execution Requirements and External Verification Modules

In the context of the paper, an execution requirement is a verifiable demand statement about the execution environment that typically has some human-understandable meaning and that holds through the lifetime of the system. For instance, a requirement may be formulated as "TCP traffic to host 192.168.1.1 on port 21 must be allowed" or "Write access is necessary to directory /home/$USERNAME".

Other typical types of execution requirements include:

- Other components – versions, names, availability. An application typically depends on one or more external code libraries to be available - for instance, XML support, specific database drivers, MS Excel object model API etc.
- Configuration files – INI files, Windows registry, .Net framework configuration files. The requirements of this kind typically ask for a configuration file to be located at a specific location or for values to be set for specific keys.
- File system access – requirements regarding existence or non-existence of specific nodes in the file system, permissions on file shares, etc.

- Network dependencies – requirements on protocols, ports, required network locations to be available.
- Relational data bases – requirements, specific to DB vendor, defining a dependence of the software system upon a certain data base. The requirements may include database locations, names, and requirements for existence of certain DB objects (tables, stored procedures, functions) or even the interface definitions of DB objects. For instance, one may require that a table "CUSTOMERS" having a field "NAME" exists in the database.

This list can be continued and is by no means limited to kinds of requirements listed here. Nearly every technology used in contemporary software development has some properties that may be significant for a software component employing the technology. For instance, both Microsoft's COM+ or Java's Hibernate (and of course, other products, too) services allow the configuration of distributed transactions' isolation level to be set declaratively [11, 12]. In both cases, a component relying on the particular technology can claim for a specific setting for distributed transaction configuration. Such a claim can be formulated as an execution requirement.

It is also advisable to define business system-specific requirements, that is, requirements that are useful only in the context of the system under test. For instance, "at least one administrator account must be present in the users' registry" or "all `inbox` folders of the system should be user writable" are system-specific, as the concepts of "administrator account", "user's registry" and „inbox folders" achieve their semantic meaning only in the context of the system. One should also provide EVMs that support verification of such requirements. This kind of EVM is actually a built-in test that is externalized from the system, but it is more concerned to checking if other components are in order rather than checking if the component itself is fulfilling its contracts. The verification modules used for this kind of tests can be bundled in the same code assemblies as the business system, and hence be versioned together with the system. Such approach allows the tests focus more on the internal stability [1] of the system and complies well with the component built-in testing as described in [13].

A specific kind of requirements is the transitivity requirement which can be read as "I will do my job if some other particular component does what it is supposed to do"; that is, a requirement for another component or application to pass the verification process successfully. For instance, in the client-server model, client components may not be able to perform their duties if the server component is not configured properly.

To complete the list of vital elements of the software profile framework (hereinafter - SWPF), one must mention the EVM – external verification modules.

An EVM is a small functional component that encapsulates logic for verifying one or a few execution requirements. Technically, an EVM looks for evidences in the execution environment and hence decides if the particular requirement is satisfied. The nature of an EVM should be similar to the way humans would verify the requirements – first analyzing the requirement and deciding what evidences are required to be sure that the requirement is fulfilled, second – performing the verification. Different kinds of evidences may exist and it is a task for the EVM developer to decide which ones are satisfactory.

For instance, when developing an EVM that verifies if a specific Windows security patch is installed in the system, the first idea would be to use Windows Management Instrumentation[1] to read the list of all patches installed and then look through the list to see if it contains the name of the needed patch. However, this may be a bit tricky, because WMI may not be installed on the particular computer or may not itself be updated and hence the specific function for reading the patch list may not work. When looking for another approach, one will notice that every patch installation results in a new "uninstall" directory created in Windows installation directory, for instance "$NtUninstallKB825119$" where KB825119 MS is a knowledge base number for the patch. Hence, one may consider that the existence of such a directory is a good enough evidence for validation.

A clear distinction must be set between validating a requirement in whole and validating the evidences for a requirement. There exists an N:N type of relation between the two (validating a requirement may mean looking for N evidences, a single evidence may refer to different requirements). The resolution of requirements into evidence searches is a task of the SWPF core, but technical implementation details are out of the scope of the present paper and are a question of further research.

## 4.5 Kinds of Data Employed by the SWP Framework

To achieve the functionality described in the previous chapters, different kinds of data are required for the software run in maintenance mode.

The most obvious knowledge base is the software profile document itself. It supplements the system under test and is delivered into particular execution environment together with binary deliverables or source code. The knowledge – listings of execution requirements - is first summarized by the developers and can later be complemented by other parties involved. The data format for storing requirements is not dictated by the methodology and may vary upon implementation; however a possibility to store complex data structures is essential. Other requirements for the description language are modularity and possibility to extend the available markup as new kinds of requirements may appear. Hence, we propose the use of XML as the carrier and XSD as a validation tool and reference. An in-depth study of the format is a question of further research.

Another knowledge base employed by the methodology, is the „inventory" list that belongs to a particular execution environment. The list contains information about the EVMs that can be used at the environment. Since the EVMs should be designed to handle verification of requirements as specific as possible, the full spectrum of EVMs may be quite ample. For instance, it includes a wide variety of EVMs that handle requirements specific to a single business system only (e.g., an EVM for a requirement "Version 2.9.4 of the HR system shall not be present at the environment"). Not all the EVMs owned by the organization may be needed at every particular environment and hence "inventory list" should be bound to the environment.

[1] Windows Management Instrumentation (WMI) is the Microsoft implementation of Web-based Enterprise Management (WBEM), which is an industry initiative to develop a standard technology for accessing management information in an enterprise environment (MSDN library, 2007)

Another task for the inventory list is to tell the verification runtime, when to use the particular EVM described in the list. That is, the list describes patterns to look for in the software profile document that are related to the particular EVM. In the XML-based form of software profile document, the names (and the namespaces) of the XML elements recognized by the EVM should be depicted in the inventory list.

## 4.6 Creating a Software Profile Document

The software profile notion is essential for systems which are developed in a different environment than the operational one (where the concept "operational environment" includes development, testing and/or acceptance testing environments, production environment).

It is assumed that the system under test is internally stable, i.e. that system developers have assured that the system is functioning and have managed to run the system in the development environment. This should be true in order to allow transition to the testing phase. Since the satisfactory requirements are met in the development environment, it can be a good sample for gathering the requirements.

The gathering of data required for generation of the software profile should be begun as close to the beginning of information system life-cycle as possible in order to minimize the documentation work to be done close to the delivery date.

A good time span for registering requirements is the coding phase when detailed system design specification is transformed into executable code. All kinds of dependencies become known to the developers during this phase: the ones dictated by the business problem (declared in the requirements specification document), the ones discovered during system planning (functional specification, class, and component diagrams) and the technical limitations that have arisen due to development methods, organizational standards, etc.

The methodology anticipates that requirement descriptions needed for the creation of the profile document are gathered from source code where it has been previously entered by the coders.

A substantial part of dependencies is known even before the coding phase i.e., during the design phase. If software is developed using model driven development approach, requirements can be recorded at an earlier phase than development. Dependency information could be attached to the model as object stereotypes if a UML model is used. During the PSM (platform specific model [14]) transformation to program code, the stereotypes would be transformed into code meta-attributes as described further in this chapter. Hence, the initial dependencies would be recorded even before the coding phase begins.

.NET framework provides a convenient way for describing code meta-data. It is called "declarative attributes" – a supplementary information block that is assigned to a particular code class, function or the software assembly all together. The .NET framework allows defining one's own attribute classes to extend the set of available declarative attributes.

Hence it is possible to define one or more declarative attribute classes for each type of requirements and use the attributes to describe the code.

For instance, an attribute class "*NetworkDependency*" with parameters *protocol*, *direction,* and *port* can be defined. When creating a program which depends on network services, a function *ReadData* that accesses the network (or the class containing the *ReadData* function) can be assigned the declarative attribute:

```
<NetworkDependency(Protocol.TCP, Direction.Out, 80)> _
Public Function ReadData()
```

Upon preparing the system for delivery, a tool for dependency extraction would scan the program code and find the meta-data attribute attached to the function 'ReadData()'. As a result, a record line would be created in the software profile document.

Most parts of the software profile document can be generated by using the following approach:

- data needed for the profile document header part are encoded as meta-attributes of the whole .Net assembly (or analogous concept in different platform, for instance, a component in terms of Java technology)
- dependencies on execution environment, i.e. requirements, are described as declarative meta-attributes of the code object which demands particular dependency, or at a higher hierarchical level if fine-grained requirements do not matter. Part of the requirements may be generated using MDA tools.
- all requirements listed in the program code are gathered by help of specific code analysis tools
- the requirements list is reviewed:
    - o dependencies that do not require more information are instantly transformed into requirements
    - o for dependencies referencing the development environment (for instance, a requirement "Short date format – the same as in development environment") specific tools are used to get the requested information from the environment (in case of the example – a function that queries the operating system for the short date format)
- the gathered list of requirements is merged with requirements from linked code libraries' profile documents, using the minimum supplement approach
- the obtained software profile document is reviewed by the developer to eliminate redundant requirements and to add the missing items.

## 5 First Practical Experience

Initial practical development has already been carried out according to the methodology provided in the paper. The development has resulted in a fully functional proof-of-concept version of the software profile verification toolset.

The demonstration software was developed using Microsoft .Net framework 2.0 platform. The chosen framework supports dynamic loading of modules from user code, which is a significant requirement for a good implementation of SWPF.

The code was separated into code assemblies according to the architecture of SWPF: the "business application program" which can be launched at maintenance mode (1), core module of SWPF (2), and an assembly containing implementations of some typical verification modules (3).

When the program (1) is loaded, it enters the "maintenance" mode and performs environment testing. To do the testing, it loads the SWPF core assembly (2) which further handles the tests. The SWPF core looks for an adequate software profile document. It considers that the software profile file is located in the same directory as the (1) executable files and named according to format *<executablefile>.swp*.

The *SWP* file is an XML file containing assembly identification information (for ensuring that the application currently under test is the correct one) and a listing of requirements each described using an XML element. The profile document used in the experiment was created manually.

When the SWP document is loaded, it is parsed into individual requirements. In the conceptual model we have introduced support only for two very simple, but potentially useful used requirement types:

- Regional settings – short date format requirement. In Latvian grammar, the format "dd.mm.yyyy" is advised. However, in Microsoft Windows the default locale settings are different and short date format is provided as "yyyy.mm.dd". The format string is frequently being changed during installation of the OS to the grammatically correct one; hence the actual setting in deployment environment may vary. In an isolated enterprise environment, the setting is typically the same on all computers and therefore the systems developed in-house are more likely not to be aware of the possible differences.
- File system – checking if specific path exists. In our experiment, the path existence evidence was used to check if "Windows XP service pack 2" is present in the system. The service pack was known to install itself at a specific location in the file system.

To find the appropriate tool for testing a particular requirement, the SWPF uses an "inventory list" – another XML file describing locations of EVMs. The conceptual model does not introduce distinction between requirements and requirement evidences; it assumes a 1:1 relation between requirements and EVMs (an EVM may handle one type of requirements). Therefore, items in the inventory list have a property describing the name of the requirement XML node that the current EVM can handle. When an appropriate EVM is found for evaluating a requirement, the assembly containing EVM's code is loaded and the requirement information is handed to the EVM object which further evaluates the requirement.

A test run of the application was performed on several computers in different environments and it was found out that even such a trivial environmental test may show inadequacies on some computers. Some of the computers tested did not have the required short date format, all had Windows XP SP 2 installed.

## 6 Conclusion

The paradigm of the software execution profile document is a step towards the development of smart technology compatible software. The methodology can be adapted in a fully manual manner – by composing the requirements profile document based on the know-how obtained during development and later using the document to check if the installation environment conforms to the requirements (this process can be reduced to textual installation descriptions as used by classical methods). However, the full power of the methodology provided can be gained when specific tools are used to generate the profile document and to validate the execution environment.

The use of the software profile concept described in the paper – verification of execution environment during installation and at run-time – is not the only one possible. Some of the other applications are:

– documentation used for systems maintenance
– discovery of cross-system bonds without inspecting the environment
– environment clean-up upon disposal of an outdated system (or previous version of the system)
– by creating a centralized registry of software execution profiles, it can automatically be perceived as a registry of available resources which allows answers to maintenance questions such as "Is this database still in use?", "Why do we have to have port 80 open on the external firewall?"
– the ideology of the software execution profile can be applied not only to executable code programs and libraries, but also, for instance, SQL "applications"

The solution for testing execution environment provided in the paper can be implemented incrementally – by expanding the set of resource types that can be verified.

The first practical experience in applying the described methodology has already been achieved and indicates that the approach is suitable for practical use.

## Acknowledgements

The research is supported by the European Social Fund (ESF).

## References

1. Bičevska Z., Bičevskis J.: Smart Technologies in Software Life Cycle. In: Münch J., Abrahamsson P. (eds.): Product-Focused Software Process Improvement. Lecture Notes in Computer Science, Vol. 4589. Springer-Verlag, Berlin Heidelberg (2007)
2. Roman, G.-C., Julien, C., Payton, J.: A Formal Treatment of Context-Awareness. In: Wermelinger, M., Margaria-Steffen, T. (eds.): Proceedings of the 7th International Conference on Fundamental Approaches to Software Engineering. Lecture Notes in Computer Science, Vol. 2984. Springer-Verlag, Berlin (2004)

3. Havelund, K., Rosu, G.: An Overview of the Runtime Verification Tool Java PathExplorer. Formal Methods in System Design Vol. 24(2), pp 189-215 (2004)
4. Tosi, D.: Research Perspectives in Self-Healing Systems. Report of the University of Milano-Bieocca (2004)
5. Nami, M., R., Bertels, K.: A Survey of Autonomic Computing Systems. In: ICAS '07: Proceedings of the Third International Conference on Autonomic and Autonomous Systems. IEEE Computer Society. Washington, DC, USA. (2007)
6. Wang, Y., King, G. Wickburg, H.: A Method for Built-in Tests in Component-based Software Maintenance. In: Proceedings of the Third European Conference on Software Maintenance and Reengineering, IEEE Computer Society, Washington, DC, USA. (1999)
7. Barbier, F., Belloir, N.: Component Behavior Prediction and Monitoring through Built-In Test. In: 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'03). IEEE Computer Society, Los Alamitos, CA, USA (2003)
8. Janzen, D. Saiedian H.: Test-Driven Development: Concepts, Taxonomy, and Future Direction. IEEE Computer. September 2005 (Vol. 38, No. 9) pp. 43-50 (2005)
9. Kim, T., Park, C., Wu, C.: Mock Object Models for Test Driven Development. In: SERA '06: Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications. IEEE Computer Society. Washington, DC, USA (2006)
10. Shamshiri, S., Esmaeilzadeh, H., Navabi, Z.: Instruction Level Test Methodology for CPU Core Software-Based Self-Testing. In: HLDVT '04: Proceedings of the High-Level Design Validation and Test Workshop, 2004. Ninth IEEE International. IEEE Computer Society. Washington, DC, USA (2004)
11. Troelsen, A.: Developer's Workshop to COM and ATL 3.0. Worldwide publishing, 2000
12. Hibernate Reference Documentation http://www.hibernate.org/hib_docs/reference/en /html/session-configuration.html
13. Beydeda, S.: Research in Testing COTS Components - Built-in Testing Approaches. In: ACS/IEEE 2005 International Conference on Computer Systems and Applications. Bonn, Germany (2005)
14. Kleppe, A., Warmer, J. Bast, W.: MDA Explained: The Model Driven Architecture--Practice and Promise. Addison Wesley, (2003)

LATVIJAS UNIVERSITĀTES RAKSTI. 2008, 733. sēj.:
DATORZINĀTNE UN INFORMĀCIJAS TEHNOLOĢIJAS 53.–71. lpp.

# Use of Design Patterns in PHP-Based Web Application Frameworks

Andris Paikens, Guntis Arnicans

Department of Computing University of Latvia
Raiņa blvd 19, Rīga, Latvia LV-1586

Andris.Paikens@di.lv, Guntis.Arnicans@lu.lv

**Abstract.** It is known that design patterns of object-oriented programming are used in the design of Web applications, but there is no sufficient information which data patterns are used, how often they are used, and the level of quality at which they are used. This paper describes the results concerning the use of design patterns in projects which develop PHP-based Web application frameworks. Documentation and source code were analysed for 10 frameworks, finding that design patterns are used in the development of Web applications, but not too much and without much consistency. The results and conclusions can be of use when planning and developing new projects because the existing experience can be taken into account. The paper also offers information which design patterns are not used because they may be artificial or hard-to-use in real projects. Alternatively, developers may simply lack information on the existence of the design patterns.

## 1 Introduction

The rapid increase in the number of Web users over the last two decades, the expanded opportunities and accessibility of software design, and the greater demand for such applications – all of this has contributed to an enormous increase in the number of people who are working on the design of Web applications. Web applications used to be nothing more than an add-on to some other serious system for a period of time not so long ago; and design of these web applications involved people with a great deal of experience in other areas of software. But today eager young people begin designing Web pages without being aware of even the simplest principles of software design. The complexity of applications has increased, and their use has become more serious.

The authors of this paper are interested in the use of design patterns in frameworks related to Web application design, because the true use of design patterns in actual projects has not been yet described to any great extent, which means that it is not yet clear whether this approach is of use for the relevant assignments.

## 2 Problems in the Use of Design Patterns

### 2.1 Design patterns

One can agree with Jason E. Sweath [1], who has argued that many software design problems are resolved again and again, and many fundamental solutions are established with the goal of speeding up the development processes, reducing the amount of work that has to be done, and improving the quality of software and, by extension, that of the resulting products. Many of these solutions can be brought together under the heading of "design patterns".

How to describe the concept of design patterns in one sentence? Design patterns represent theoretical material collected by experts in the field regarding the principles which should be used in solving a specific problem, the issues that should be taken into account, and the issues that are unimportant in the context of the relevant problem.

Use of design patterns may seem quite natural when working with a language that supports the object-oriented software design and programming approach (OOP), but it is a fact that languages which support OOP are not mandatory. However, even if a language that does not support OOP principles is used, it is necessary to be familiar with the foundations of OOP to understand the principles of design patterns and their use.

The literature about design patterns offers several definitions [2], [3], [1]. If we correlate these, we come up with a new definition of design patterns:

Design patterns describe a specific design problem which is repeatitive and which appears in the context of a specific design, offering a proven and general scheme to solve the problem. The solution scheme is detailed through a description of its components, their responsibilities and relationships, and the way in which they work together.

Here we present the three-part scheme which is at the foundation of every design pattern [9]:

1. *Context* – the situation in which the problem occurs;
2. *Problem* – the problem which repeatedly occurs in this context;
3. *Solution* – a proven resolution to the problem.

The context of a design pattern refers to the environment in which a problem arises. Context also covers all steps that have been taken before the specific problem occurred. Context can also include the steps that will have to be taken after the problem occurs.

Problem in this regard is an issue which must be handled by implementing specification requirements and by taking the context into account. When the problem is identified, its overall specifications must be designed, and its essence must be understood – what is the specific design problem that has to be solved?

Problem can be divided into several sections:

1. Requirements for the solution, e.g., it must be effective;
2. Issues that must be taken into account, e.g., which standards should be observed;
3. The desired properties.

Solution in a design pattern shows how to resolve the repeated problem and how to balance out related issues. There are two parts to the solution – the structural part, which is the static part of the solution, and the performance part, which is the dynamic part of the solution.

### 2.2 Properties of Design Patterns

Each design pattern is different from others, because each is intended for the solution of a different problem, or for a different solution to one and the same problem. At the same time, all design patterns have certain properties in common.

In [9], [1], [10], and other sources, the following properties of design patterns are mentioned; to a greater or lesser extent, all design patterns have them in common:

- A design pattern focuses on a design problem which is repeatitive and which occurs in specific design situations, offering a solution to the problem;
- Design patterns document existing, proven, and applied design solutions;
- Design patterns identify and make more precise abstractions, which is more than just defining an individual class, instance, or component;
- Design patterns ensure common vocabulary and understanding regarding the relevant design principles;
- Design patterns are a way of documenting software architecture;
- Design patterns support the development of software with certain desirable properties;
- Design patterns help to design complicated and heterogeneous software systems;
- Design patterns help to put reins on the complexity of software.

### 2.3 The Specifics of Web Applications

Web applications are quite different from other types of software. The execution process is one of the main differences – most of the processing is carried out on the server, the result is sent to the user's browser, and then the user can engage in other activities related to the application. For that reason, it is harder to work with objects – the PHP software design language, for instance, establishes an object which exists only for a short period of time – while the code is responsible for handling the particular request execution. Not all software design languages that are used in the design of Web applications are supported by OOP, and that makes it more difficult to deal with more complicated problems and to use design patterns.

The most important difference between Web applications and others is that most people have to use a Web browser to employ the software, and that covers some of the relevant functionality. Web applications are also affected by the fact that they are stored on servers, are called up, and are carried out both on the server and on the workstation of the user.

## 2.4 Positioning the Problem

While researching the use of design patterns in Web applications, we sought to learn the current situation in terms of whether design patterns are used in Web applications at this time at all and, if so, to what extent.

Theories about software design are good if they can be sooner or later applied in practice. Much time has passed since the first publications about design patterns, and design patterns also are of use in the design of Web applications. Hence, review of several projects can lead to the discovery of overall trends in the use of the design patterns.

There are many different design patterns, but many popular OOP design patterns must be viewed differently when it comes to Web applications. From all available design patterns, these were reviewed: Abstract Factory, Application Controller, Active Record, Adapter, Builder, Command, Composite, Custom Tag, Data Mapper, Decorator, Dependency Injection, Domain Model, Factory Method, Front Controller, Handle-Body, Iterator, MockObject, Model-View-Controller, Mono State, Observer, Proxy, Registry, Server Stub, Singleton, Specification, State, Strategy, Table Data Gateway, Template View, Template Method, Transform View, ValueObject, View Helper, Visitor. For detailed description of these design patterns see Appendix A.

Some of these have emerged in a natural way. Others are artificial and seldom used in real life. The review of the situation told us which design patterns are more popular and how many projects use them. The results could be used as hypotheses regarding the purposes toward which design patterns are used – are they used to reuse fragments of source code? Are they used to divide up the project into logical components to reduce the complexity of understanding and development? Or are they merely a fashion statement aimed at advertising one's own product?

## 2.5 Benefits from Researching the Existing Situation

The benefits of learning about the existing situation in the use of design patterns for the development of Web applications might be the following:

- Information about the existing situation helps others to start using design patterns, achieving the level which specialists in this area achieved after years of practice in testing the proposed design patterns more quickly. A beginner can immediately use the most popular tested design patterns, which means that he or she has more time to look for other design patterns aimed at specific tasks.
- Additional information is obtained about the diversity and use of design patterns. Research cannot immediately lead to precise answers to all the possible questions, but it outlines the directions and areas in which answers could be obtained through additional research. Our research showed that not all design patterns are actually needed from the list of so many. There are a few seemingly attractive design patterns that are not used at all.
- One can draw conclusions whether design patterns fulfil their mission in the first place. Do they help people to understand software architecture and logic? To what extent do design patterns support the principle of "divide and conquer", i.e., to what extent are the logic of systems described via design patterns?

- One gets an idea about the attitude of designers vis-à-vis design patterns. Is the use of design patterns in source code displayed openly, or is it hidden and thus hard to recognise?
- There is a better understanding of problems which are not solved via the use of design patterns. Hence, existing design patterns can be reworked, and new ones can be developed. It may well be that many design patterns which are used in classical object-oriented projects are defined as inappropriate for the development of Web applications.

## 2.6 The Method for Studying Design Patterns

To gain an objective idea about the use of design patterns, the research was based on the following principles:

1. Several solutions in one and the same class of tasks must be studied, i.e., the relevant Web applications must be used for more or less the same task;
2. Web applications basically use one and the same software design language;
3. There is access to documentation about the application and the source code (open code software is reviewed);
4. Developers have at least minimal knowledge about design patterns, learned from available books about design patterns.

Framework development projects which are written in the PHP language corresponded to these principles quite well. All of the selected projects were based on open source software, so project documentation and source code were available.

Web application frameworks are ideologically close to design patterns because they handle many of the same functions, albeit at a different level. This means that the developers of web application frameworks are far better informed and knowledgeable about design patterns than is the average Web application designer; and this is very believable. The first review of project documentation confirmed it – design patterns were discussed to a sufficient degree.

It was decided that the research would be based on the price/performance ratio, thus obtaining information about the existing situation via sensible use of resources. Without preliminary knowledge, it is hard to choose the frameworks which are needed for the research if more than 40 frameworks are available. The research was essentially based on reviewing 10 PHP frameworks that were mentioned in the article [5]. On the basis of various criteria, these are among the best and most popular frameworks. In terms of projects, researchers studied project documentation, project Web page, and the source code.

Many design patterns which are frequently mentioned in the literature were chosen. The researchers checked whether each selected design pattern is mentioned in some way in documentation, the homepage, and the source code. The source code was examined to look for the names of the methods as well as comments.

The information allowed researchers to determine whether the relevant design pattern was being used for project implementation. Unclear situations were also noted. The researchers did not analyse whether the design patterns were used in strict accordance with the principles defined in the literature.

Initially, the goal was to interview the organiser of each project, but that would have required greater resources and would not have had all that much effect on the results. We wanted to learn about the most important trends in the use of design patterns, and small omissions in the process could not have had a major effect on the results.

## 3 The Research Object

### 3.1 Design Patterns Used in the Research

Information on design patterns can be found in many books, for example [6] and [7], and we used many resources in our work. We would particularly like to point out three resources and the information about design patterns contained therein.

First, the book [1], which contains many design patterns and also explains the implementation of these patterns in PHP (*Value Object, Factory, Singleton, Registry, Mock Object, Strategy, Iterator, Observer, Specification, Proxy, Decorator, Adapter, Active Record, Table Data Gateway, Data Mapper, Model-View-Controller*).

A clear and understandable description of design patterns is found in www.dofactory.com – *Factory Method, Adapter, Builder, Bridge, Proxy, State, Strategy, Template Method, Visitor.*

Very useful information is also found in Wikipedia (http://en.wikipedia.org), although that resource does not provide information on all specific design patterns (*Dependency Injection, Abstract Factory, Mock Object, Singleton, Composite, Decorator, Chain-of-Responsibility, Observer, Iiterator, Active Record, Model-View-Controller*).

For detailed description of these design patterns see Appendix A.

### 3.2 Frameworks Used in the Research

"*A framework is a basic conceptual structure used to solve a complex issue. This very broad definition has allowed the term to be used as a buzzword, especially in a software context.*

*A software framework is a re-usable design for a software system (or subsystem). A software framework may include support programs, code libraries, a scripting language, or other software to help develop and glue together the different components of a software project. Various parts of the framework may be exposed through an API.*" [8]

The frameworks used in the research are defined in [5], which has all-encompassing information on each of them – see Table 1. Some of the information is unquestionably out of date, but we must remember that many frameworks are constantly being developed and improved, and that makes it difficult to complete the list.

Table 1 is presented not to define the best framework, but instead to give the reader a sense of what each framework does. The functions could influence the decision to use or not to use a specific design pattern. Detailed description of frameworks listed below can be found in Appendix B.

**Table 1:** Comparison of 10 different frameworks

| | PHP4 | PHP5 | MVC | Multiple DB's | ORM | DB Objects | Templates | Caching | Data Validation | Ajax | Authentification module | Modules |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Zend Framework | | + | + | + | + | + | + | + | + | | + | + |
| CakePHP | + | + | + | + | + | + | + | + | + | + | + | |
| Symfony Project | | + | + | + | + | + | | + | + | + | + | |
| Seagull Framework | + | + | + | + | + | + | + | + | + | | + | + |
| WACT | + | + | + | + | | + | + | | + | | | |
| Prado | | + | | + | | | + | + | + | + | + | + |
| PHP on TRAX | | + | + | + | + | + | | | + | + | | |
| ZooP Framework | + | + | + | + | | + | + | | + | + | + | + |
| eZ Components | | + | | + | | + | | + | + | | | + |
| CodeIgniter | + | + | + | | | + | + | + | + | | | + |

### 3.3 The Results of the Study of Frameworks

Before we analyse the results in depth, we must stress once again that using or not using design patterns cannot be the indicator to determine the best or worst framework. The comparison of two different frameworks only makes it possible to draw conclusions about the methods used by their developers. It is not possible to decide that one is better than the other – unless, of course, the comparison is based on the use of a specific design pattern in both frameworks, also looking at other indicators such as the amount of time that is needed to update the project code and to make the relevant changes in designing the project. However, these indicators are not really appropriate for comparison because if a project represents more than the strict implementation of a specific design pattern, then it includes several design patterns or, perhaps, solutions which have nothing to do with design patterns. Business logic and databases are mutually related, and it is all but impossible to measure the amount of time spent on the implementation of a single design pattern.

In other words, we evaluated the use of design patterns, not the frameworks within which the design patterns were used.

The research allowed us to create several result tables which have been combined for the purposes of this article in Table 2. The following notations are used:

"d" – the design pattern is mentioned in project documentation or on the homepage;

"c" – the design pattern is mentioned in the comments on the software code;

"m" – the design pattern's name is part of the name of the software design method;

**Table 2**: Research results

| | Zend Framework | CakePHP | Symfony Project | Seagull Framework | WACT | Prado | PHP on TRAX | ZooP Framework | eZ Components | CodeIgniter | “d” | “c” and “m” | “c” and “m” and “d” |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Abstract Factory | c | | | | | | | | | | | 1 | 1 |
| Application Controller | c | c | | c | d | | cd | c | | mc | 2 | 6 | 7 |
| Active Record | | d | | | | cmd | mcd | | | dc | 4 | 3 | 4 |
| Adapter | mdc | | cm | c | | m | | | | c | 1 | 5 | 5 |
| Builder | mc | | mc | c | m | d | | | cm | | 1 | 5 | 6 |
| Command | ? | ? | mc | ? | | ? | ? | ? | ? | ? | | 9 | 9 |
| Composite | | | mc | cd | ? | m | | | | | 1 | 4 | 4 |
| Custom Tag | | | | | | | | | | | | | |
| Data Mapper | | d | | | | cm | | | | | 1 | 1 | 2 |
| Decorator | mc | | cmd | cd | m | cm | | | | | 2 | 5 | 5 |
| Dependency Injection | | | | | | | | | | | | | |
| Domain Model | | | | | d | | | | | | 1 | | 1 |
| Factory Method | c | | c | dc | c | c | | | | | 1 | 5 | 5 |
| Front Controller | cmd | d | d | c | cd | | | | | c | 4 | 4 | 6 |
| Handle-Body | | | | | | | | | | | | | |
| Iterator | cm | | cm | d | c | m | | | c | | 1 | 5 | 6 |
| MockObject | | | | c | c | | | | | | | 2 | 2 |
| Model-View-Controller | d | dc | d | cd | d | c | d | d | | cd | 8 | 4 | 9 |
| Mono State | | | | | | | | | | | | | |
| Observer | cm | m | cm | cd | m | | m | | | | 1 | 6 | 6 |
| Proxy | cmd | | ? | cmd | | | | | | | 2 | 3 | 3 |
| Registry | m | mc | m | cd | c | | | | | c | 1 | 6 | 6 |
| Server Stub | | | | | | c | | | | | | 1 | 1 |
| Singleton | c | c | cm | ?d | | c | c | c | c | c | 1 | 9 | 9 |
| Specification | ? | | ? | | ? | ? | | | | | | 4 | 4 |
| State | ? | | ? | ? | ? | ? | | ? | | | | 6 | 6 |
| Strategy | cm | | | ?d | ? | | | | | | 1 | 3 | 3 |
| Table Data Gateway | | | | | | | | | | | | | |
| Template View | | | | d | d | | | | | | 2 | | 2 |
| Template Method | | | | | c | | | | | | | 1 | 1 |
| Transform View | | | | | | | | | | | | | |
| ValueObject | mc | | | mc | | | | | | | | 2 | 2 |
| View Helper | d | c | | | | | | | | | 1 | 1 | 2 |
| Visitor | | | | c | | | | | c | | | 2 | 2 |
| “d” | 5 | 4 | 3 | 11 | 5 | 2 | 3 | 1 | | 2 | | | |
| “c” and “m” | 17 | 7 | 13 | 18 | 13 | 13 | 5 | 2 | 5 | 8 | | | |
| “c” and “m” and “d” | 19 | 10 | 15 | 19 | 16 | 14 | 6 | 5 | 5 | 7 | | | |

"?" – the design pattern's name is mentioned in the name of the class or method, but without detailed analysis, so it cannot be understood whether the method or class implements the design pattern, or whether the name of the design pattern simply coincides with the name of the method or class.

## 3.4 Mention of Design Patterns in Documentation

Before examining the overall situation, let us look at those design patterns which are mentioned in documentation. Documentation for us meant the homepage, the user instructions, or any other documentation offered by the project developer (except for software comments).

The frequency of use is seen in column "d", but it has to be remembered that this is perhaps not a depiction of the actual situation. Still, it does point to overall trends. Some of the design patterns might be a marketing trick with which the project developers try to attract users. Some design patterns may be mentioned but not really used, being planned only for the next project versions. Many of the indicated design patterns are used, but it is hard to tell the extent to which they are. An additional survey of developers would not change the results substantially because we assume that there are no cardinal differences between the documentation and reality.

It is also impossible to determine whether the noted design patterns have been the only ones used in each specific project because it is possible that a specific design pattern was used but is not mentioned in the documentation. Some projects had fairly incomplete information sections in terms of homepages and documentation. However, the general trends can be determined with certainty.

Table 2 shows that the *MVC (Model-View-Controller)* design pattern has been used most often in frameworks – in 8 projects in total. Other frequently used include *Front Controller* and *Active Record* (4 times). Still other design patterns are used in just one or two projects, and there are some design patterns which are not used at all.

Table 2 shows that the framework which uses most design patterns is *Seagull* – 11 different design patterns are used in it.

## 3.5 Mention of Design Patterns in Source Code

When it comes to the design patterns mentioned in documentation, it can be asserted that they have been used in the implementation of frameworks, or at least the developers have wanted to use them but have not done so for various reasons. The use of design patterns described in comments and methods is perhaps more questionable because of the unambiguous way in which methods and classes are named and comments about them are developed. Only in the guidelines to the *Zend Framework* there is a mention of the fact that the name of a design pattern must be included in the name of a class or method if it is used therein.

The names of some of the design patterns may coincide with the names that have been chosen by the developers for methods designed for very different purposes. It is impossible to make the design patterns table more precise because that would require not only a complete study of all the design patterns, which is very difficult, but also an in-depth insight into all of the frameworks. That is almost impossible considering how many different frameworks there are. The task would be unrealistic for a few people who have a few months to spend on the task. Certainly the amount of resources that would be needed is not justified in terms of the results that could be obtained.

Table 2 suggests that the scene is very different if we compare the design pattern usage in code (column "c and m") to the mention of design patterns in documentation (the column "d"). There is no distinct leader here – the *Singleton* design pattern is used most often (9 times). The *Command* design pattern is used with equal frequency, but this fact is quite questionable because the design pattern bears a name which is also used for many software design methods; hence it may well be that in some cases the *Command* design pattern was not used at all. The design patterns *Application Controller* and *Observer* are used six times apiece. Others are used quite often but with less frequency. Still other design patterns are not used at all.

Among the frameworks in terms of the use of design patterns, *Seagull Framework* is the leader with 18 design patterns.

## 3.6 The Timeframe of the Research and Evolution of Frameworks

Initial research took place in late 2005 and at the beginning of 2006. Up to date results were collected until October 2007. Several things have changed since the authors of this paper investigated the subject for the first time, but these changes are not shocking as evolving products are upgraded time after time. The authors upgraded the original research by refreshing information on the usage of design patterns and related conclusions; some information was added in separate chapters of this paper so that readers could get an insight of how various frameworks have been developed and improved. As it can be observed, most of the products have not changed much over this period of time.

It is interesting how different frameworks have evolved over time. In Table 3, information is presented about the versions that were available in the spring of 2006 and the versions that are available in the autumn of 2007.

Improvements over time are not an unusual phenomenon if we speak about software. Changes show us the activities and intensity of development. The maturity level of the software can also be observed from such record of change as products do not change too much if a stable version is reached and no new functionality is demanded. Table 4 contains differences between the use of design patterns in spring 2006 and autumn 2007. Use of patterns mentioned here is implemented in recent versions of these frameworks and was absent in early 2006.

As we can see from Table 4, there are no many changes.

**Table 3**: Framework version evolution

| Framework | Spring 2006 | Autumn 2007 |
|---|---|---|
| **Zend Framework** | **Preview 0.1.3** | **1.0.2** |
| **CakePHP** | **1.0.1.2708** | **1.2.0** |
| **Symfony Project** | **Beta 0.6.2** | **1.0.7** |
| **Seagull Framework** | **0.6.0RC2** | **0.6.2** |
| **WACT - Web Application Toolkit** | **0.2 alpha** | **0.2alpha** |
| **Prado** | **3.0.0 RC2** | **3.1.0** |
| **PHP on Trax** | **0.13.0** | **0.14.0** |
| **ZooP Framework** | **1.1** | **1.3** |
| **eZ Components** | **1.0.1** | **1.3.1** |
| **CodeIgniter** | **1.3.2** | **1.5.4** |

**Table 4**: Changes in the use of design patterns in the period 2006-2007

| | Zend Framework | CakePHP | Symfony Project | Prado | PHP on TRAX | ZooP Framework | eZ Components | CodeIgniter |
|---|---|---|---|---|---|---|---|---|
| Abstract Factory | c | | | | | | | |
| Application Controller | c | | | | | | | |
| Active Record | | | | cmd | | | | d |
| Adapter | c | | | | | | | |
| Builder | mc | | | d | | | cm | |
| Command | ? | ? | | ? | | ? | ? | ? |
| Data Mapper | | | | cm | | | | |
| Decorator | c | | | | | | | |
| Factory Method | c | | | | | | | |
| Front Controller | cm | | | | | | | |
| Iterator | c | | | | | | | |
| Model-View-Controller | | c | | | | | | |
| Observer | cm | | | | | | | |
| Proxy | cm | | | | | | | |
| Registry | | | | | | | | c |
| Server Stub | | | | c | | | | |
| Singleton | c | | | | c | c | | c |
| Specification | | | ? | | | | | |
| Strategy | c | | | | | | | |
| Visitor | | | | | | | c | |

### 3.7 Conclusions on the Use of Design Patterns

For a more complete understanding, the two aforementioned results were joined together to get an overall sense of the main trends in the use of design patterns in frameworks (column "c+m+d"). It cannot be said that the result is dramatically different, but the numbers are adjusted to a certain extent here.

The leading design patterns are *MVC* and *Singleton*, which are used in 9 of 10 frameworks. Next on the list is *Application Controller*, which is used 7 times. *Builder*, *Front Controller, Iterator, Observer, Registry*, and *State* are following with 6 times. *Adapter*, *Decorator*, and *Factory Method* are used 5 times each.

The following design patterns were not used in any projects among the studied: *Abstract Factory*, *Custom Tag*, *Dependency Injection*, *Handle Body*, *Mock Object*, *Mono State*, *Table Data Gateway* and *Transform View*. Apparently the problems which are resolved by these design patterns are not all that important for the designers of frameworks.

The overall view regarding frameworks is quite similar to the previous ones, but there are slightly higher number of design patterns that are used. *Seagull Framework* remains the leader, sharing first position with *Zend Framework*.

The most accurate idea about the use of design patterns in Web application frameworks is seen in the gray-shaded boxes of Table 2, as they indicate that the design pattern is mentioned both in the documentation and in source code.

### 3.8 Additional Observations

During our basic research, we noticed a few other important issues that have to do with the subject discussed here.

Our first observation is displayed in Table 2 – documentation and the actual software code are often not in line with one another. This does not necessarily mean that most frameworks are poorly documented, but the fact is that in most cases, there is no unambiguous link between the documentation and the source code. What is more, the design patterns that are used probably are cited in the documentation of the developers, but they are not noted in the code. This may occur because the use of a design pattern often involves development of several classes, each of which is well commented. However, the source code does not mention the design pattern which all of the classes have in common.

We already mentioned that documentation can be affected by a marketing trick – the desire to present wishes as facts. It is also possible that the opposite is true – developers do not want their competitors to learn about the most important elements of their project, although they can be studied through the analysis of the source code.

What might it mean?

1. If the basic goal is to ensure understanding of an open source project by dividing it up into logical segments in which design patterns are one of the instruments, then the goal has not been achieved. Documentation and the source code are incomplete and sometimes contradictory.
2. If the developers offer their own frameworks and ask others to trust them, then they should also popularise more standard approaches, i.e., the use of design patterns.

3. More precise use of design patterns would make it easier for new participants to become involved in projects.

Neither should we forget about the psychology of software designers. It is hard to place them in strict frameworks. Each software designer likes to reinvent the bicycle. The use of design patterns is something of a manufacturing line, and the creativity of the software designer can apply only to the details. Open source projects are based on enthusiasm and freedom of participation, and it is quite possible that the developers have greater knowledge about design patterns than is seen in their projects. However, they want to design software without any limitations on what they are doing.

It must also be remembered that large projects involving large or medium-sized groups of developers will inevitably move away from the intended route sooner or later. Over the course of time, people will forget agreements about standards related to coding and preparation of commentary, production of test examples, and procedures related to the submission of code. This creates a certain disorder as well as problems with documentation. Only strict control can prevent this; it is most possible if overall responsibility is taken by a small group of people such as Zend in the case of the "Zend Framework."

## 4 Conclusions

It may seem that the task of researching the use of design patterns in the design of Web applications is simple, but as soon as the work begins, it becomes clear that the situation is far more complicated than it has been expected. Some frameworks have good homepages, which most framework developers have not really ensured. Hence, finding information about projects can be very complicated or even impossible. Most developers have taken the time to prepare sensible comments in relation to their software codes, but sometimes it is quite hard to understand whether the method has a name because a specific design pattern has been used, or the developer has simply decided on what he or she considers to be the most appropriate name without even thinking about the design pattern.

After collecting all the information, we found that *MVC* and *Singleton* are still the leading design patterns. That is logical because the separation of data, business logic, and visualisation are the basic ideas in designing complicated systems.

Other design patterns which are frequently used include *Application Controller*, *Builder*, *Front Controller, Iterator, Observer, Registry*, and *State*. However, these are not the only design patterns used, which reminds us once again how very diverse the frameworks are – each uses a different set of design patterns which suggests that developers have encountered various problems during the development process.

Further research could focus on drafting the design for a framework which uses as many design patterns as possible, taking into account the experience in the use of design patterns in the surveyed projects. This could show the strength of design patterns in the way that most of the functionality could be described through known design patterns while new design patterns could be developed for the rest. Alternatively, researchers might find out that not everything can be described with the help of design patterns.

Still other alternative might be the attempt to create and study a completely functional open code project such as the *E-store*, which is based on a certain framework. The analysis of such a project might allow researchers to identify the real problems that can be resolved or hindered by the use of design patterns.

When we analysed the use of design patterns, we found it necessary to group all design patterns into logical groups because it is hard to manage a large number of design patterns which each is completely independent; therefore, it is also necessary to indicate the links and interaction among various design patterns.

In conclusion, we must also say that the success of a project does not depend on the use or omission of a design pattern. There are frameworks which use very few design patterns while others use many of them. It can never be claimed that this fact accounts for one framework being better or worse than other.

## References

1. Jason E. Sweat. PHP | Architect's Guide to PHP Design Patterns. Marco Tabini & Associates, Inc., 2005
2. Dirk Riehle, Heinz Züllighoven. Understanding and Using Patterns in Software Development. Theory and Practice of Object Systems 2, 1, 1996
3. Richard P. Gabriel. Patterns of Software: Tales from the Software Community. Oxford Univ. Pr., April 1, 1996
4. Christopher Alexander. The Timeless Way of Building. Oxford University Press, 1979
5. Dennis Pallett. Taking a look at ten different PHP frameworks. [online] www.phpit.net [referenced 13.10.2007]. Available online: http://www.phpit.net/article/ten-different-php-frameworks
6. Martin Fowler. Patterns of Enterprise Application Architecture. Addison-Wesley Professional, 2002
7. Design Patterns – Elements of Reusable Object-Oriented Software. Erich Gamma, Richard Helm, Ralph Johnson, John Vissides, Addison Wesley Professional, March 1995
8. http://en.wikipedia.org/wiki/Framework [referenced 30.11.2007]
9. Tony Marston. Design patterns – a personal perspective. [online] [referenced 14.05.2006]. Available online: http://www.tonymarston.net/
10. Brad Appleton. Patterns and Software: Essential Concepts and Terminology. [online] [referenced 15.05.2006]. Available online: http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html

## Appendix A: Design Patterns

The description of all design patterns mentioned in this table was taken from book [1].

| Design pattern | Description |
|---|---|
| Abstract Factory | *“Facilitates the building of families of related objects”* |
| Application Controller | *“A central point for handling navigation for an application, typically implemented in an index.php file dispatching based on URL query parameters.”* |
| Active Record | *“Creates an object that wraps a row from a database table or view, provides database access one row at a time, and encapsulates relevant business logic.”* |
| Adapter | *“Allows classes to support a familiar interface so you can use new classes without refactoring old code.”* |
| Builder | *“Facilitates the initialization of complex object state.”* |
| Command | *“Encapsulates a request as an object.”* |
| Composite | *“Manages a collection of objects where each "part" can stand in as a "whole". Typically organized in a tree hierarchy. ”* |
| Custom Tag | *“Improves presentation separation by encapsulating components to appear as new HTML tags.”* |
| Data Mapper | *„An object that acts as a translation layer between domain objects and the database table that contains related data.”* |
| Decorator | *“Attaches responsibilities to an object dynamically. Can simplify class hierarchies by replacing subclasses.”* |
| Dependency Injection | *“Constructs classes to accept collaborators through the constructor or setter methods, so that a framework can assemble your objects.”* |
| Domain Model | *“An object model of business logic that includes both data and behavior.”* |
| Factory Method | *“Facilitates the creation of objects.”* |

| | |
|---|---|
| Front Controller | *"A controller that handles all requests for a web application."* |
| Handle-Body | *"A collective name for design patterns that hold a reference to a subject object (for example, Proxy, Decorator, and Adapter)."* |
| Iterator | *"Easily manipulates collections of objects."* |
| MockObject | *"Supplies a stub that validates whether certain methods were or were not called during testing."* |
| Model-View-Controller | *"An application layering pattern that separates concerns between your domain model, presentation logic and application flow.*<br><br>*The Model-View-Controller (MVC) pattern organizes and separates your software into three distinct roles:*<br><br>*• the Model encapsulates your application data, application flow, and business logic;*<br><br>*• the View extracts data from the Model and formats it for presentation;*<br><br>*• the Controller directs application flow and receives input and translates it for the Model and View."* |
| Mono State | *"Allows all instances of an object to share the same state."* |
| Observer | *"Registers objects for later callback. Event-based notification. Publish/Subscribe."* |
| Proxy | *"Provides access to an object through a surrogate object to allow for delayed instantiation or protection of subject methods."* |
| Registry | *"Manages references to objects through a single, well-known object."* |
| Server Stub | *"Simulates a portion of your application for testing purposes."* |
| Singleton | *"Provides global access to a single instance of an object."* |

| | |
|---|---|
| Specification | *"Flexible evaluation of objects against dynamic criteria."* |
| State | *"Has an object change its behaviour depending on state changes."* |
| Strategy | *"Allows for switching between a selection of algorithms by creating objects with identical interfaces."* |
| Table Data Gateway | *"An object that acts as a gateway to a database table or view, providing provide access to multiple rows."* |
| Template View | *"Renders a page by replacing embedded markers with domain data."* |
| Template Method | *"Defines an algorithm with "hook" methods allowing subclasses to change the behavior without changing the structure."* |
| Transform View | *"Processes domain data sequentially to transform it to some form of output."* |
| ValueObject | *"Handles objects whose equality is determined by the value of the objects' attributes, not by the identity of the objects."* |
| View Helper | *"A class that helps the View by collecting data from the Model."* |
| Visitor | *"Defines an algorithm as an object that "visits" each member of a aggregate performing an operation."* |

## Appendix B: PHP Frameworks

Descriptions of the frameworks are taken from their own homepages.

**Zend Framework**

*Extending the art & spirit of PHP, Zend Framework is based on simplicity, object-oriented best practices, corporate friendly licensing, and a rigorously tested agile codebase. Zend Framework is focused on building more secure, reliable, and modern Web 2.0 applications & web services, and consuming widely available APIs from leading vendors like Google, Amazon, Yahoo!, Flickr, as well as API providers and cataloguers like StrikeIron and ProgrammableWeb.*

*Expanding on these core themes, we have implemented Zend Framework to embody extreme simplicity & productivity, the latest Web 2.0 features, simple* *corporate-friendly licensing, and an agile well-tested code base that your enterprise can depend upon.*

http://framework.zend.com [referenced 03.11.2007]

**CakePHP**

*Cake is a rapid development framework for PHP which uses commonly known design patterns like ActiveRecord, Association Data Mapping, Front Controller and MVC. Our primary goal is to provide a structured framework that enables PHP users at all levels to rapidly develop robust web applications, without any loss to flexibility.*

http://www.cakephp.org/ [referenced 03.11.2007]

**Symfony**

*Based on the best practices of web development, thoroughly tried on several active websites, symfony aims to speed up the creation and maintenance of web applications, and to replace the repetitive coding tasks by power, control and pleasure.*

http://www.symfony-project.org/ [referenced 03.11.2007 ]

**Seagull**

*Seagull is a mature OOP framework for building web, command line and GUI applications. Licensed under BSD, the project allows PHP developers to easily integrate and manage code resources, and build complex applications quickly.*

*Many popular PHP applications are already seamlessly integrated within the project, as are various templating engines, testing tools and managed library code. If you're a beginner, the framework provides a number of sample applications that can be customised and extended to suit your needs. If you're an intermediate or advanced developer, take advantage of Seagull's best practices, standards and modular codebase to build your applications in record time.*

http://seagullproject.org/ [referenced 03.11.2007]

**Wact**

*The Web Application Component Toolkit is a framework for creating web applications. WACT facilitates a modular approach where individual, independent or reusable components may be integrated into a larger web application. WACT assists in implementing the Model View Controller pattern and the related Domain Model, Template View, Front Controller and Application Controller patterns.*

*The WACT framework is developed with the philosophy of continuous refactoring and Unit Testing. WACT encourages these activities in applications based on the framework. WACT uses Simple Test as a unit testing framework.*

http://www.phpwact.org/ [referenced 03.11.2007]

**Prado**

*PRADO is a component-based and event-driven framework for rapid Web programming in PHP 5. PRADO reconceptualizes Web application development in terms of components, events and properties instead of procedures, URLs and query parameters.*

http://www.xisc.com/ [referenced 03.11.2007]

**PHP On Trax**

*Php On Trax (formerly Php On Rails) is a web-application and persistance framework that is based on Ruby on Rails and includes everything needed to create database-backed web-applications according to the Model-View-Control pattern of separation. This pattern splits the view (also called the presentation) into "dumb" templates that are primarily responsible for inserting pre-build data in between HTML tags. The model contains the "smart" domain objects (such as Account, Product, Person, Post) that holds all the business logic and knows how to persist themselves to a database. The controller handles the incoming requests (such as Save New Account, Update Product, Show Post) by manipulating the model and directing data to the view.*

*In Trax, the model is handled by what's called a object-relational mapping layer entitled Active Record. This layer allows you to present the data from database rows as objects and embellish these data objects with business logic methods.*

http://www.phpontrax.com/ [referenced 03.11.2007]

**ZOOP Framework**

*Zoop is an object oriented PHP framework. Zoop is modeled after the MVC design pattern. It is a high performance, secure, and scalable framework for PHP. It is designed to be very fast and efficient and very nice for the programmer to work with. Zoop has been built in a modular way so it is both easily extensible, and light. It has been in development and production use since 2001 and is quite mature.*

http://zoopframework.com/ [referenced 03.11.2007]

**eZ Components**

*eZ Components is an enterprise ready general purpose PHP components library used independently or together for PHP application development.*

http://ez.no/ezcomponents [referenced 03.11.2007]

**CodeIgniter**

*CodeIgniter is a powerful PHP framework with a very small footprint, built for PHP coders who need a simple and elegant toolkit to create full-featured web applications. If you're a developer who lives in the real world of shared hosting accounts and clients with deadlines, and if you're tired of ponderously large and thoroughly undocumented frameworks.*

http://www.codeigniter.com/ [referenced 03.11.2007]

# MDA AND MODEL TRANSFORMATIONS

# The Base Transformation Language L0+ and Its Implementation

Sergejs Rikacovs

University of Latvia, IMCS, 29 Raiņa blvd, Rīga, Latvia
sergejs.rikacovs@lumii.lv

**Abstract.** An efficient implementation of high level model transformation languages is well known as a complex problem. It is believed that the most appropriate way to implement transformation languages is bootstrapping. However, bootstrapping is not possible without an efficient base language. In this paper, a new low level model transformation language L0+ is proposed, for which there exists an efficient implementation. This language can be used as a base language in the bootstrapping process. L0+ does not have advanced pattern definition facilities, but the expressive power of this language is comparable to some more advanced languages. In spite of the fact that L0+ is quite a low level language, it can also be used for the development of model transformations directly. The presented paper is an extended version of the second chapter of [1].

**Keywords**: model transformation language, compiler, bootstrapping.

## 1 Introduction

During the last few years a new approach to complex system building was developed – MDA. Model transformations are considered to be one of the pillars of this approach [2].

Model transformation languages are a comparatively new kind of languages. The first standardization effort in this area was OMG MOF 2.0 Query/Views/Transformations (QVT) request for Proposal (RFP) [3]. In response to this request, QVT specification was developed [4]. QVT defines the standard way of defining transformations. According to this specification, source and target models should correspond to the MOF metamodel. QVT consists of three sublanguages:

- Relations – a high level declarative language;
- Core – also a declarative language, but it is more verbose and does not provide an implicit creation of trace instances, the semantics of the Relations language is defined in terms of this language;
- Operational mappings – a language allowing either to define transformations using a fully imperative approach (operational transformations), or to complement relational transformations with imperative operations implementing the relations (hybrid approach).

There are several independent model transformation languages apart from QVT. These languages can be divided into two groups: graphical and textual languages. It is worth noting that QVT belongs to both groups, because of its dual forms: the graphical and the textual one. While specifying transformations in a graphical form, transformation developer has the opportunity to represent mappings between patterns of source and target models in a direct and natural way. Graphical languages, such as GReAT[5] or MOLA[6], define transformations as a set of transformation rules. Every rule has a pattern part and an action part, in which the pattern specifies instance sets to be processed according to transformations described in the rule action part. The difference between various graphical transformation languages is in the expressiveness of patterns and control flow structures. Typical representatives of textual model transformation languages are [7, 8, 9]. Most of the textual model transformation languages are declarative languages also having some kind of pattern definition facilities. A recursion is usually used as the main control flow facility.

However, the research in the area of model transformations and effective implementation of model transformation languages is still topical.

In this paper, a new low level model transformation language called L0+ is proposed. This language has two important features:

- it is very simple, so being easy learnable by transformation developers;
- it has a very efficient implementation (principles of this implementation are also described in this paper).

The **main** use case for this language is the implementation of higher level languages (through the bootstrapping approach). The so called Lx language family is implemented this way [10]. Despite the importance of the use case mentioned above, several other use cases also exist. As an example (quite a significant one), the Transformation Based Graphical Tool Building Platform [11] can be mentioned, where L0+ is used as the main language for transformation development.

A more important feature of this language is that L0+ works not only with the model as a significant part of model transformations do, but also with the metamodel level (a notable example of a transformation language containing metamodel processing facilities is Viatra [12]). More precisely, L0+ = L0 + MM, where L0 is oriented towards model processing, and MM is oriented towards metamodel processing.

## 2 The Base Transformation Language L0

### 2.1 Basic Ideas

The language L0 contains minimal but sufficient constructions for model processing:

- creation/deletion of objects;
- getting/setting the value of an attribute of an object;
- creation/deletion of links;

- iteration through instances;
- low level control flow instructions;
- labels;
- unconditional control flow transfer operator;
- conditional control flow transfer operators.

## 2.2 Precise Definition of L0

To give a precise definition of L0 syntax and semantics, we should precisely fix the allowed metamodeling constructs. OMG suggests using MOF 2.0 for such purposes [13]. However, more simple approaches are usually used in practice. We will follow this tradition and will use a subset of MOF 1.4[14] seen in Fig. 1. to define metamodeling constructs to be allowed in metamodel definitions.

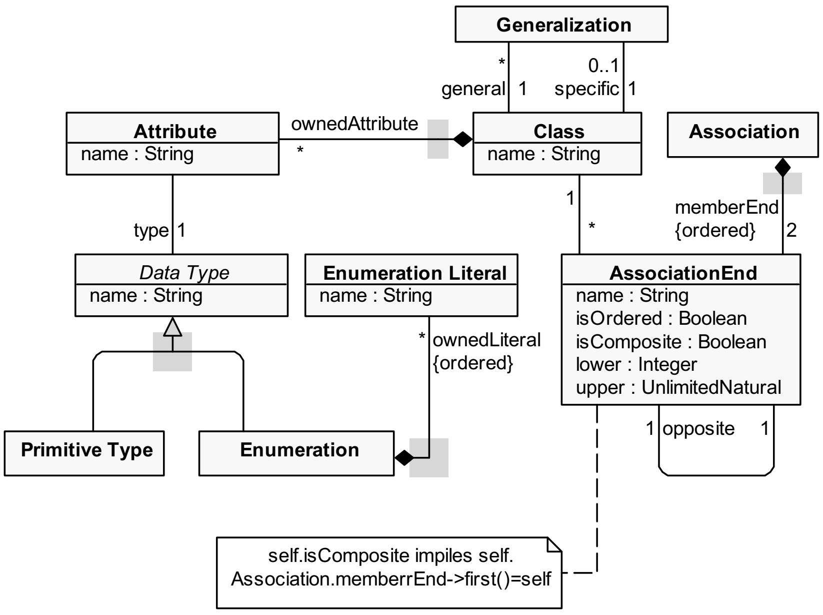


**Fig. 1.** Meta-metamodel

One can notice that packages are not present as an independent concept in this metamodel. In L0+, packages are simulated through qualified names.

L0 transformation program contains the following elements:

- transformation header, i.e. **transformation** <transformationName>;
- global variable definition part;
- the "useMM" directive – a path to a metamodel definition file is given through this directive. This file can contain following commands:
  - o **class** <className>**;**
    Defines a class with a given name.

- **attr** <className>.<attrName>**:**<ElementaryTypeName>**;**
  Defines an attribute with a given name and type.
- **assoc** <className>.**{**ordered**}**<card><roleName>**/**
  <roleName><card>**{**ordered**}.**<className>**;**
  Defines an association with corresponding properties.
- **compos** <compositeClassName>**.{**ordered**}**<card><roleName>**/**
  <roleName><card>**{**ordered**}.**<partClassName>**;**
  Defines a composition with corresponding properties.
- **rel** <subClassName>**.subClassOf.**<superClassName>**;**
  Defines a generalization relationship between given classes.
- **enum** <enumName>**:{** <enumLiteral1> **,** < enumLiteral2>**};**
  Defines an enumeration with given elements.

- A “native” subprogram (function or procedure) declaration part (headers of C++ functions used in the transformation). Like in every programming language, there can be some tasks for which L0 is not quite suitable (for instance, string processing, text parsing, etc.). To deal with these situations in L0, there exists a possibility to call a C++ function. For example, there is a String data type in L0, but the language per se does not define some such useful operations as Length, CharAt, and Substring on this type. If needed, transformation developers can easily implement these functions in C++ and then access them from L0 by using the concept of “native” subprograms.
- An L0 subprogram definition part (it is expected that exactly one subprogram of this part is labeled with the reserved word **main** thus defining the entry point for the transformation). An L0 subprogram definition also consists of several parts:
  - the subprogram header;
  - local variable definitions;
  - the keyword **begin**;
  - the subprogram body definition;
  - the keyword **end**;
- a transformation footer, i.e. **endTransformation;**

An elementary unit of any L0 transformation program is a **command** (an imperative statement). Before giving a detailed description of individual commands, it should be noted that the name of the meta-model element (i.e. class name, role name, attribute name, enumeration name, and enumeration literal name) can be specified in two different ways:

- as a String literal; for example, **addObj** x : Person;
- as a String variable; for example, **addObj** x : (s); In this case, the name of a meta-model element will be equal to the value of the corresponding String variable at the command execution time.

L0 contains the following commands:

1. **transformation** <transformationName>; Starts the transformation definition.
2. **endTransformation**; Ends the transformation definition.
3. **pointer** <pointerName> : <className>; Defines a pointer to an object of the class <className>.

3.1. **pointer** <pointerName> : **Void** ; Void pointer can point to objects of an arbitrary class.

4. **var** <varName> : <ElementaryTypeName>; Defines a variable of elementary data type – Boolean, Integer, Real or String.
5. **procedure** <procName>**(**<formalPrmList>**)**; Formal parameter list consists of formal parameter definitions separated by ",". A parameter definition consists of its name, the parameter type (the type can be an elementary type, a class from the meta-model or the reserved word **Void**), and the passing method (parameters can be passed by reference or by value). If the parameter is passed by reference, its type name is preceded by the **&** character.
6. **function** <funcName>**(**<formalPrmList>**)**:<returnTypeName>; Return type name can be an elementary type name, class name or the reserved word **Void**.
7. **begin**; Starts subprogram definition.
8. **end**; Ends subprogram definition.
9. **return**; Returns execution control to the caller.
10. **return** <identifier>; Returns the value of <identifier> to the caller. The type of <identifier> must coincide with the return type of the function. <identifier> is an elementary variable name or a pointer name. Instead of <identifier>, the reserved word `null` can be used. In this case function return type must be class or Void.
11. **call** <subProgName>**(**<actualPrmList>**)**; Actual parameter list can be empty. It consists of binary expressions (<binExpr>) separated by ",". More about <binExpr> can be found in the item 23.
12. **first** <pointerName> **:** <className> **else** <labelName>; Positions <pointerName> to an arbitrary [the first object (ordering is implementation dependent)] object of <className>. Typically, this command is used in combination with **next** command to traverse all objects of the given class. If <className> has no objects, <pointerName> becomes equal to null, and execution control is transferred to <labelName>. <className> in this command must be the same as or a subclass of the class used in the pointer definition. If it is a subclass, the value set of the pointer is narrowed (for the following executions of `next`).

    12.1. **first** <pointerName> : **(**<stringVarClassName>**)** **else** <labelName>;

13. **first** <pointerName>$_1$**:** <className> **from** <pointerName>$_2$ **by** <roleName> **else** <labelName>; Positions <pointerName>$_1$ to an arbitrary [the first (object ordering is implementation dependent)] object, which is reachable from <pointerName>$_2$ by a link <roleName>. Typically, this command is used in combination with the **next** command to traverse all objects connected to the given object by a link with the specified type. If there are no such objects, <pointerName>$_1$ becomes equal to null, and execution control is transferred to <labelName>. It should be noted that this command specifies (narrows) the value set of the pointer, which is taken into account when performing the **next** execution and assignment. After the command is executed, the value set of the pointer is narrowed to those objects, which are reachable from <pointerName>$_2$ by links with the given type (specified by <roleName>).

    13.1. **first** <pointerName>$_1$ **: (**<stringVarClassName>**)** **from** <pointerName>$_2$ **by** **(**<stringVarRoleName>**)** **else** <labelName>;

14. **next** <pointerName> **else** <labelName>; Gets the next object satisfying conditions formulated during the execution of "first" command and not visited

(iterated) with this variable yet. If there is no such object, <pointerName> becomes null, and execution control is transferred to <labelName>.

15. **goto** <labelName>; Unconditionally transfers control to <labelName>. <labelName> should be located in the current subprogram definition.
16. **label** <labelName>; Defines a label with the given name.
17. **addObj** <pointerName>**:**<className>; Creates a new object of the class <className>.

    17.1. **addObj** <pointerName>**:(**<stringVarClassName>**)**;
18. **addLink** <pointerName>$_1$**.**<roleName>**.**<pointerName>$_2$; Creates a new link (of type specified by <roleName>) between objects pointed to by <pointerName>$_1$ and <pointerName>$_2$ , respectively.

    18.1. **addLink** <pointerName>**.(**<stringVarRoleName>**).**<pointerName>;
19. **deleteObj** <pointerName>; Deletes an object pointed to by <pointerName>.
20. **deleteLink** <pointerName>$_1$**.**<roleName>**.**<pointerName>$_2$; Deletes a link, whose type is specified by <roleName>, between objects pointed to by <pointerName>$_1$ and <pointerName>$_2$, respectively.

    20.1. **deleteLink** <pointerName>$_1$**.(**<stringVarRoleName>**).**<pointerName>$_2$;
21. **setPointer** <pointerName>$_1$**=**<pointerName>$_2$; Sets <pointerName>$_1$ to the object pointed to by <pointerName>$_2$. If the value set of <pointerName>$_1$ does not contain the object pointed to by <pointerName>$_2$, then <pointerName>$_1$ is set to **null**. In place of <pointerName>$_2$ **null** can be used. In this case <pointerName>$_2$ will not point to any object (it will point to **null**).
22. **setPointerF** <pointerName>**=**<funcName>**(**<actualPrmList>**)**; Sets <pointerName>$_1$ to the object returned by <funcName>.
23. **setVar** <varName> **=** <binExpr>; <binExpr> is a binary expression consisting of the following elements: elementary variables, subprogram parameters, literals, attribute values (<pointerName>.<attrName>) and standard operators (+,-,*,/,&&,||,!) of elementary types. Besides the traditional way (i.e. <pointerName>.<attrName>) of getting/setting values of object attributes, there is a special way to do it: <pointerName>.(<stringVarAttrName>). The result type of this operation is **String**. For example, **setVar** <varName> **=** <pointerName>**.**(<stringVarAttrName>). Here, the value of the attribute is stored as a string in <varName>.
24. **setVarF** <identifier>**=**<funcName>**(**<actualPrmList>**)**; This command can be used to obtain the result value of the function of an elementary type. Identifier is a name of a variable. Variable type must coincide with the return type of the function.
25. **setAttr** <pointerName>**.**<attrName>**=**<binExpr>; Sets the value of the attribute <attrName> of the object pointed to by <pointerName> to the value of <binExpr>.

    **25.1. setAttr** <pointerName>**.(**<stringVarAttrName>**)** **=** <stringExpr>**;**
26. **type** <pointerName> **==** <className> **else** <labelName>; If the type of the object is identical to <className>, the control is transferred to the next command, else the control is transferred to <labelName>. Instead of equality symbol **==** inequality symbol **!=** can be used. Inheritance is not taken into account (i.e. this command works as oclIsTypeOf meaning the result of the comparison will be true, if and only if types are identical).

26.1. **type** <pointerName> **==** **(**<stringVarClassName>**)** **else** <labelName>;

27. **var** <varName>**==**<binExpr> **else** <labelName>; If condition is not true, the control is transferred to <labelName>. Instead of equality symbol, other (<, <=, >, >=, !=) relational operators compatible with argument types can be used.
28. **attr** <pointerName>**.**<attrName> **==** <binExpr> **else** <labelName>; If condition is not true, the control is transferred to <labelName>. Instead of equality symbol other (<, <=, >, >=, !=) relational operators compatible with argument types can be used.
29. **link** <pointerName>**.**<roleName>**.**<pointerName> **else** <labelName>; Checks whether there is a link (which type is specified by <roleName>) between objects pointed to by <pointerName>$_1$ and <pointerName>$_2$, respectively. If condition is not true, the control is transferred to <labelName>.

    29.1. **link** <pointerName>**.(**<stringVarRoleName>**).**<pointerName> **else** <labelName>;
30. **noLink** <pointerName>**.**<roleName>**.**<pointerName> **else** <labelName>; Checks whether there is no link (its type is specified by <roleName>) between objects pointed to by <pointerName>$_1$ and <pointerName>$_2$, respectively. If condition is not true, the control is transferred to <labelName>.

    30.1. **noLink** <pointerName>**.(**<stringVarRoleName>**).**<pointerName> **else** <labelName>;
31. **pointer** <pointerName>$_1$**==**<pointerName>$_2$ **else** <labelName>; Checks whether objects pointed to by <pointerName>$_1$ and <pointerName>$_2$, respectively, are identical. Instead of **==** inequality symbol **!=** can be used. If condition is not true, the control is transferred to <labelName>. Instead of <pointer2> `null` can be used.

It is easy to see that the language L0 contains only the very basic facilities for defining transformations. At the same time, it is obviously **complete** in the sense of its functional capabilities. Namely, this is why L0 is called the base transformation language.

### 2.2.1 Object-Oriented L0 constructs

L0 was designed as a low level language, and originally it was not supposed to be used for direct development of transformations. However, experiments proved that it is possible to use this language for direct development of transformations without significant loss of development speed.

For example, L0 is used for development of transformations in the context of Transformation Based Graphical Tool Building Platform. The total size of the source code of transformations developed in this project exceeds 20000 lines of L0. It is clear that it is becoming more and more difficult to ensure adequate modularization of code base of such a size with the only modularization facility being the concept of sub procedure.

Today the most popular code modularization method is OO. According to [15], OO has several fundamental elements:

- Class
- Object

- Method
- Message passing
- Inheritance
- Encapsulation
- Abstraction
- Polymorphism

It is easy to see that when we are working with model transformation language we have many of these constructs readily available through the metamodel definition or because of the fact that we are working with models. However, several important concepts are absent (most notably the concepts of method, message passing and polymorphism).

To let L0 users take advantages of OO approach, L0 is supplemented with the notion of method. It can be defined in the following ways:

- **procedure** <ClassName>**::**<methodName>**(**<formPrmList>**);**
- **function** <ClassName>**::**<methodName>**(**<formPrmList>**):**<ReturnType>**;**

To reference to the object, this method is called on from the body of the method, reserved word **this** can be used.

As it can be seen, the only difference between the method declaration and the ordinary function or procedure declaration is the fact that method declaration is linked to a certain class.

In a similar way constructs for method calling are introduced:

- **call** <pointerName>**.**<methodName>**(**<actPrmList>**);**
- **setVarF** <varName> **=** <pointerName>**.**<methodName>**(**<actPrmList>**);**
- **setPointerF** <pointerName> **=** <pointerName>**.**<methodName>**(**<actPrmList>**);**

Every method call is polymorphic – it depends on the actual type of the object this particular method is called on. An example of using these new constructs can be found in section 2.3.

It should be noted that there are transformation languages providing much more advanced constructs. For example, in QVT Operational Mappings it is allowed to define the so called mapping operation. That can be defined in the following way:

**mapping** <dirkind> <contexttype>::<mappingname> (<parameters>,) : <result-parameters>
**when** {<exprs>}
**where** { <exprs>}
{
**init** { … }
**population** { … }
**end** { … }
}

A mapping operation is syntactically described by a signature, a guard (a *when* clause), a mapping body and a post-condition (a *where* clause). The **init** section contains a code to be executed before the instantiation of the declared outputs. The **population** section contains a code for populating the result parameters, and the **end** section contains an additional code to be executed before exiting from the operation. In its simplest case (in case **when** and **where** clauses are not used), a QVT mapping operation is almost equivalent to an L0 method.

## 2.3 Example of an L0 Transformation

Let us consider oriented graphs. **Fig. 2.** presents one possible metamodel for oriented graphs.

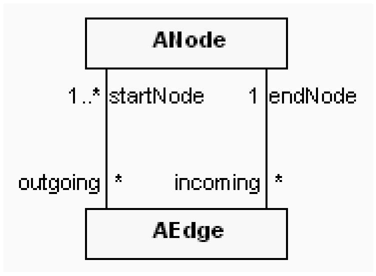


**Fig. 2.** Metamodel for oriented graphs

According to this metamodel, a graph in **Fig. 3.** corresponds to the instance found in **Fig. 4.**.

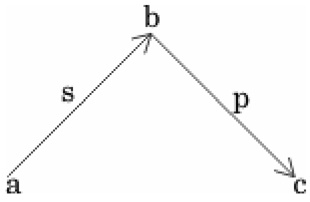


**Fig. 3.** An example of an oriented graph

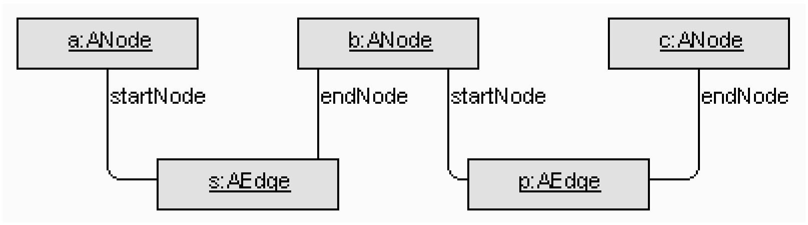


**Fig. 4.** Instances of the metamodel for oriented graphs

Several other metamodels (for example, the one found in **Fig. 5.**) are also possible for oriented graphs.

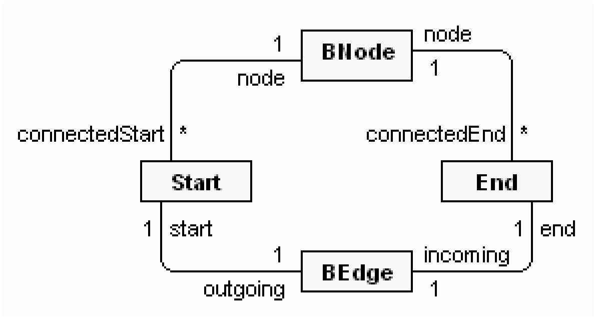


**Fig. 5.** Another metamodel for oriented graphs

According to this metamodel, a graph found in **Fig. 3.** will correspond to the instances found in **Fig. 6.**.

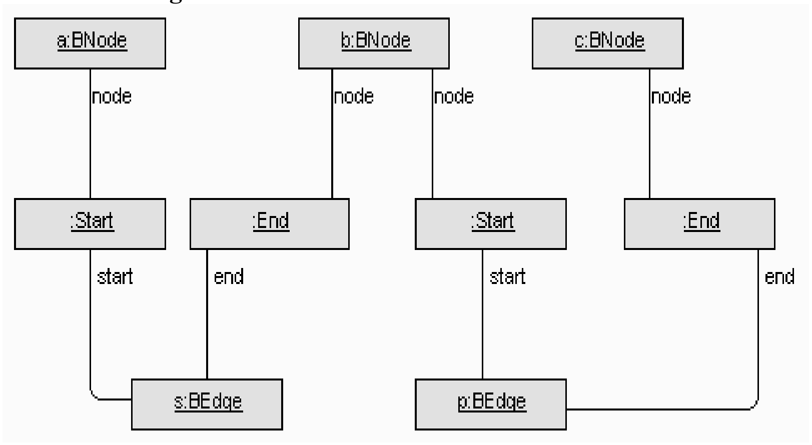


**Fig. 6.** Instances of another metamodel for oriented graphs

As it can be seen from the examples, we get different instances (models) for one and the same graph. At the same time it seems that these different models are quite close to each other. A natural problem arises – how to define a transformation taking a graph model corresponding to the metamodel A and producing a graph model corresponding to the metamodel B. The basic idea is to create one BNode for every ANode and to transform every AEdge to BEdge with corresponding Start and End.

To simplify this transformation we add a mapping association to the metamodel between classes ANode and BNode.

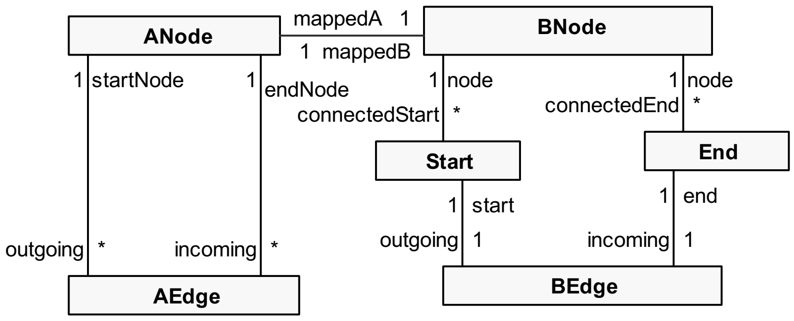


**Fig. 7.** Metamodel for oriented graphs with a mapping association

Transformation program in L0 implementing this algorithm can be found below.

```
transformation Graphs;
main procedure Graph2Graph();
  pointer a : ANode;
  pointer b : BNode;
  pointer aEd : AEdge;
  pointer bEd : BEdge;
  pointer edgeStart  : Start;
  pointer edgeEnd    : End;
  pointer aEdgeStNode : ANode;
  pointer aEdgeEnNode   : ANode;
  pointer mapBNode : BNode;
begin;
//copy nodes;
 first a : ANode else aNodeProcessed;
label loopANode;
   addObj  b : BNode;
   addLink a . mappedB . b;
   next a else aNodeProcessed;
   goto loopANode;
label aNodeProcessed;
//copy edges;
 first aEd : AEdge else aEdgesProcessed;
label loopAEdge;
    addObj bEd : BEdge;
    addObj edgeStart : Start;
    addObj edgeEnd : End;
    addLink bEd.start.edgeStart;
    addLink bEd.end.edgeEnd;
    //quit if not found;
    first aEdgeStNode : ANode  from aEd by startNode
    else aEdgesProcessed;
    first mapBNode : BNode  from aEdgeStNode by mappedB
    else aEdgesProcessed;
```

```
      addLink edgeStart.node.mapBNode;
    first aEdgeEnNode : ANode  from aEd by endNode
    else aEdgesProcessed;
    first mapBNode : BNode  from aEdgeEnNode by mappedB
    else aEdgesProcessed;
    addLink edgeEnd . node. mapBNode;
    next aEd else aEdgesProcessed;
    goto loopAEdge;
label aEdgesProcessed;
end;
endTransformation;
```

This transformation can be rewritten to use object-oriented L0 constructs:

```
transformation graphsOO;

procedure ANode::mapToBNode();
  pointer b : BNode;
begin;
   addObj  b : BNode;
   addLink this . mappedB . b;
end;

procedure AEdge::mapToBEdge();
  pointer bEd : BEdge;
  pointer edgeStart  : Start;
  pointer edgeEnd    : End;

  pointer aEdgeStNode : ANode;
  pointer aEdgeEnNode   : ANode;
  pointer mapBNode : BNode;
begin;
    addObj bEd : BEdge;
    addObj edgeStart : Start;
    addObj edgeEnd : End;
    addLink bEd.start.edgeStart;
    addLink bEd.end.edgeEnd;

    first aEdgeStNode : ANode  from this by startNode
    else quit;
    first mapBNode : BNode from aEdgeStNode by mappedB
    else quit;
    addLink edgeStart.node.mapBNode;
    first aEdgeEnNode : ANode  from this by endNode
    else quit;
    first mapBNode : BNode from aEdgeEnNode by mappedB
```

```
      else quit;
      addLink edgeEnd . node. mapBNode;

      label quit;
  end;

  //main procedure Graph2Graph_OO();
  procedure Graph2Graph_OO();
    pointer a : ANode;
    pointer aEd : AEdge;
  begin;

  //copy nodes;
  first a : ANode else aNodeProcessed;
  label loopANode;

     call a.mapToBNode();

     next a else aNodeProcessed;
     goto loopANode;
  label aNodeProcessed;

  //copy edges;
  first aEd : AEdge else aEdgesProcessed;
  label loopAEdge;

     call aEd.mapToBEdge();

     next aEd else aEdgesProcessed;
     goto loopAEdge;
  label aEdgesProcessed;

  end;
  endTransformation;
```

It is obvious that the body of the procedure "Graph2Graph_OO" is now more readable (and thus maintainable) than that of "Graph2Graph".

## 2.4 L0 and Higher Level Model Transformation Languages

One of the main features of model transformation languages is pattern definition facilities. In transformation languages, the pattern is used to select a set of objects satisfying some known constraints (there are several kinds of constraints: type constraints, attribute value constraints, and structure constraints). L0+ does not provide pattern definition facilities. It is an intentional decision, because, on the one hand, an efficient implementation of patterns is one of the main challenges in the implementation of transformation languages [16], and ,on the other hand, pattern match can be relatively easily specified with the help of L0 imperative constructs.

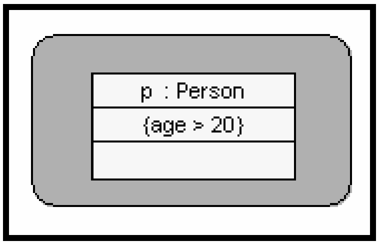


**Fig. 8.** Example of MOLA pattern

For instance, an example of MOLA **foreach** loop containing a pattern can be seen in Fig. **8.**. Semantically this means to iterate through all the instances of the class Person that satisfy the given attribute constraint (the value of the attribute “age” must be greater than 20). Despite the fact that L0 does not have explicit pattern definition facilities, the abovementioned MOLA pattern can be relatively easily specified in L0:

```
first p : Person else done;
  label loop_Person;
    var p.age > 20 else try_next_inst;
    //...;
    //process matched instance;
    //...;
  label try_next_inst;
  next p else done;
  goto loop_Person;
label done;
```

In more general terms, automatic pattern matching is a process that can be reduced to iteration through instances and checking a list of elementary conditions. These conditions are quite trivial – for example, check if the value of some attribute of the given object is equal to the corresponding value in the pattern specification. Another example – check whether or not there is a link with the given type between two objects. Consequently, if a language allows iterating through instances and there are conditional control flow operators, and it is possible to conduct abovementioned checks, it is possible to select a set of objects that satisfies the given constraints in this language. That allows us to assert that our language will be as powerful as a typical transformation language, but certainly transformation specification will be more verbose in this case.

# 3 An Extension of the Base Transformation Language L0 – Metamodel Processing Constructs

There are use cases for model transformation languages where an access not only to a model level, but also to a metamodel level is necessary. A substantial part of the existing transformation languages does not allow to process entities found at the metamodel level. To overcome this drawback, in the case of L0 we supplement it with constructs for metamodel processing, thus obtaining the language L0+.

## 3.1 Choosing Metamodel Processing Constructs

To allow transformation developer to process metamodels, we provide constructs to work with concepts defined in the metamodel found in figure 1. These are:

- classes
- attributes
- generalizations hierarchies
- associations
- enumerations and enumeration literals

Language users should have the possibility to create new, delete the existing and iterate through the existing elements of the metamodel. To satisfy these requirements, new commands for processing metamodel elements are introduced. These elements are identified by their names or by combinations of names.

The first activity a metamodel processing usually starts from, is the creation of the metamodel. While constructing the metamodel, the user can create classes, attributes of these classes, and associations (including composition) between classes. It is possible to create generalization hierarchies as well. In a similar way users can delete classes, attributes, associations, and generalization hierarchies. Precise syntax for these commands can be found in section 3.2.1.

The next group of commands deals with metamodel element scanning. Taking into account the fact that metamodel elements are identified by their names or by combinations of names, iterating through the elements of the metamodel actually means to iterate through the names of these elements. For example, to traverse classes of the metamodel, the commands **firstClass** and **nextClass** can be used. The semantics of these commands is close to the semantics of the ordinary **first** and **next** commands (again, precise syntax can be found in section 3.2.2.). Analogous commands are introduced for scanning each kind of metamodel elements:

- associations starting from the given class
- direct associations starting from the given class
- attributes of the given class
- subclasses of the given class
- enumerations
- enumeration elements

With the word "direct" we understand associations and attributes that are defined in the given class and not in superclasses of this class.

Using the abovementioned L0+ constructs, it is possible to dynamically explore and modify an arbitrary (potentially unknown at the compile time) metamodel. Now let us get back to the model level.

When working with a model which corresponds to a metamodel, we are not limited with only those metamodel elements that are known at the compile time. Since we have the possibility to explore an arbitrary metamodel, we need to have a way both to reference the name of an arbitrary element of this metamodel, and to reference objects of an arbitrary class (this can be done with the help of a **Void** pointer). With such a possibility it is sufficient to be able to process arbitrary models of an arbitrary metamodel.

For example, with the abovementioned L0+ constructs it is possible to create a new class at runtime and populate it with instances.

```
var currClassName : String;
pointer x  : Void;
//...;
addClass (currClassName);
addObj x : (currClassName);
//...;
```

The situation with attributes is special in some way, because problems with expression compilation can arise in case the attribute type is not known. One can notice that the type of a dynamically created attribute is unknown only at the compile time, but is known to the programmer creating this program. That is why the values of dynamically created attributes can be manipulated only as strings. To get the value of an attribute of a previously unknown type, a special form (in which the name of the attribute is specified as a String variable) of the command getting the attribute value should be used. For instance, **setVar** attrValStr = x.(attrNameStr); Here, x is a pointer name and attrNameStr is a **String** variable containing the name of the attribute. attrValStr is a String variable as well. After execution of this command, the value of attrValStr will be equal to the value of the corresponding attribute of the object encoded as a string. If attrNameStr value is equal to "weight" and x points to an object for which the value of an **Integer** attribute named "weight" is equal to 10, then attrValStr will be equal to a String value "10".

It should be noted that for **Void** pointers it is the only way to receive the value of an attribute. To simplify conversions between different representation forms of values, special built-in functions are introduced:

- IsInt ( str : String ) : Boolean;
- IsReal ( str : String ) : Boolean;
- IsBool ( str : String ) : Boolean;
- StrToInt ( str : String ) : Integer;
- StrToBool ( str : String ) : Bool;
- StrToReal ( str : String ) : Real;
- IntToStr ( i : Integer ) : String;
- BoolToStr ( b : Bool) : String;
- RealToStr ( r : Real ) : String;

### 3.2 The Definition of Metamodel Processing Commands

Before giving a precise definition of L0+ commands, it should be stressed that the names of metamodel elements can be specified in two ways (a similar situation was with the names of metamodel elements in the case of L0):

- as literals, for instance, **addClass** Person;.
- as String variables, for instance, **addObj** x : (s); In this case the name of the metamodel element will be equal to the value of the corresponding String variable.

#### 3.2.1 Metamodel Building Commands

This part of language definition describes commands for dynamic meta-model building.

1. **addClass** <clName>**;**
   Dynamically creates a class with a specified name. If a class with specified name already exists, a warning message is issued.

   1.1. **addClass (**<strVarClName>**);**

2. **addAttr** <clName>**.**<attrName>**:**<ElementaryTypeName>**;**
   Dynamically creates an attribute belonging to the specified class with a specified name and type. If an attribute with specified properties already exists, a warning message is issued.

   2.1. **addAttr (**<strVarClName>**).(**<strVarAttrName>**):**
   **(**<strVarElemTypeName>**);**

3. **addAssoc** <clName>**.{ordered}**<card><roleName>**/**
   <roleName><card>**{ordered}.**<clName>**;**
   Dynamically creates an association with specified properties. If an association with specified properties already exists, a warning message is issued.

   3.1. **addAssoc** (<strVarClName>**).{ordered}**<card>**(**<strVarRoleName>**)/**
   **(**<strVarRoleName>**)**<card>**{ordered}.(**<strVarClName>**);**

4. **addCompos** <compositeClName>**.{ordered}**<card><roleName>**/**
   <roleName><card>**{ordered}.**<partClName>**;**
   Dynamically creates a composition with specified properties. If a composition with specified properties already exists, a warning message is issued.

   4.1. **addCompos (**<strVarClName>**).{ordered}**<card>**(**<strVarRoleName>**)/**
   **(**<strVarRoleName>**)**<card>**{ordered}.(**<strVarClName>**);**

5. **addRel** <subClName>**.subClassOf.**<superClName>;
   Dynamically creates a generalization relation between the specified classes. If a generalization relation between these classes already exists, a warning message is issued.

   5.1. **addRel (**<strVarSubClName>**).subClassOf. (**<strVarSuperClName>**)**;

6. **deleteClass** <clName>**;**
   Deletes a class with a specified name.

   6.1. **deleteClass (**<strVarClName>**);**

7. **deleteAttr** <clName>**.**<attrName>**;**
   Deletes an attribute with the given properties.

   7.1. **deleteAttr (**<strVarClName>**).(**<strVarAttrName>**);**

8. **deleteAssoc** <clName>**.**<roleName>**.**<clName>**;**
   Deletes an association  with a given role name between the given classes.

   8.1. **deleteAssoc (**<strVarClName>**).(**<strVarRoleName>**).(**<strVarClName>**);**

9. **deleteRel** <subClName>**.subClassOf.**<superClName>;
   Deletes a generalization relation between the specified classes.

   9.1. **deleteRel (**<strVarSubClName>**).subClassOf. (**<strVarSuperClName>**)**;

10. **addEnum** <enumName>**:{** <enumElem1>, <enumElem2>, …  };
    Dynamically creates an enumeration with a specified name and specified enumeration literals.

    10.1. **addEnum (**<strVarEnumName>**):{<**enumElemName**>,**
    **(**strVarEnumElemName**),…};**

11. **deleteEnum** <enumName>;
    Deletes an enumeration with a specified name.

    11.1. **deleteEnum** (<strVarEnumName >);

12. **addEnumElem**  <enumElemName> **to** <enumName>;
    Adds an enumeration literal to an enumeration.

    12.1. **addEnumElem**  (<strVarEnumElemNameIn>) **to** (<strVarEnumNameIn>);

13. **deleteEnumElem**  <enumElemName> **from** <enumName>;
    Removes an enumeration literal form an enumeration.

13.1. **deleteEnumElem** (<strVarEnumElemNameIn>) **from** (<strVarEnumNameIn>);

### 3.2.2 Meta-Model Element Scanning Commands

This part of language definition describes commands for scanning meta-model elements.

1. **firstClass** <strVarClNameOut> **else** <labName>**;**
   Stores the name of the first class (ordering is implementation dependent) in the <strVarClNameOut>. If there are no classes, then <strVarClNameOut> value is not changed and execution control is transferred to <labName>. Typically, this command is used in combination with the **nextClass** command to iterate through all class names.

2. **nextClass** <strVarClNameOut> **else** <labName>**;**
   Stores the name of the next class which has not yet been visited (iterated) in <strVarClNameOut>. If there is no such class, <strVarClNameOut> value is not changed, and execution control is transferred to <labName>.

3. **firstAssoc** <clName>**.**<strVarRoleNameOut>**/** <strVarInvRoleNameOut>**.**<strVarClNameOut> **else** <labName>**;**
   Stores the role name, inverse role name, and target class name of the first association (ordering is implementation dependent) starting from <clName> in <strVarRoleNameOut>, <strVarInvRoleNameOut> and <strVarClNameOut>, respectively. If there are no associations starting from <clName>, then <strVarRoleNameOut>, <strVarInvRoleNameOut> and <strVarClNameOut> values are not changed and execution control is transferred to <labName>. Typically, this command is used in combination with the **nextAssoc** command to iterate through all associations starting from the given class (or from ancestors of this class).
   3.1. **firstAssoc (**<strVarClNameIn>**).**<strVarRoleNameOut>**/** <strVarInvRoleNameOut>**.**<strVarClNameOut> **else** <labName>**;**

4. **firstAssocDirect** <clName>**.**<strVarRoleNameOut>**/** <strVarInvRoleNameOut>**.**<strVarClNameOut> **else** <labName>**;**
   This command is similar to the previous one, the difference is that it takes into account only those associations which are defined exactly for this class and ignores associations which are defined in ancestor classes.
   4.1. **firstAssocDirect (**<strVarClNameIn>**).**<strVarRoleNameOut>**/** <strVarInvRoleNameOut>**.**<strVarClNameOut> **else** <labName>**;**

5. **nextAssoc** <clName>**.**<strVarRoleNameOut>**/** <strVarInvRoleNameOut>**.**<strVarClNameOut> **else** <labName>**;**
   Stores the role name, inverse role name, and target class name of the next association starting from <clName>, which has not yet been visited (iterated), in

<strVarRoleNameOut>, <strVarInvRoleNameOut> and <strVarClNameOut>, respectively. If there are no such associations, <strVarRoleNameOut>, <strVarInvRoleNameOut>, <strVarClNameOut> values are not changed, and execution control is transferred to <labName>.

5.1. **nextAssoc** **(**<strVarClNameIn>**).**<strVarRoleNameOut>**/** <strVarInvRoleNameOut>**.**<strVarClNameOut> **else** <labName>**;**

6. **nextAssocDirect** <clName>**.**<strVarRoleNameOut>**/** <strVarInvRoleNameOut>**.**<strVarClNameOut> **else** <labName>**;**
Similar to the previous one, but associations from ancestors are ignored.
6.1. **nextAssocDirect** **(**<strVarClNameIn>**).**<strVarRoleNameOut>**/** <strVarInvRoleNameOut>**.**<strVarClNameOut> **else** <labName>**;**

7. **firstAttr** <clName>**.**<strVarAttrNameOut>**.**<strVarAttrTypeNameOut > **else** <labName>**;**
Stores the name and type name of the first attribute (ordering is implementation dependent) of <clName> in <strVarAttrNameOut>, <strVarAttrTypeNameOut >, respectively. If <clName> has no attributes, then <strVarAttrNameOut>, <strVarAttrTypeNameOut > values are not changed and execution control is transferred to <labName>. Typically, this command is used in combination with the **nextAttr** command to iterate through all attributes of the given class (including ancestor attributes).
7.1. **firstAttr** **(**<strVarClNameIn>**).**<strVarAttrNameOut>.<strVarAttrTypeNameOut> **else** <labName>**;**

8. **firstAttrDirect** <clName>**.**<strVarAttrNameOut>**.**<strVarAttrTypeNameOut > **else** <labName>**;**
This command is similar to the previous one, the difference is that it takes into account only those attributes which are defined exactly in this class and ignores attributes which are defined in ancestor classes.
8.1. **firstAttrDirect** **(**<strVarClNameIn>**).**<strVarAttrNameOut>**.**<strVarAttrTypeNameOut> **else** <labName>**;**

9. **nextAttr** <clName>**.**<strVarAttrNameOut>**.**<strVarAttrTypeNameOut > **else** <labName>**;**
Stores the name and type name of the next attribute of <clName>, which has not yet been visited (iterated), in <strVarClNameOut> and <strVarAttrTypeNameOut>, respectively. If there are no such attributes, <strVarClNameOut> and <strVarAttrTypeNameOut > values are not changed and execution control is transferred to <labName>.
9.1. **nextAttr** **(**<strVarClNameIn>**).**<strVarAttrNameOut>**.**<strVarAttrTypeNameOut> **else** <labName>**;**

10. **nextAttrDirect** <clName>**.**<strVarAttrNameOut>**.**<strVarAttrTypeNameOut> **else** <labName>**;**
    Similar to the previous one, the difference is that it takes into account only those attributes which are defined exactly in this class and ignores attributes which are defined in ancestor classes.
    10.1. **nextAttrDirect** **(**<strVarClNameIn>**)**.<strVarAttrNameOut>**.**<strVarAttrTypeNameOut> **else** <labName>**;**

11. **firstSubClass** <superClName>**.**<strVarSubClNameOut> **else** <labName>**;**
    Stores the name of the first subclass (ordering is implementation dependent) in <strVarSubClNameOut>. If there are no subclasses, then <strVarSubClNameOut> value is not changed and execution control is transferred to <labName>. Typically, this command is used in combination with the **nextSubClass** command to iterate through all subclasses.
    11.1. **firstSubClass** **(**<strVarSuperClNameIn>**).**<strVarSubClNameOut> **else** <labName>**;**

12. **nextSubClass** <superClName>**.**<strVarSubClNameOut> **else** <labName>**;**
    Stores the name of the next subclass of <superClName>, which has not yet been visited (iterated), in <strVarSubClNameOut>. If there are no such classes, <strVarSubClNameOut> value is not changed, and execution control is transferred to <labName>.
    12.1. **nextSubClass** **(**<strVarSuperClNameIn>**).**<strVarSubClNameOut> **else** <labName>**;**

13. **firstEnum** <strVarEnumNameOut> **else** <labName>;
    Stores the name of the first enumeration (ordering is implementation dependent) in <strVarEnumNameOut>. If there are no enumerations, then <strVarEnumNameOut> value is not changed and execution control is transferred to <labName>. Typically, this command is used in combination with the **nextEnum** command to iterate through all enumeration names.

14. **nextEnum** <strVarEnumNameOut> **else** <labName>;
    Stores the name of the next enumeration which has not yet been visited (iterated) in <strVarEnumNameOut>. If there are no such enumerations, <strVarEnumNameOut> value is not changed and execution control is transferred to <labName>.

15. **firstEnumElem** <enumName>**.**<strVarEnumElemNameOut> **else** <labName>;
    Stores the name of the first <enumName> enumeration literal (ordering is implementation dependent) in the <strVarEnumElemNameOut>. If there are no enumeration literals in <enumName>, then <strVarEnumElemNameOut> value is not changed and execution control is transferred to <labName>. Typically, this command is used in combination with the **nextEnumElem** command to iterate through all given enumeration literals.

15.1. **firstEnumElem (**<strVarEnumNameIn>**).**<strVarEnumElemNameOut> **else** <labName>;

16. **nextEnumElem** <enumName>**.**<strVarEnumElemNameOut> **else** <labName>;
Stores the name of the next <enumName> enumeration literal, which has not yet been visited (iterated) in <strVarEnumElemNameOut>. If there are no such enumeration literals, <strVarEnumNameOut> value is not changed and execution control is transferred to <labName>.
16.1. **nextEnumElem (**<strVarEnumNameIn>**).(** <strVarEnumElemNameOut>**)** **else** <labName>;

17. **existsClass** <className> **else** <labName>**;**
If a class with a specified name exists, execution control is transferred to the next command, otherwise execution control is passed to <labName>.
17.1. **existsClass (**<strVarClassNameIn>**)** **else** <labName>**;**

18. **existsEnum** <enumName> **else** <label>**;**
If an enumeration with a specified name exists, execution control is transferred to the next command, otherwise execution control is passed to <labName>.
18.1. **existsEnum (**<strVarEnumNameIn>**)** **else** <labName>**;**

19. **existsEnumElem** <enumName>**.**<enumElemName> **else** <label>**;**
If an enumeration with a specified name has an enumeration literal with a specified name, execution control is transferred to the next command, otherwise execution control is passed to <labName>.
19.1. **existsEnumElem** (<strVarEnumNameIn>**).(**<strVarEnumElemNameIn>**)** **else** <labName>**;**

20. **existsAttr** <clName>**.**<attrName>**.**<typeName> **else** <labName>**;**
If an attribute with specified properties exists, execution control is transferred to the next command, otherwise execution control is transferred to <labName>.
20.1. **existsAttr**
**(**<strVarClNameIn>**).(**<strVarAttrNameIn>**).(**<strVarAttrTypeNameIn>**)** **else** <labName>**;**

21. **existsAssoc** <clName>**.**<roleName>**.**<clName> **else** <label>**;**
If an association with specified properties exists, execution control is transferred to the next command, otherwise execution control is transferred to <labName>.
21.1. **existsAssoc**
**(**<strVarClNameIn>**).(**<strVarRoleNameIn>**).(**<strVarClNameIn>**)** **else** <labName>**;**

22. **existsCompos** <clName>**.**<roleName>**.**<clName> **else** <label>**;**
If a composition with specified properties exists, execution control is transferred to the next command, otherwise execution control is transferred to <labName>.

22.1. **existsCompos**
**(**<strVarClNameIn>**).(**<strVarRoleNameIn>**).(**<strVarClNameIn>**)**
**else** <labName>**;**

23. **existsRel** <subClName>**.subClassOf.**<superClName> **else** <label>**;**
If there is a generalization relationship between specified classes,execution control is transferred to the next command, otherwise execution control is transferred to <labName>.
23.1. **existsRel (**<strVarSubClNameIn>**).subClassOf. (**<strVarSuperClNameIn>**)** **else** <labName>**;**

## 4 Implementation of L0+

### 4.1 Selection of the Runtime Environment

The implementation of a transformation language starts from the selection of a runtime environment. The selection of the run-time environment is not limited to the selection of a target language, because we have to provide a way of storing and accessing persistent data (metamodel and its instances) while implementing a model transformation language.

Quite natural choices in this situation are in-memory metamodel-based data stores (repositories) [17, 18, 19].

For the implementation of L0 and L0+, the in-memory data store developed at the IMCS was selected [19]. This repository proved to be reasonably efficient. For instance, in [19] it is shown, that this data store is at least as efficient in typical instance selection tasks as one of the most popular open-source RDF data stores Sesame [20].

The API of the chosen repository is implemented as a library of C++ functions. This library provides the following possibilities:

- a set of functions for creation and deletion of metamodel elements, as well as iteration through them;
- model processing functions that can be divided into two subcategories:
  - functions for creation and deletion of instances (objects and links) and functions for getting/setting the value of an attribute, for example:
    ```
    long CreateObject(long ObjectTypeId);
    int DeleteObjectHard(long ObjectId);
    int CreateLink(long LinkTypeId, long ObjectId1, long ObjectId2);
    int DeleteLink(long LinkTypeId, long ObjectId1, long ObjectId2);
    ```
  - efficient searching functions (these functions are based on sophisticated indexing mechanisms):
    ```
    int GetObjectNum(long ObjectTypeId);
    long GetObjectIdByIndex(long ObjectTypeId, int Index);
    int GetLinkedObjectNum(long ObjectId, long LinkTypeId);
    ```

long GetLinkedObjectIdByIndex(long ObjectId, long LinkTypeId, int Index);

The main advantage of the use of this repository is that there are close counterparts for a substantial part of L0 and L0+ commands in the repository API. It means that it will be quite easy to implement these commands, and there will be no substantial difference (at least in the aspect of efficiency) between a hand-written and a compiler-generated code.

Taking into account the fact that the selected repository provides a C++ API, C++ was chosen as a target language for L0 and L0+ compilation. Since effective C++ compilers are known, it is possible to generate a C++ code without thinking of its extensive optimization, because all optimizations of the C++ code will be done by the C++ compiler. Thus we can omit final phases of traditional compilers i.e. intermediate code generation and optimization.

### 4.2 Compilation Schema

L0 and L0+ compilation process consists of four phases:

- Preprocessing phase, when compiler directives (for example, "useMM") are analyzed;
- Lexical analysis phase, when transformation program is divided into lexemes;
- Syntactical analysis phase, when the list of lexemes of the program is divided into groups of lexemes corresponding to definite commands;
- Code generation phase, when C++ code is generated from a group of lexemes, corresponding to a definite command.

L0 and L0+ languages do not contain recursive constructs. Moreover, the start and the end of every command are easily identifiable. Because of these two facts, syntactical and lexical analysis is quite simple and will not be described further.

C++ code generation seems to be more interesting. Let us start with some general principles. Firstly we have to understand how to compile general constructs: subprograms (with corresponding parameters passing mechanisms), control flow commands, elementary variables and pointers to class instances. It is not difficult to spot similarities between C++ functions and L0 subprograms, C++ elementary variables and L0 elementary variables, C++ control flow possibilities, and L0 control flow possibilities.

However, the situation with L0 pointers (references to class instances) is somewhat more special, because in C++ there is no direct way of simulating them. To represent L0 pointers in C++ program, we define a C++ class L0_Var_2 with operations corresponding to L0 commands. This class has the following operations:

- bool moveNext();
- bool isNull();
- bool setFirstToRoot(const string & className ),;
- bool setFirstFrom( const L0_Var_2 & rhs , const string & assocName );
- bool setStringAttributeValue(const string & attrName, const string & newValue);

etc.

The implementation of the class L0_Var_2 is based on model processing functions from the Repository API.

Now, when it is clear how individual constructs are compiled, an overall compilation schema can be given:

- Every L0 subprogram is compiled to a corresponding C++ function;
- Every elementary L0 variable is compiled to a corresponding C++ variable;
- Every L0 pointer is compiled to an object of the C++ class L0_Var_2;
- Every L0 command call on a given L0 pointer is compiled to a corresponding C++ method call on a C++ object.

The situation with L0+ commands is similar. Every L0+ command is mapped to a C++ function that relies on the metamodel processing functions from the Respository API.

## 4.3 Elementary Tracing Facilities

If a program flow is specified with conditional and unconditional control flow transfer operators (i.e. structured programming constructs are not used), then it becomes rather difficult to trace program execution flow (as a consequence, it is difficult to debug these programs). L0 does not provide structured programming constructs. That is why typical errors in L0 programs are related to incorrectly specified control flows. To simplify L0 transformation debugging, L0 compiler can generate code in debugging mode. When a program generated in this mode runs, it logs execution path and other significant information. For example, the following program finds the least of three numbers.

```
transformation traceDemo;
DEBUG ON;
main procedure p();
  var i1 : Integer;
  var i2 : Integer;
  var i3 : Integer;
  var min: Integer;
begin;
  setVar i1 = 10;
  setVar i2 = 8;
  setVar i3 = 6;

  var i1 < i2 else i2LessThani1;
    var i1 < i3 else i3LessThani1;
     setVar min = i1;
     return;

    label i3LessThani1;

     setVar min = i3;
```

```
        return;

    label i2LessThani1;

    var i2 < i3 else i3LessThani2;
      setVar min = i2;
      return;

      label i3LessThani2;

      setVar min = i3;
      return;
  end;

  endTransformation;
```

When running this program in debug mode, it will produce the following output:

```
     12 : procedure p

     18 : setVar i1 = 10
       i1 = 10
     19 : setVar i2 = 8
       i2 = 8
     20 : setVar i3 = 6
       i3 = 6
     22 : var i1 < i2 else i2LessThani1
     32 : label i2LessThani1
     34 : var i2 < i3 else i3LessThani2
     38 : label i3LessThani2
     40 : setVar min = i3
       min = 6
     41 : return
   Return from p
```

To implement this functionality, generated C++ code is appended with some code fragments logging activities of the program.

For example, when L0 compiler receives a command "**setVar** `i1 = 10;`", it emits the following C++ code: "`elemVar___p_i1_ =  10;`". But if L0 compiler is generating debug code, it will emit substantially different code for the same L0 command:

```
„
Logger::inst().wrtLineNum( 17 );
  Logger::inst().wrtLine( "setVar i1 = 10" );
elemVar___p_i1_ =  10 ;
Logger::inst().wrtPref( );  Logger::inst().wrt( "i1 =
");
```

```
Logger::inst().wrtInt(elemVar___p_i1_ );
Logger::inst().newLine();
".
```

## 5 Conclusions

This paper was devoted to the problems of effective implementation of model transformation languages. It was stated that direct implementation of high level transformation languages is a difficult and costly process that does not guarantee effectiveness of the obtained implementation. It is believed that a more optimal way to implement high level model transformation language is to use bootstrapping. Bootstrapping in its turn is not possible without an effective base language.

One of the main results of this paper is the definition of a new low level model transformation language L0+, that can be used as a base language in bootstrapping process. L0 is called a base language, because:

- it contains minimal, but sufficient model transformation constructs;
- an effective implementation for this language does exist.

One more reason justifying L0+ usage in bootstrapping process is the increased portability of a high level language being compiled to L0+. L0+ naturally becomes a kind of a repository abstraction layer, meaning that if we want to port an implementation of a high level language to another target environment (that uses L0+ as a target language), it is sufficient to port only the implementation of L0+.

The second notable result of this paper is principles of an effective implementation of this language. According to these principles, an effective implementation of L0+ was obtained.

L0+ was designed to be of as low level as possible to simplify its implementation, and it does not contain some the of constructs (mainly, patterns) found in more advanced languages. Nevertheless, this language is also used for a direct development of model transformations.

## References

1. J. Barzdins, A. Kalnins, E. Rencis, S. Rikacovs, Model Transformation Languages and their Implementation by Bootstrapping Method, Pillars of Computer Science: Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday, Arnon Avron, Nachum Dershowitz, and Alexander Rabinovich, editors, Lecture Notes in Computer Science, vol. 4800, Springer-Verlag, Berlin, 2008.
2. A.Kleppe, J. Warmer, W. Bast, MDA Explained: The Model Driven Architecture -- Practice and Promise, Addison Wesley, 2003.
3. Request for Proposal : MOF 2.0 Query / Views / Transformations RFP, URL: http://www.omg.org/docs/ad/02-04-10.pdf
4. OMG, MOF 2.0 Query/View/Transformation Specification. URL: http://www.omg.org/docs/ptc/07-07-07.pdf

5. Agrawal A., Karsai G, Shi F.: Graph Transformations on Domain-Specific Models. Technical report, Institute for Software Integrated Systems, Vanderbilt University, ISIS-03-403, November 2003.
6. A.Kalnins, J. Barzdins, E.Celms. Basics of Model Transformation Language MOLA. - ECOOP 2004 (Workshop on Model Transformation and execution in the context of MDA), Oslo, Norway, June 14-18, 2004.
7. ATL. URL: http://www.sciences.univ-nantes.fr/lina/atl/
8. MTF. URL: http://www.alphaworks.ibm.com/tech/mtf
9. Tefkat.URL: http://www.dstc.edu.au/Research/Projects/Pegamento/tefkat/
10. E.Rencis. Model Transformation Languages L1, L2, L3 and their Implementation, Scientific Papers. University of Latvia, "Computer Science and Information Technologies", 2008.
11. J. Barzdins, A. Zarins, K. Cerans, A. Kalnins, E. Rencis, L. Lace, R. Liepins, A. Sprogis, GrTP: Transformation Based Graphical Tool Building Platform, MODELS 2007, Workshop on Model Driven Engineering Languages and Systems, 2007.
12. VIATRA2 URL: http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/VIATRA2/index.html
13. Meta Object Facility (MOF) Specification Version 2.0 URL : http://www.omg.org/docs/formal/06-01-01.pdf
14. Meta Object Facility (MOF) Specification Version 1.4.1 URL : http://www.omg.org/docs/formal/05-05-05.pdf
15. Deborah J. Armstrong, The quarks of object-oriented development, Communications of the ACM, 2006.
16. A.Sostaks., A.Kalnins. The implementation of MOLA to L3 compiler, Scientific Papers University of Latvia, "Computer Science and Information Technologies", 2008.
17. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.: Eclipse Modeling Framework. Addison Wesley, 2003.
18. Metadata Repository (MDR). URL: http://mdr.netbeans.org/
19. J. Barzdins, G. Barzdins, R. Balodis, K. Cerans, A. Kalnins, M. Opmanis, K. Podnieks. Towards Semantic Latvia. Communications of the 7th International Baltic Conference on Databases and Information Systems (Baltic DB&IS'2006 , 2006), pp. 203-218.
20. Sesame. URL: http://www.openrdf.org/

LATVIJAS UNIVERSITĀTES RAKSTI. 2008, 733. sēj.:
DATORZINĀTNE UN INFORMĀCIJAS TEHNOLOĢIJAS 103.–139. lpp.

# Model Transformation Languages L1, L2, L3 and Their Implementation

Edgars Rencis[1]
University of Latvia, IMCS, 29 Raiņa Blvd, Rīga, Latvia
Edgars.Rencis@lumii.lv

**Abstract.** In this paper a family of model transformation languages L1, L2, and L3 following the language L0 is introduced. The first language L0, not being part of this paper, is very simple and serves as a base language. It is implemented through an efficient compiler to C++ [1]. Each of the next languages L1, L2, and L3 is an extension of the previous one, and they are implemented by the bootstrapping method based on the language L0, that is, three compilers are written in L0: from L1 to L0, from L2 to L1, and from L3 to L2. The language L1 contains powerful pattern definition facilities, L2 – loops, and L3 – the branching facility. The language L3 is considered to be both sufficiently easy-to-use to serve as an intermediate language in the implementation of higher-level transformation languages, and expressive enough to be used in real model transformation tasks. The presented paper is an extended version of sections 4 – 6 of [10].

**Keywords.** Model transformation languages, L0, Lx, L1, L2, L3, compiler, bootstrapping.

## 1 Introduction

Although model transformation languages are the very heart of the MDA [2] – the most advanced architecture used to build systems nowadays – the implementation of various model transformation languages encountered in the world has not been very extensively researched. Actually, there exist only a few attempts to implement a model transformation language through some other language by using bootstrapping method [3-5]. The goal of this paper is to demonstrate the use of such an approach. It includes defining a sequence of model transformation languages and then implementing these languages by bootstrapping method one through another until the base transformation language is reached. In addition, another goal is to propose a language L3 that is, on the one hand, simple enough to be easily implementable, and, on the other hand, expressive enough to be used in practical model transformation tasks. Some the of results expounded on in this paper are also briefly outlined in [10].

[1] Partially supported by ESF (European Social Fund),
project 2004/0001/VPD1/ESF/PIAA/04/NP/3.2.3.1/0001/0001/0063

The structure of this paper is the following:

1) the base transformation language L0 is described in Section 2 – it is very simple and contains only the basic transformation facilities; an efficient compiler to C++ is built for this language [1];
2) a sequence of model transformation languages L0', L1, L2, and L3 is introduced in Section 3; every next language of the so called Lx family is made based on the previous one by adding some new features; both the metamodel and the textual syntax is provided for each of languages; the language L3 is of a sufficiently high level to be used in practical model transformation tasks, however, it is still sufficiently easy-to-use to be used as an intermediate language in the implementation of higher-level model transformation languages (for example, the graphical transformation language MOLA [6,7,15]) by using the bootstrapping method;
3) the implementation of languages L0', L1, L2, and L3 is provided in Section 4; every next language is compiled to the previous one using the bootstrapping method.

## 2 Model transformation language L0

L0 is a textual model transformation language. It offers simple commands to work with arbitrary fixed instances of a given metamodel (for example, a command for creating a new instance, deleting an instance, getting and setting attribute's values, making and deleting links between instances, searching for instances etc.) and to handle simple control flows (it is done using the so called "goto" commands, as well as "else" branches that are attached to some L0 commands). To store persistent data, an in-memory repository has been developed at the University of Latvia, Institute of Mathematics and Computer Science [8].

An effective compiler from the language L0 to the language C++ has been developed. It means that it is possible to translate a program written in L0 into a C++ code, which can further be compiled to a ".dll" file. When it is done, the resulting ".dll" file can be executed on a metamodel given by the user.

A more detailed description of the language L0 is available in [1], however, an overview of this language (commands + metamodel) is given in the next sections of this paper in order to make this paper understandable without the necessity to read the abovementioned paper.

### 2.1 Command of the Transformation Language L0

Base model transformation language L0 is a fully procedural language and contains the following commands (that can be found in the body of any procedure or function) [9]:

**1)** **call** <subProgName> **(**<actualParamList>**)** – calls the subprogram with the given parameters;
**2)** **return** – returns the control to the calling program;
**3)** **return** <identifier> – returns the value of <identifier> to the calling program;

4) **first** <pointerName> **:** <className> [ **else** <labelName> ] – positions the pointer <pointerName> to an arbitrary instance of the class <className>. If there are no instances of the given class, the control is given to the label <labelName> ;
5) **first** <pointerName1> **:** <className> **from** <pointerName2> **by** <roleName> [ **else** <labelName> ] – positions the pointer <pointerName1> to such an arbitrary instance of the class <className> that is reachable from the pointer <pointerName2> by the role <roleName>. If there are no such instances, the control is given to the label <labelName>. After the command has been executed, the value set of the pointer <pointerName1> is limited to exactly those instances of the class <className>, which are reachable from the pointer <pointerName2> by the role <roleName> ;
6) **next** <pointerName> [ **else** <labelName> ] – positions the pointer <pointerName> to the next instance that satisfies conditions raised by the respective "first" command (the previous one with the same pointer <pointerName>) and that is not yet visited by commands "first" or "next". If there are no such instances, the control is given to the label <labelName> ;
7) **goto** <labelName> – gives the control the label <labelName> ;
8) **label** <labelName> – defines the label <labelName> ;
9) **addObj** <pointerName> **:** <className> – creates a new instance of the class <className> ;
10) **addLink** <pointerName1> **.** <roleName> **.** <pointerName2> – creates a link between instances <pointerName1> and <pointerName2> with the role name <roleName> at the end of the instance <pointerName2> ;
11) **deleteObj** <pointerName> – deletes the instance <pointerName> ;
12) **deleteLink** <pointerName1> **.** <roleName> **.** <pointerName2> – deletes the link between instances <pointerName1> and <pointerName2> with the role name <roleName> at the end of the instance <pointerName2> ;
13) **setPointer** <pointerName1> = <pointerName2> – positions the pointer <pointerName1> to the instance pointed to by the pointer <pointerName2> ;
14) **setPointerF** <pointerName> = <funcName> **(**<actualParamList>**)** – positions the pointer <pointerName> to the instance returned by the function <funcName> called with the given parameters ;
15) **setVar** <varName> = <binExpr> – sets the value of the variable <varName> to the value of the binary expression <binExpr> ;
16) **setVarF** <varName> = <funcName> **(**<actualParamList>**)** – sets the value of the variable <varName> to the value returned by the function <funcName> called by given parameters ;
17) **setAttr** <pointerName> **.** <attrName> = <binExpr> – sets the value of the attribute <attrName> of the instance <pointerName> to the value of the binary expression <binExpr> ;
18) **type** <pointerName> == <className> [ **else** <labelName> ] – if the pointer <pointerName> points to the instance of the class <className>, the control is given to the next command, otherwise the control is given to the label <labelName>. Inequality ("!=") is allowable instead of the equality as well;

**19)** **var** <varName> == <binExpr> [ **else** <labelName> ] – if the value of the variable <varName> is equal to the value of the binary expression <binExpr>, the control is given to the next command, otherwise the control is given to the label <labelName>. Any other comparison operators ("<", "<=", ">", ">=",or "!=") are allowable instead of the equality as well;

**20)** **pointer** <pointerName1> == <pointerName2> [ **else** <labelName> ] – if pointers <pointerName1> and <pointerName2> point to the same instance, the control is given to the next command, otherwise the control is given to the label <labelName>. Inequality ("!=") is allowable instead of the equality as well;

**21)** **attr** <pointerName> **.** <attrName> == <binExpr> [ **else** <labelName> ] – if the value of the attribute <attrName> of the instance <pointerName> is equal to the value of the binary expression <binExpr>, the control is given to the next command, otherwise the control is given to the label <labelName>. Any other comparison operators ("<", "<=", ">", ">=" or "!=") are allowable instead of the equality as well;

**22)** **link** <pointerName1> **.** <roleName> **.** <pointerName2> [ **else** <labelName> ] – if there exists a link with the role name <roleName> at the end of the instance <pointerName2> between instances <pointerName1> and <pointerName2>, the control is given to the next command, otherwise the control is given to the label <labelName> ;

**23)** **nolink** <pointerName1> **.** <roleName> **.** <pointerName2> [ **else** <labelName> ] – if there does not exist a link with the role name <roleName> at the end of the instance <pointerName2> between instances <pointerName1> and <pointerName2>, the control is given to the next command, otherwise the control is given to the label <labelName> ;

**24)** **DEBUG_ON –** turns on the debugging mode;

**25)** **DEBUG_OFF** – turns off the debugging mode.

Since the transformation language L0 is a strongly typified language, it is required that any variable is declared in a separate block in each procedure or function in the following manner:

1) **var** <varName> **:** <typeName> – declares a variable with a primitive data type (*Integer*, *Real*, *String* or *Boolean*)
2) **pointer** <pointerName> **:** <className> – declares a pointer to instances of the class <className>

There actually exists an extension of the language L0 – language L0+. In the language L0+, commands working in metamodel level are added. Namely, it is possible, for example, to make and delete classes, associations and attributes in L0+. So it is possible to make a specific metamodel in L0+ and then to execute the program written in L0 (or L0+) on this metamodel. As it is not the goal of this paper, commands of the language L0+ have not been discussed here.

## 2.2 The Metamodel of the Language L0

Since metamodels of the further introduced transformation languages will be based on the metamodel of the language L0, it is necessary to discuss this metamodel in detail (see Fig. 1).

The metamodel of the language L0 is quite intuitive – every transformation program (an instance of the class "Transformation") contains procedures/functions that in their turn contain command blocks starting with one command while every command does not have more than one next command. Every procedure/function has its variable definition block as well.

In the language L0, four types of commands exist:

1) instances of the class "GotoCom" – control flow commands;
2) instances of the class "FNCom" – instance searching commands ("first" and "next");
3) instances of the class "ECom" – commands with a possible "else" branch ("type", "var", "pointer", "attr", "link" and "noLink");
4) instances of the class „SCom" – other commands ("call", "return", "label", "addObj", "addLink", "deleteObj", "deleteLink", "setPointer", "setPointerF", "setVar", "setVarF", "setAttr", "DEBUG_ON" and "DEBUG_OFF").

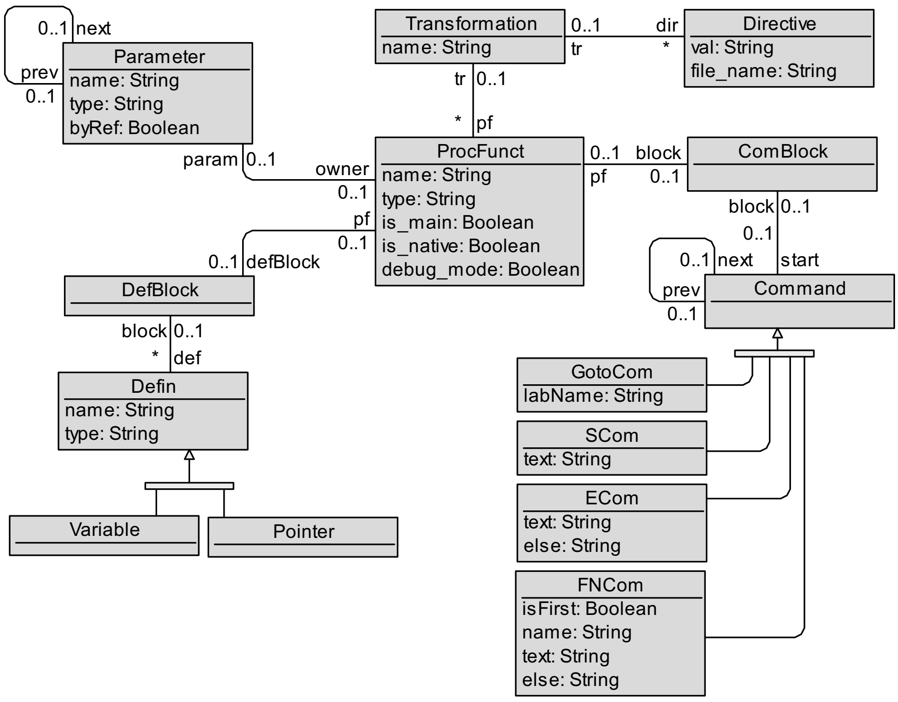


**Fig 1.** The metamodel of the language L0

## 2.3 Model Transformation Example in the Language L0

Let's assume we have given a metamodel consisting of two classes (Fig. 2). Students have name, age, and average marks in each of the eight bachelor's study examination periods. Instances of the class "Course" are courses of master studies and the attribute "hasGoodStudents" shows whether the average mark of all bachelor's examination periods for all adult students of the particular course is at least 8. The attribute "title" of the class "Course" is supposed to be unique. It must be mentioned that the given metamodel is not the best solution for such a fragment of the world, but it is in return very appropriate for the demonstration of the use of languages Lx.

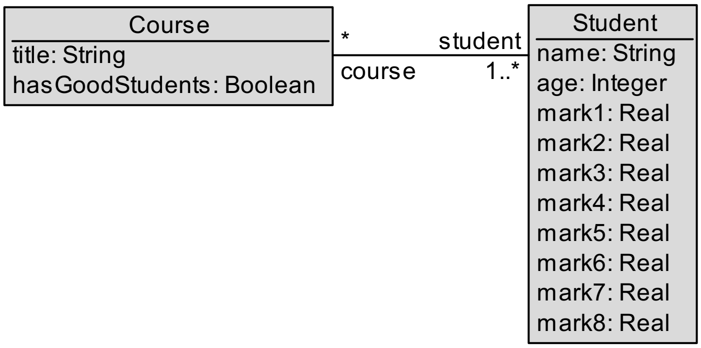


**Fig. 2.** Metamodel used in the example

The problem to solve is as follows – set the correct value of the attribute "hasGoodStudents" for the course named "Operating Systems". The solution written in the language L0 is given below.

```
transformation example;
 main procedure main();
  pointer c:Course;
  pointer s:Student;
  var x:Real;
  var avg:Real;
  var count:Integer;
 begin;
  first c:Course else endOfProg;
  label startFinding;
  attr c.title=="Operating Systems" else getNextCourse;
  goto courseFound;
  label getNextCourse;
  next c else endOfProg;
  goto startFinding;
  label courseFound;
  setVar x=0;
```

```
    setVar count=0;
    first s:Student from c by student
      else noMoreStudents;
    label startCounting;
    attr s.age>=18 else getNextStudent;
    setVar count=count+1;
    setVar avg=s.mark1;
    setVar avg=avg+s.mark2;
    setVar avg=avg+s.mark3;
    setVar avg=avg+s.mark4;
    setVar avg=avg+s.mark5;
    setVar avg=avg+s.mark6;
    setVar avg=avg+s.mark7;
    setVar avg=avg+s.mark8;
    setVar avg=avg/8;
    setVar x=x+avg;
    label getNextStudent;
    next s else noMoreStudents;
    goto startCounting;
    label noMoreStudents;
    var count>0 else writeGood;
    setVar x=x/count;
    var x<8 else writeGood;
    setAttr c.hasGoodStudents=false;
    goto endOfProg;
    label writeGood;
    setAttr c.hasGoodStudents=true;
    label endOfProg;
   end;
  endTransformation;
```

## 3 Model transformation languages L0’ until L3

Transformation languages Lx (or, the so called Lx language family) contain the transformation language L0 and its related transformation languages L0’, L1, L2, and L3. Each of these languages is built based on the previous language of this family by adding some extra features. The syntax and semantics of languages L0’, L1, L2, and L3 are described in this section.

### 3.1 Transformation Language L0’

Model transformation language L0’ (read – „*L0 prim*”) is based on the language L0. The new feature of L0’ is the possibility to make long arithmetic expressions (in L0, only unary and binary expressions were allowed).

Arithmetic expressions of an arbitrary length are allowed in L0’. It means that it is allowed to use each of the four arithmetic operators and traditional brackets (“(” and “)”) when building long expressions. Variables, constants, attributes, and functions can be used as operands in such expressions. The use of operators with respect to the data types is shown in Table 1.

**Table 1.** The use of arithmetic operators with respect to data types

| Operator | Left hand operand | Right hand operand | Result |
|---|---|---|---|
| + | *Integer* | *Integer* | *Integer* |
|  | *Integer* | *Real* | *Real* |
|  | *Real* | *Integer* | *Real* |
|  | *Real* | *Real* | *Real* |
|  | *String* | *String* | *String* |
| - | *Integer* | *Integer* | *Integer* |
|  | *Integer* | *Real* | *Real* |
|  | *Real* | *Integer* | *Real* |
|  | *Real* | *Real* | *Real* |
| * | *Integer* | *Integer* | *Integer* |
|  | *Integer* | *Real* | *Real* |
|  | *Real* | *Integer* | *Real* |
|  | *Real* | *Real* | *Real* |
| / | *Integer* | *Integer* | *Real* |
|  | *Integer* | *Real* | *Real* |
|  | *Real* | *Integer* | *Real* |
|  | *Real* | *Real* | *Real* |

The traditional operator execution sequence is taken into account (from the highest to the lowest):

1) function calls;
2) brackets;
3) multiplication and division;
4) addition and subtraction.

The metamodel of L0’ is made by taking the metamodel of L0 and supplementing it with some new classes and associations. In this metamodel, the class “Expression” together with some other classes is added. Every expression can be attached either to some instance of the class “Ecom” (if it is a comparison) or to some instance of the class “Scom” (if it is an assignment). Every expression contains one starting primitive (instance of the class “Eelem”), and every expression’s primitive has at most one next primitive. Primitives can be of various types – variables, attributes, function calls, constants, operators, and brackets (Fig. 3, bold classes and associations are new in L0’).

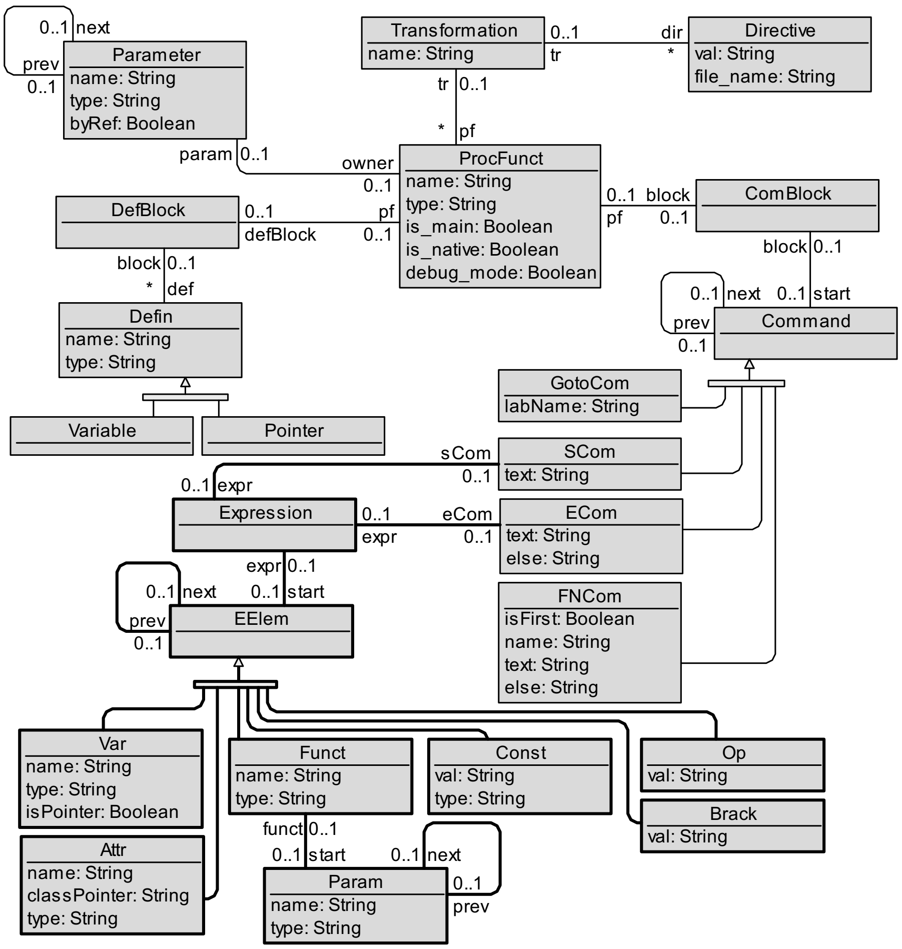


**Fig. 3.** The metamodel of the transformation language L0’

Commands of L0 are the same in L0’. The only difference is in those places where some binary expression could be in the language L0 – now an expression of an arbitrary length is allowed in the language L0’. So it needs to be specified how to write so long expressions. An arithmetic expression can be defined as one of the following:

1) a constant of the type *String* (for example, “17”);
2) a positive constant of the type *Integer* (for example, 17) or *Real* (for example, 17.0);
3) (-C), where C – a positive constant of the type *Integer* or *Real*;
4) a variable of the type *Integer*, *Real* or *String*;
5) an attribute of the type *Integer*, *Real* or *String* that is written in the following way – <pointerName> **.** <attributeName> **:** <typeName>, where <pointerName> is declared as a pointer to the class whose attribute is to inspect;

6) a function call, where the function is of the type *Integer*, *Real* or *String*;
7) (E), where E – an arithmetic expression;
8) E+F, where E and F – arithmetic expressions with compatible types;
9) E-F, where E and F – arithmetic expressions with compatible types;
10) E*F, where E and F – arithmetic expressions with compatible types;
11) E/F, where E and F – arithmetic expressions with compatible types.

For example, correct commands in the language L0' are as follows (if based on the metamodel shown in Fig. 4):

```
1) setVar x=x+y+2;
2) setVar s=z+":"+z+"..."+s1;
3) var x==i*(y+(17/2));
4) attr p.age!=i+person1.age:Integer-1;
5) var y==17.5+3*5/(x+y);
```

It is assumed in those commands that variables and pointer are defined like this:

```
var x:Real;
var y:Real;
var i:Integer;
var s:String;
var s1:String;
var z:String;
pointer p:Person;
pointer person1:Person;
```

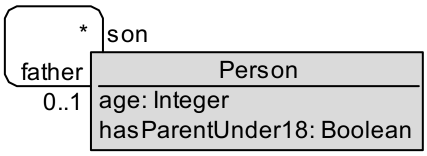


**Fig.4.** The metamodel used in L0' examples

The transformation that solves the problem proposed in Section "2.3. Model transformation example in the language L0" can resemble this in the language L0':

```
transformation example;
 main procedure main();
  pointer c:Course;
  pointer s:Student;
  var x:Real;
  var count:Integer;
 begin;
  first c:Course else endOfProg;
  label startFinding;
  attr c.title=="Operating Systems" else getNextCourse;
  goto courseFound;
  label getNextCourse;
  next c else endOfProg;
  goto startFinding;
```

```
    label courseFound;
    setVar x=0;
    setVar count=0;
    first s:Student from c by student
      else noMoreStudents;
    label startCounting;
    attr s.age>=18 else getNextStudent;
    setVar count=count+1;
    setVar x = x + ( s.mark1:Real + s.mark2:Real +
      s.mark3:Real + s.mark4:Real + s.mark5:Real +
      s.mark6:Real + s.mark7:Real + s.mark8:Real ) / 8;
    label getNextStudent;
    next s else noMoreStudents;
    goto startCounting;
    label noMoreStudents;
    var count>0 else writeGood;
    setVar x=x/count;
    var x<8 else writeGood;
    setAttr c.hasGoodStudents=false;
    goto endOfProg;
    label writeGood;
    setAttr c.hasGoodStudents=true;
    label endOfProg;
   end;
 endTransformation;
```

### 3.2 Transformation Language L1

Transformation language L1 (if compared to L0') is supplemented with a pattern matching facility, so that it is possible to search for some instances satisfying a given pattern. Any L1 pattern can contain conditions put on values of variables or attributes, links between instances and other. Although pattern matching can be considered to be one of the most fundamental modeling concept, the only thing that differs L1 metamodel from the metamodel of the language L0' is one association between classes "FNCom" and "ComBlock" (Fig. 5, the new association is drawn in bold).

So it is now possible to attach the so called "suchthat" block to every instance searching command (these are instances of the class „FNCom"). This block can contain arbitrary L1 commands and thus the pattern can be specified.

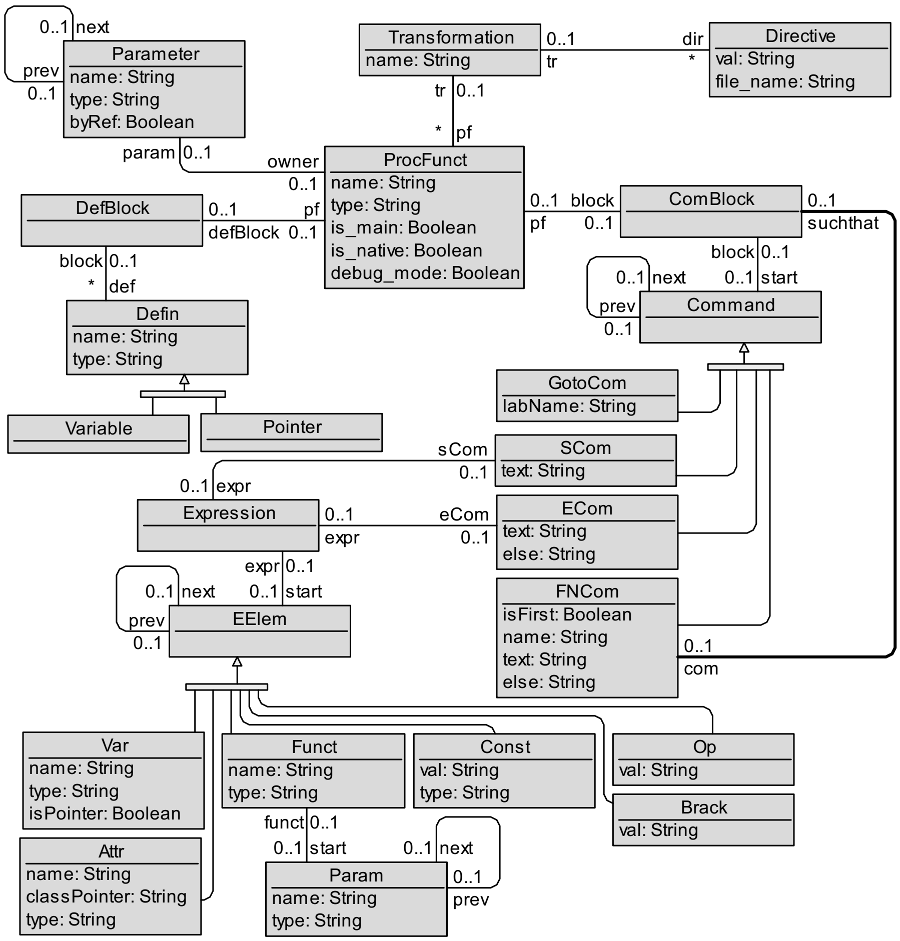


**Fig. 5.** The metamodel of the transformation language L1

In textual syntax, the only difference between languages L0’ and L1 is in commands “first” and “next”. Now it is possible to attach a pattern to them:

```
first <pointerName1> : <className> [ from
 <pointerName2> by <roleName> ] [ suchthat
begin
 <L1Commands>
end ];
next <pointerName> [ suchthat
begin
 <L1Commands>
end ];
```

What is the semantics of the “suchthat” block at all? Commands of this block can always give an answer to the question – does the particular instance satisfy the given pattern or not? Therefore the pattern matching block can be treated like a novel expression of the logical type (*Boolean*) that will further be called the *begin-end* expression [10]. In more formal terms – a *begin-end* expression is any construction built like this:

```
begin
 <L1Commands>
end
```

Now it is possible to define the semantics of a "suchthat" block (or a *begin-end* expression) – a *begin-end* expression is true if, taking into account particular instance and executing all the commands of the given block one by another (starting from the first one), it is possible to successfully reach the end of the block (meaning – successfully execute its last command).

What does it mean in L1 to successfully execute a command? In order to answer this question it will be enough to inspect commands of just two types – "goto" command and commands with a possible "else" branch ("ECom" and "FNCom" instances in the metamodel). In the case of any other L1 command it is assumed that these commands are always successfully executable. Let's take a more detailed view of the two types of commands mentioned above:

1) "goto" commands in the language L0 must be supplemented with exactly one label name (to which label the control must be given after the execution of this "goto" command). In L1, "goto" commands – if used in *begin-end* expressions – must be supplemented with no more than one label. It means the label attached to this command can be empty. If that is the case, the value of the particular *begin-end* expression becomes equal to false when reaching such a "goto" command, and no more commands of this block are to be executed. So the "goto" command is successfully executable if there is exactly one label name attached to it.
2) "ECom" and "FNCom" commands in L0 can contain no more than one "else" branch. If some command contains no "else" branch and it is the case when some comparison of instance searching fails, the control is given to the end of this particular procedure/function. In L1, a non-existing "else" branch in the situation the control would have given to the label specified in this "else" branch leads to the false value of the particular *begin-end* expression that contains this command. So a command that is an instance of the class "ECom" or an instance of the class "FNCom" is successfully executable if it contains either an "else" branch or the comparison, or instance searching does not fail.

Since the semantics of the instance searching commands ("first" and "next") might not be intuitively precisely clear, it is necessary to explain it in detail. In L0, the semantics of these commands are explained in the following manner:

1) When reaching the "first" command with a pointer <pointerName> to the class <className> attached, a possible value set is assigned for this pointer, that is – those instances of the class <className> are distinguished to which it will be further possible for this particular pointer to point. If there is no "from ... by ..." part in this command, the value set contains all the instances of the class <className>, otherwise the value set of <pointerName> is limited to exactly those instances of the class <className> that are reachable from the instance by the role specified in the "from ... by ..." part. After the value set is determined, an arbitrary instance from this set is assigned to <pointerName>

and then withdrawn from the set. If the set is found to be empty, the control is given to the label specified in the “else” branch (if it exists).

2) When reaching the “next” command with a pointer <pointerName> attached (the same pointer that has been attached to some “first” command before), an arbitrary instance of the previously made value set of this pointer is assigned to <pointerName> and then withdrawn from the set. If the set is found empty, the control is given to the label specified in the “else” branch (if it exists).
3) When reaching the “next” command with a pointer attached that is not yet processed in any “first” command (so the value set is not determined for it), program execution semantics is not defined.

In L1, the semantics of instance searching commands is adopted from the language L0, and some conditions according to the semantics of the pattern matching block are added:

1) When reaching the “first” command with a pointer <pointerName> attached, its value set is determined in the same way it was done in the case of the language L0. After that, an arbitrary instance of this value set that satisfies the given *begin-end* expression (if it exists) is assigned to <pointerName> and then withdrawn from the set. If there are no such instances, the control is given to the label specified in the “else” branch (if it exists).
2) When reaching the “next” command with a pointer <pointerName> attached that has previously determined value set (the “first” command on this pointer is executed before), an arbitrary instance of this value set that satisfies the given *begin-end* expression (if it exists) is assigned to <pointerName> and then withdrawn from the set. If there are no such instances, the control is given to the label specified in the “else” branch (if it exists).
3) When reaching the “next” command with such a pointer attached that is not yet processed in any “first” command (so the value set is not determined for it), program execution semantics is not defined.

Let’s consider some examples now. A simple pattern based on which the first instance of the class “Person” is found, where the condition holds that the age of the particular person is 24 (examples used in this section are based on the metamodel shown in Fig. 4):

```
first p:Person suchthat
begin
 p.age==24;
end;
```

In this case, first such p from the class “Person” will be found whom it will be possible to successfully execute this only command – “p.age==24;”. Since it is a command of type “ECom” and it does not contain an “else” branch, the only possible way for this command to be able to execute successfully is the way when the comparison holds. So the *begin-end* expression is true in this case if the value of the attribute “age” of the instance pointed to by p is equal to 24.

To find the next instance of the same class based on the same condition, the “next” command with a pattern matching block needs to be executed:

```
next p suchthat
begin
 attr p.age==24;
```

```
end
else no_more_persons;
```

A pattern based on which the first instance of the class “Person” is found whom a condition holds that it is 24 years old and it is the son of another person pointed to by the pointer *father*:

```
first p:Person suchthat
begin
    attr p.age==24;
 link father.son.p;
end
else no_such_persons;
```

A problem might arise – find the persons that have a 24 year-old son. In this case, the command in L1 that finds the first such person can look like this:

```
first parent:Person suchthat
begin
 first p:Person suchthat
 begin
  link parent.son.p;
  attr p.age==24;
 end;
end
else no_such_persons;
```

If this command executes and the control is not given to the “else” label, the pointer *parent* will point to such instance of the class “Person” that satisfies the condition specified above (moreover – the pointer *p* will point to the instance of the class “Person” that has the link with the given name to the instance pointer to by *parent*). The inner “first” command can be read as “exists”, that is, all the pattern can be read as “Find the first *parent* whom there exists such *p* that is in a relation *son* with the pointer *parent* and that is 24 years old”.

The transformation that solves the problem proposed in Section “2.3. Model transformation example in the language L0” can resemble this in the language L1:

```
transformation example;
 main procedure main();
  pointer c:Course;
  pointer s:Student;
  var x:Real;
  var count:Integer;
 begin;
  first c:Course suchthat
  begin
   attr c.title=="Operating Systems";
  end else endOfProg;
  setVar x=0;
  setVar count=0;
  first s:Student from c by student suchthat
  begin
   attr s.age>=18;
  end else noMoreStudents;
```

```
    label startCounting;
    setVar count=count+1;
    setVar x = x + ( s.mark1:Real + s.mark2:Real +
       s.mark3:Real + s.mark4:Real + s.mark5:Real +
       s.mark6:Real + s.mark7:Real + s.mark8:Real ) / 8;
    next s suchthat
    begin
     attr s.age>=18;
    end else noMoreStudents;
    goto startCounting;
    label noMoreStudents;
    var count>0 else writeGood;
    setVar x=x/count;
    var x<8 else writeGood;
    setAttr c.hasGoodStudents=false;
    goto endOfProg;
    label writeGood;
    setAttr c.hasGoodStudents=true;
    label endOfProg;
   end;
  endTransformation;
```

## 3.3 The Comparison of L1 and a First-Order Logic

How expressive exactly are the pattern definition blocks of the transformation language L1? What are the types of problems solvable by these constructions? This section is devoted to these questions.

Pattern definition block (or to be more precise – the *begin-end* expression attached to it) gives exactly one answer of the logical data type (true or false) for each object of the set under consideration. If looking at the pattern block in such a way, one can start to draw an analogy with formulae of first-order logic that are objects of the logical type as well. While transformation language L1 is known only by a small set of people, first-order logic is considered to be a classic and is ranked as one of the basic disciplines of mathematics. Therefore the comparison of L1 and a first-order logic would give us a better notion of the scope of L1.

Let's consider a many-sorted first-order logic [11]. According to the definition, the alphabet of such a language consists of seven sets of symbols:

1) a countable set $S \cup \{bool\}$ of sorts (or types) containing the special sort *bool* such that S is non-empty and does not contain *bool*;
2) logical connectives: $\wedge$ (conjunction), $\vee$ (disjunction), $\supset$ (implication) and $\equiv$ (equivalence) that are all of rank ($bool \times bool \rightarrow bool$), $\neg$ (negation) of rank ($bool \rightarrow bool$) and $\perp$ (a bottom concept) of rank ($\varnothing \rightarrow bool$);
3) quantifiers: $\forall_s$ (universal quantifier) and $\exists_s$ (existential quantifier) for every set $s \in S$;
4) an equality symbol: $=_s$ of rank ($s \times s \rightarrow bool$) for every set $s \in S$;

5) variables: a countably infinite set $V_s = \{x_0, x_1, x_2, ...\}$ for every set $s \in S$ each variable $x_i$ being of rank $(\varnothing \rightarrow s)$;
6) auxiliary symbols: "(" and ")";
7) an alphabet L of non-logical symbols consisting of:
   a. function symbols: a countable set $FS = \{f_0, f_1, ...\}$ and a rank function $r: FS \rightarrow S^+ \times S$ ($S^+$ contains all the words of S excepting the empty word, that is, all the strings of length $n>0$ whose all elements belong to the set S), assigning a pair $r(f) = (u,s)$ called rank to every function symbol f; the string u is called the arity of f, and the symbol $s \in S$ – the sort (or type) of f;
   b. constants: a countable set $CS_s = \{c_0, c_1, ...\}$ for every set $s \in S$ each $c_i$ being of rank $(\varnothing \rightarrow s)$;
   c. predicate symbols: a countable set $PS = \{P_0, P_1, ...\}$and a rank function $r: PS \rightarrow S^* \times \{bool\}$ ($S^*$ contains all the words of S including the empty word) assigning a pair $r(P) = (u, bool)$ to each predicate symbol P; the string u is called the arity of P.

It is assumed that all the sets $V_s$, FS, $CS_s$ and PS is mutually disjoint for every possible value of $s \in S$.

Taking into account such a definition, terms and atomic formulae in the first-order logic are defined as follows:

1) every constant and every variable of sort s is a term of sort s;
2) if $t_1, ..., t_n$ are terms, each $t_i$ of sort $u_i$, and f is a function symbol of rank $(<u_1, ..., u_n> \rightarrow s)$, then $f(t_1, ..., t_n)$ is a term of sort s;
3) every predicate symbol of arity $\varnothing$, as well as the bottom concept ($\perp$) is an atomic formula;
4) if $t_1$ and $t_2$ are terms of sort s, then $=_s(t_1, t_2)$ is an atomic formula;
5) if $t_1, ..., t_n$ are terms, each $t_i$ of sort $u_i$, and P is a predicate symbol of arity $u_1, ..., u_n$, then $P(t_1, ..., t_n)$ is an atomic formula.

Formulae are defined as follows:

1) every atomic formula is a formula;
2) for any two formulae A and B, $(A \wedge B)$, $(A \vee B)$, $(A \supset B)$, $(A \equiv B)$ and $\neg A$ are also formulae;
3) for any variable x of sort s and any formula A, $\forall_s x(A)$ and $\exists_s x(A)$ are also formulae.

Let's look now at a subset of full many-sorted first-order logic called the language $P^-$, that contains only binary predicate symbols and functions with only one argument. In that case, the alphabet of the language $P^-$ can be defined in the following manner:

1) a countable set $S \cup \{bool\}$ of sorts (or types) containing the special sort *bool* such that S is non-empty and does not contain *bool*;
2) logical connectives: $\wedge$ (conjunction) and $\vee$ (disjunction) of rank $(bool \times bool \rightarrow bool)$, $\neg$ (negation) of rank $(bool \rightarrow bool)$ and $\perp$ (a bottom concept) of rank $(\varnothing \rightarrow bool)$;
3) quantifiers: $\forall_s$ (universal quantifier) and $\exists_s$ (existential quantifier) for every set $s \in S$;
4) an equality symbol: $=_s$ of rank $(s \times s \rightarrow bool)$ for every set $s \in S$;
5) variables: a countably infinite set $V_s = \{x_0, x_1, x_2, ...\}$ for every set $s \in S$ each variable $x_i$ being of rank $(\varnothing \rightarrow s)$;

6) auxiliary symbols: “(” and “)”;
7) an alphabet L of non-logical symbols consisting of:
   a. function symbols: a countable set FS = $\{f_0, f_1, ...\}$ and a rank function r: FS $\rightarrow$ S $\times$ S, assigning a pair r(f) = (s,s) to every function symbol f;
   b. constants: a countable set $CS_s = \{c_0, c_1, ...\}$ for every set $s \in S$ each $c_i$ being of rank ($\varnothing \rightarrow$ s);
   c. predicate symbols: a countable set PS = $\{P_0, P_1, ...\}$and a rank function r: PS $\rightarrow S^2 \times \{bool\}$, assigning a pair r(P) = (<s,s>, *bool*) to each predicate symbol P.

So terms and atomic formulae in $P^-$ can be defined as follows:
1) every constant and every variable of sort s is a term of sort s;
2) if t is a term of sort u, and f is a function symbol of rank (u $\rightarrow$ s), then f(t) is a term of sort s;
3) $\perp$ is an atomic formula;
4) if $t_1$ and $t_2$ are terms of sort s, then $=_s(t_1, t_2)$ is an atomic formula;
5) if $t_1$ and $t_2$ are terms, each $t_i$ of sort $u_i$, and P is a predicate symbol of arity $<u_1, u_2>$, then $P(t_1, t_2)$ is an atomic formula.

Formulae in $P^-$ are defined as follows:
1) every atomic formula is a formula;
2) for any two formulae A and B, $(A \wedge B)$, $(A \vee B)$ and $\neg A$ are also formulae;
3) for any variable x of sort s and any formula A, $\forall_s x(A)$ and $\exists_s x(A)$ are also formulae.

Now it is possible to see some similarities between languages $P^-$ and L1. Although different terms are used to define these two languages, it is possible to establish some links between them (see Table 2).

**Table 2.** Linking concepts of languages $P^-$ and L1

| **Concept of $P^-$** | **Concept of L1** |
|---|---|
| The set of sorts S | The set C $\cup$ {*Integer*, *Real*, *String*, *Boolean*} where C – the set of all classes found in the metamodel used |
| The bottom concept $\perp$ | *Boolean* value *false* |
| Other logical connectives | Will be interpreted in the context |
| Existential quantifier $\exists_s$ where $s \in C$, and C – the set of all classes found in the metamodel used | The command “first” |
| Universal quantifier $\forall_s$ where $s \in C$, and C – the set of all classes found in the metamodel used | Will be interpreted by transforming the expression containing the universal quantifier into the form of that containing an existential quantifier |
| The equality symbol $=_s$ where $s \in$ {*Integer*, *Real*, *String*, *Boolean*} | The command “var” |
| The equality symbol $=_s$ where $s \in C$, and C – the set of all classes found in the metamodel used | The command “pointer” |

| | |
|---|---|
| The set of variables $V_s$ where $s\in\{$*Integer*, *Real*, *String*, *Boolean*$\}$ | Variables of primitive data types, declared by the keyword "var" |
| The set of variables $V_s$ where $s\in C$, and C – the set of all classes found in the metamodel used | Pointers to instances, declared by the keyword "pointer" |
| Auxiliary symbols "(" and ")" | Will be interpreted in the context |
| The function symbol with one argument | The attribute of a class |
| Constants of types | Constants of primitive data types |
| Binary predicate symbols $P(t_1, t_2)$ where $t_1, t_2\in C$, and C – the set of all classes found in the metamodel used | The command "link" |

Theorem. For each formula of the predicate language $P^-$, there exists a *begin-end* expression in the language L1 of the same truth value.

Proof. A constructive proof is provided for this theorem. For the theorem to be proven it is sufficient to produce a valid *begin-end* expression for each type of formulae of $P^-$ shown in Table 2. To do this, two auxiliary formulae need to be introduced:

1) expr: <$P^-$ formula> → <L1 *begin-end* expression> – a function assigning an L1 *begin-end* expression to the given $P^-$ formula;
2) insert: <L1 *begin-end* expression> × <*String*> → <L1 *begin-end* expression> – a function calculating a new *begin-end* expression from the existing one by adding the given label name (second parameter) to missing places of the initial expression (to "goto" commands without a label and to non-existing "else" branches of those commands that can contain an "else" branch).

All types of $P^-$ formulae and their respective L1 *begin-end* expressions are shown in Table 3. (labels "unicalLabel", "unicalLabelForA", and "endLabel", as well as pointers "unicalPtrName1" and "unicalPtrName2", and variables "unicalVarName1" and "unicalVarName2" are considered to be unique in the whole given procedure/function).

**Table 3.** Construction of an L1 code from $P^-$ formulae

| **F** | **expr(F)** |
|---|---|
| $\perp$ | **goto**; |
| $=_s(t_1,t_2)$ where $s\in\{$*Integer*, *Real*, *String*, *Boolean*$\}$ | **setVar** unicalVarName1=$t_1$;<br>**setVar** unicalVarName2=$t_2$;<br>**var** unicalVarName1==unicalVarName2; |
| $=_s(t_1,t_2)$ where $s\in C$, and C – the set of all classes found in the metamodel used | **setPointer** unicalPtrName1=$t_1$;<br>**setPointer** unicalPtrName2=$t_2$;<br>**pointer** unicalPtrName2==unicalPtrName2; |
| $P(t_1, t_2)$ | **setPointer** unicalPtrName1=$t_1$;<br>**setPointer** unicalPtrName2=$t_2$;<br>**link** unicalPtrName1.P.unicalPtrName2; |

| | |
|---|---|
| $A \wedge B$ | expr(A)<br>expr(B) |
| $A \vee B$ | **insert**(expr(A),"unicalLabel")<br>**goto** endLabel;<br>**label** unicalLabel;<br>expr(B)<br>**label** endLabel; |
| $\neg A$ | **insert**(expr(A),"unicalLabel")<br>**goto**;<br>**label** unicalLabel; |
| $\exists_s x(A)$ | **first** x:S **suchthat**<br>**begin**<br>  expr(A)<br>**end**; |
| $\forall_s x(A) \equiv \neg\exists_s x(\neg A)$ | **first** x:S **suchthat**<br>**begin**<br>  insert(expr(A),"unicalLabelForA")<br>  **goto**;<br>  **label** unicalLabelForA;<br>**end else** unicalLabel;<br>**goto**;<br>**label** unicalLabel; |

It is worth mentioning that it is easier to use the form of an existential quantifier and to produce a *begin-end* expression based on that in the case of a universal quantifier.

In order to get a clearer understanding of the functions used to construct the L1 code, examples of all the different cases are given in Table 4.

**Table 4.** Construction of an L1 code from $P^-$ formulae – examples

| F | expr(F) |
|---|---|
| $\perp$ | **goto**; |
| $=_{Integer}(x,17)$ | **setVar** unicalVarName1=x;<br>**setVar** unicalVarName2=17;<br>**var** unicalVarName1==unicalVarName2; |
| $=_{Person}(p,q)$ | **setPointer** unicalPtrName1=p;<br>**setPointer** unicalPtrName2=q;<br>**pointer** unicalPtrName1== unicalPtrName2; |
| father(p,q) | **setPointer** unicalPtrName1=p;<br>**setPointer** unicalPtrName2=q;<br>**link** unicalPtrName1.father.unicalPtrName2; |
| $(father(p,q) \wedge =_{Integer}(age(p),18))$ | **setPointer** unicalPtrName1=p;<br>**setPointer** unicalPtrName2=q;<br>**link** unicalPtrName1.father.unicalPtrName2;<br>**setPointer** unicalPtrName3=p;<br>**attr** unicalPtrName3.age==18; |

| | |
|---|---|
| $((\mathrm{father}(p,q) \wedge =_{\mathrm{Integer}}(\mathrm{age}(p), 18)) \vee =_{\mathrm{Integer}}(\mathrm{age}(q),18))$ | **setPointer** unicalPtrName1=p;<br>**setPointer** unicalPtrName2=q;<br>**link** unicalPtrName1.father.unicalPtrName2 **else** unicalLabel;<br>**setPointer** unicalPtrName3=p;<br>**attr** unicalPtrName3.age==18 **else** unicalLabel;<br>**goto** endLabel;<br>**label** unicalLabel;<br>**setPointer** unicalPtrName4=q;<br>**attr** unicalPtrName4.age==18;<br>**label** endLabel; |
| $\neg=_{\mathrm{Integer}}(\mathrm{age}(p),18)$ | **setPointer** unicalPtrName1=p;<br>**attr** unicalPtrName1.age==18 **else** unicalLabel;<br>**goto**;<br>**label** unicalLabel; |
| $\exists_{\mathrm{Person}} p\ (=_{\mathrm{Integer}}(\mathrm{age}(p),18))$ | **first** p:Person **suchthat**<br>**begin**<br>  **setPointer** unicalPtrName1=p;<br>  **attr** unicalPtrName1.age==18;<br>**end**; |
| $\forall_{\mathrm{Person}} p\ (=_{\mathrm{Integer}}(\mathrm{age}(p),18)) \equiv \neg\exists_{\mathrm{Person}} p\ (\neg=_{\mathrm{Integer}}(\mathrm{age}(p),18))$ | **first** p:Person **suchthat**<br>**begin**<br>  **setPointer** unicalPtrName1=p;<br>  **attr** unicalPtrName1.age==18 **else** unicalLabelForA;<br>  **goto**;<br>  **label** unicalLabelForA;<br>**end else** unicalLabel;<br>**goto**;<br>**label** unicalLabel; |

Although the construction of *begin-end* expressions is inductive in most cases, it is easy to see that it is indeed possible to construct a *begin-end* expression with the same truth value as that of the given $P^{-}$ formula in each case. End of proof.

Actually, *begin-end* expressions are even more powerful than the predicate language mentioned above. This is so mainly because of three reasons [10]:

1) it is possible to operate with variables of primitive types in *begin-end* expressions;
2) a *begin-end* expression specifies the command execution order during the pattern matching (i.e., the order in which instances are traversed);
3) when a pattern is matched, all its elements are assigned an identity which can be used further for referencing these elements.

## 3.4 Transformation Language L2

The new feature of the language L2 if compared to the language L1 is the possibility to make loops. A special command exists in L2 with which it is possible to visit either all instances of the specified class or just those instances of the class that match the given pattern.

In the metamodel of L2, one class is added ("ForeachCom") if compared to the metamodel of L1 (Fig. 6, bold class and associations are new in L2). Two associations from this class to the class "ComBlock" exist – one for the commands of the loop and the other for the pattern definition block of the loop.

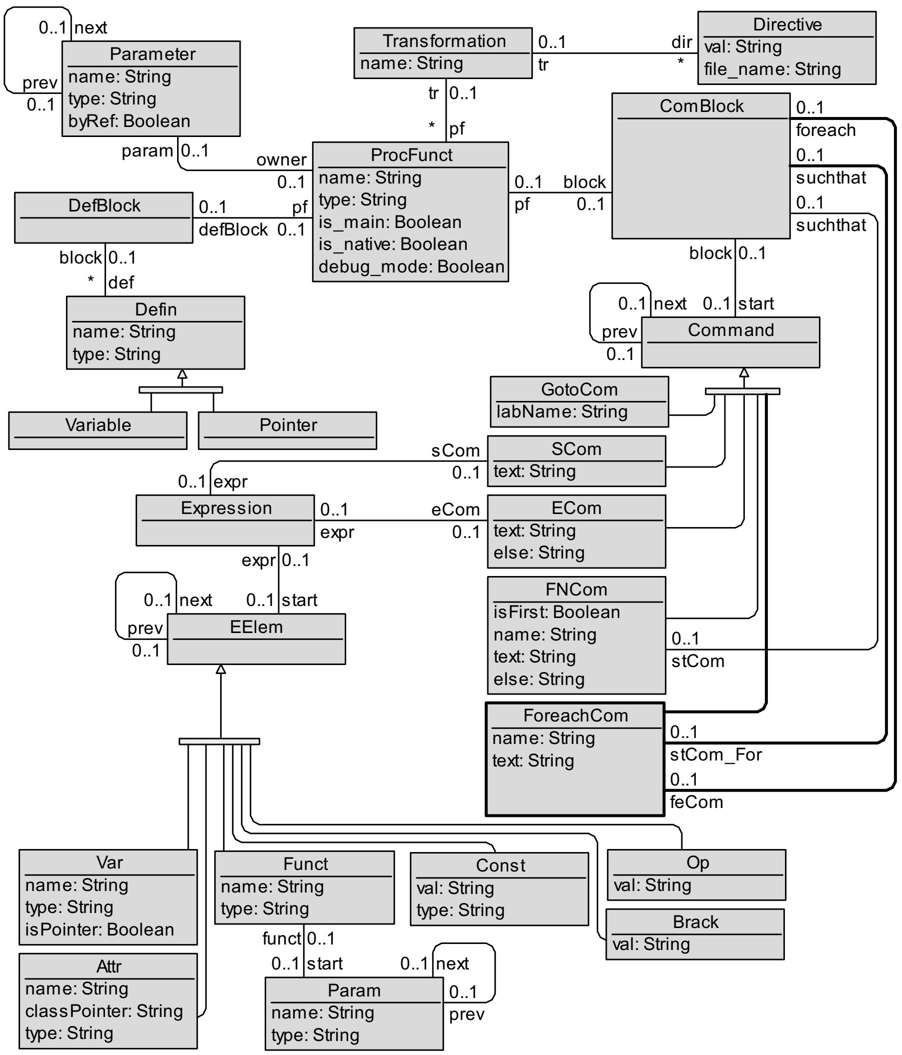


**Fig. 6.** The metamodel of the transformation language L2

The textual syntax for a loop command is as follows:

```
foreach <pointerName1> : <className> [ from
  <pointerName2> by <roleName> ] [ suchthat
begin
    <L2Commands>
end ]
do
begin
    <L2Commands>
end;
```

The semantics of the "suchthat" block is the same as in the case of the language L1. Since this block is optional, the semantics of the "foreach" command is as follows – every instance of the specified class that matches the given pattern (if such exists; otherwise it is considered that every instance is to be taken) is traversed and all the commands of the "do" block are executed for it.

Let's consider some examples. Increase the value of the attribute "age" of all instances of the class "Person" by 1 (all examples in this section are based on the metamodel seen in Fig. 4):

```
foreach p:Person do
begin
 setAttr p.age=p.age+1;
end;
```

Increase the age of all persons younger than 18 by 1:

```
foreach p:Person suchthat
begin
 attr p.age<18;
end
do
begin
 setAttr p.age=p.age+1;
end;
```

Nested loop example – set the value of the attribute "hasParentUnder18" to *true* for those persons that are sons of a person younger than 18:

```
foreach parent:Person suchthat
begin
 attr parent.age<18;
end
do
begin
 foreach p:Person suchthat
 begin
  link parent.son.p;
 end
 do
 begin
  setAttr p.hasParentUnder18=true;
 end;
end;
```

The transformation that solves the problem proposed in Section "2.3. Model transformation example in the language L0" can resemble this in the language L2:

```
transformation example;
 main procedure main();
  pointer c:Course;
  pointer s:Student;
  var x:Real;
  var count:Integer;
 begin;
  first c:Course suchthat
  begin
   attr c.title=="Operating Systems";
  end else endOfProg;
  setVar x=0;
  setVar count=0;
  foreach s:Student from c by student suchthat
  begin
   attr s.age>=18;
  end
  do
  begin
   setVar count=count+1;
   setVar x = x + ( s.mark1:Real + s.mark2:Real +
     s.mark3:Real + s.mark4:Real + s.mark5:Real +
     s.mark6:Real + s.mark7:Real + s.mark8:Real ) / 8;
  end;
  var count>0 else writeGood;
  setVar x=x/count;
  var x<8 else writeGood;
  setAttr c.hasGoodStudents=false;
  goto endOfProg;
  label writeGood;
  setAttr c.hasGoodStudents=true;
  label endOfProg;
 end;
endTransformation;
```

## 3.5 Transformation Language L3

The new feature of the language L3 if compared to the language L2 is the branching command – a standard "if-then-else" construction than can be used instead of constructions made using "goto" commands in some cases.

A new class is added in the metamodel of L3 if compared to the metamodel of L2 – "IfCom" (Fig. 7, bold class and associations are new in L0'). Three associations from this command to the class "ComBlock" exist – one for the "if" clause, one for the "then" clause, and one for the "else" clause of the command.

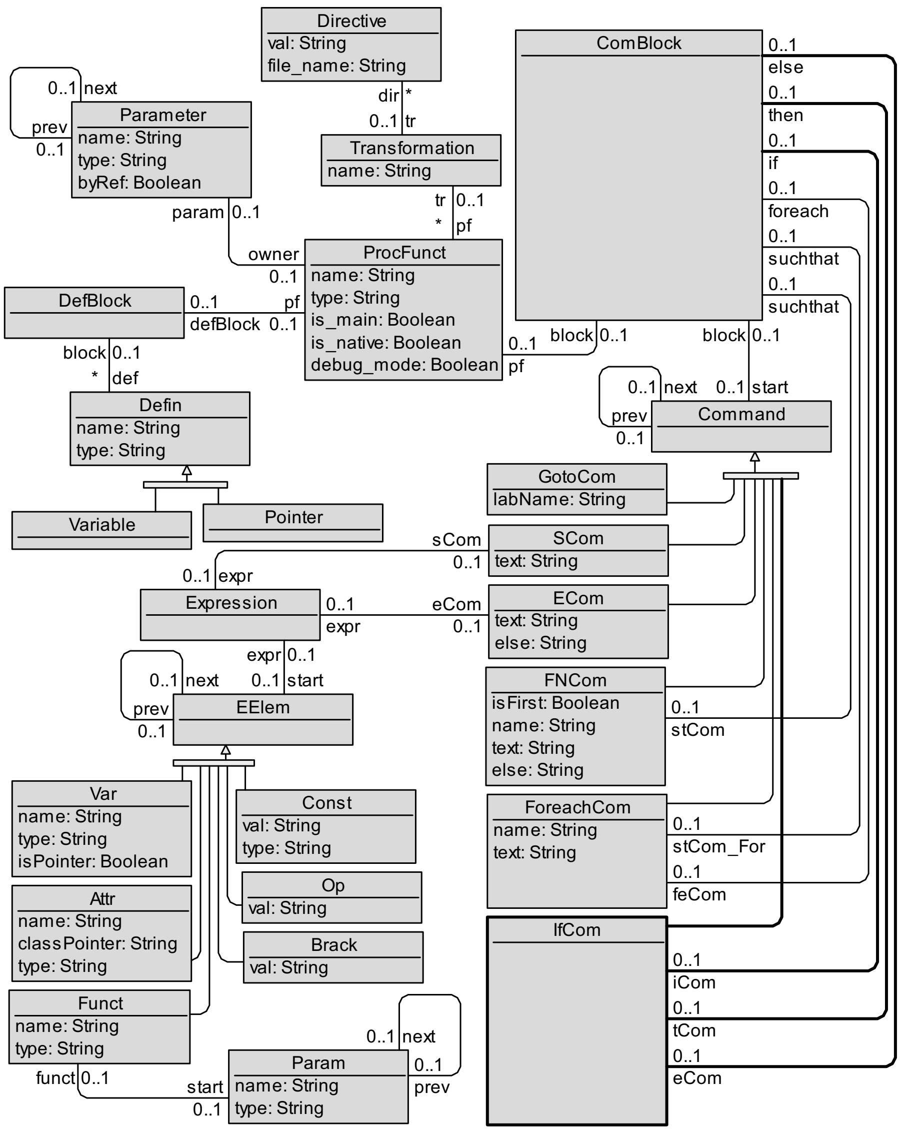


**Fig. 7.** The metamodel of the transformation language L3

The situation with "then" and "else" blocks is intuitively quite clear – these blocks must contain commands to be executed in the case of respectively true and false value of some condition. But what about the "if" block? This is again the case of *begin-end* expressions – an expression is attached to the "if" clause of an "IfCom" command, and so the condition of the "IfCom" command is true if the respective *begin-end* expression is true.

The textual syntax of the branching command is as follows:

```
if
begin
     <L3Commands>
end
then
begin
     <L3Commands>
end
[ else
begin
     <L3Commands>
end ];
```

Since the "else" part is optional, it is possible that no commands are to be executed in the case of false value of the condition.

Let's consider some examples. Let's assume we have a pointer p pointing to some instance of the class "Person". Increase the value of the attribute 'age" of this instance by 1 if it is less than 18 (all examples in this section are based on the metamodel shown in Fig. 4):

```
if
begin
 attr p.age<18;
end
then
begin
 setAttr p.age=p.age+1;
end;
```

Increase the age of the person pointed to by p by 1 if it is less than 18, otherwise decrease it by 1:

```
if
begin
 attr p.age<18;
end
then
begin
 setAttr p.age=p.age+1;
end
else
begin
 setAttr p.age=p.age-1;
end;
```

A more complicated example – assign a value "Less than hundred" or "Hundred or more" to a *String* variable *s* based on the fact whether the total age of all persons younger than 18 is less than 100 or not:

```
if
begin
 setVar sum=0;
 foreach p:Person suchthat
 begin
```

```
    attr p.age<18;
   end
   do
   begin
    setVar sum=sum+p.age;
   end;
   var sum<100;
  end
  then
  begin
   setVar s="Less than hundred";
  end
  else
  begin
   setVar s="Hundred or more";
  end;
```

In this example, the value of the variable *sum* could also be calculated before the branching command, however, it can easily be done in the same command when the semantics of the variable tells it is only needed in the "IfCom" command.

The transformation that solves the problem proposed in Section "2.3. Model transformation example in the language L0" can resemble this in the language L3:

```
transformation example;
 main procedure main();
  pointer c:Course;
  pointer s:Student;
  var x:Real;
  var count:Integer;
 begin;
  first c:Course suchthat
  begin
   attr c.title=="Operating Systems";
  end else endOfProg;
  setVar x=0;
  setVar count=0;
  foreach s:Student from c by student suchthat
  begin
   attr s.age>=18;
  end
  do
  begin
   setVar count=count+1;
   setVar x = x + ( s.mark1:Real + s.mark2:Real +
     s.mark3:Real + s.mark4:Real + s.mark5:Real +
     s.mark6:Real + s.mark7:Real + s.mark8:Real ) / 8;
  end;
  if
  begin
   var count>0;
```

```
      setVar x=x/count;
      var x<8;
     end
     then
     begin
      setAttr c.hasGoodStudents=false;
     end
     else
     begin
      setAttr c.hasGoodStudents=true;
     end;
     label endOfProg;
    end;
  endTransformation;
```

A full syntax definition of the transformation language L3 based on *Backus-Naur* notation [12] is given in Appendix A.

## 4 The implementation of model transformation languages L0', L1, L2, and L3

### 4.1 The Main Principles of the Implementation of L0' Until L3

As mentioned in the first sections of this paper, there already exists an effective implementation of the language L0, that is, a compiler to the language C++. Since languages L0', L1, L2, and L3 have been built based on the language L0, a very logical step would be the implementation of these languages through the language L0. So actually the basic idea of the implementation of languages L0' until L3, is to build a compiler for each of the languages L0', L1, L2, and L3 to L0. However, for the task to be accomplished more easily, each compiler will be built to the language that is a direct ancestor to it instead of building each compiler to the language L0. Since every next language in the Lx family was built based on the previous one by just adding some new features, such an implementation is possible. The algorithm used to implement these compilers is called the bootstrapping algorithm [13]. The bootstrapping principle is to build a compiler of one language to the other language that is written in the target language. So, translating this into the language of Lx – to write a compiler in L0 that transforms an L0' program into an L0 program, then to write a compiler in L0' that transforms an L1 program into an L0' program, then to write a compiler in L1 that transforms an L2 program into an L1 program, and finally to write a compiler in L2 that transforms an L3 program into an L2 program. When turning to the details of the actual implementation, it is worth mentioning that each compiler can actually be written in L0 (because L0 is a subset of every other Lx language). In doing so, the idea of bootstrapping algorithm is not violated – it can still be considered that each compiler is written in the target language perhaps just without using all features the language offers.

In contrast with the ideas of the traditional compiler building [14] in which an analysis of textual forms of programs is taken for a base, the idea of the bootstrapping algorithm is very convenient in the case of model transformation languages – any program in a model transformation language is considered as a particular model in the metamodel of that particular language. If both metamodels of the source and target languages are known, as well as the particular source model corresponding to the program to be compiled is known, the compilation task transforms into a standard model transformation task.

When consideringthe possibility to implement the bootstrapping algorithm, it must be verified whether it is really possible for each program of the source language to make an equivalent program in the target language (this could not be possible due to new features of the target language). Therefore it must be verified for the case of languages Lx immediately.

The main question about the compiler from L0' to L0 is whether it is possible for every arithmetic expression of L0' to build an equivalent construction in L0. It is easy to see that every such expression can be divided into some binary or unary expressions that are allowed in L0 (by assigning intermediate results to temporary variables). Consequently it is really possible to build a compiler from L0' to L0.

In L1, a pattern matching is possible. The question that arises – is each L1 pattern block translatable into L0'? The answer is – yes. Every pattern block can be simulated with L0' commands (see Table 5). The general idea is to make some extra labels and to add a label to commands without any labels to intercept those control flows that correspond to the situation in L1 the pattern matching fails.

**Table 5.** The principle of the implementation of a pattern definition block

| L1 | L0' |
|---|---|
| **first** <ptrName1>**:**<className> | **first** <ptrName1>**:**<className> [**from** |
| [**from** <ptrName2> **by** <roleName>] | <ptrName2> **by** <roleName>] [**else** |
| **suchthat** | <labelName>]; |
| **begin** | **label** ___L_i; |
| <command_1>; | <command_1> [**else** ___L_i+1]; |
| <command_2>; | <command_2> [**else** ___L_i+1]; |
| ... | ... |
| <command_n>; | <command_n> [**else** ___L_i+1]; |
| **end** | **goto** ___L_i+2; |
| [**else** <labelName>]; | **label** ___L_i+1; |
|  | **next** <ptrName1> [**else** <labelName>]; |
|  | **goto** ___L_i; |
|  | **label** ___L_i+2; |
| **next** <ptrName> **suchthat** | **next** <ptrName> [**else** <labelName>]; |
| **begin** | **label** ___L_i; |
| <command_1>; | <command_1> [**else** ___L_i+1]; |
| <command_2>; | <command_2> [**else** ___L_i+1]; |
| ... | ... |
| <command_n>; | <command_n> [**else** ___L_i+1]; |
| **end** | **goto** ___L_i+2; |
| [**else** <labelName>]; | **label** ___L_i+1; |

| | |
|---|---|
| | **next** <ptrName> [**else** <labelName>];<br>**goto** ___L_i;<br>**label** ___L_i+2; |

Of course, "else" branches are attached to only those commands which can (but do not) have an "else" branch. A label in the form "___L_i" is considered to be a new label that cannot be found in any L0' program written by the user. If a command of the pattern definition block proves to be an instance searching command with a "suchthat" block, it needs to be expanded to L0' commands recursively.

The question about the language L2 – is every "foreach" loop writable in L1? Again, the answer is implicitly clear – it is! Since there are commands "goto" and "label" in L1, as well as "else" branches, the control flow can be moved around to one's liking, inter alia, making a loop over either all instances of some particular class, or just those instances that match the specified pattern (pattern matching constructions are present in L1, so they do not need to be transformed in any way). The scheme of the implementation of loops is shown in Table 6.

**Table 6.** The principle of the implementation of a "foreach" loop

| L2 | L1 |
|---|---|
| **foreach** <ptrName> **:** <className><br>[**suchthat**<br>**begin**<br>  <command_1>;<br>  <command_2>;<br>  ...<br>  <command_n>;<br>**end**]<br>**do**<br>**begin**<br>  <do_command_1>;<br>  <do_command_2>;<br>  ...<br>  <do_command_k>;<br>**end**; | **first** <ptrName> **:** <className><br>[**suchthat**<br>**begin**<br><command_1>;<br><command_2>;<br>...<br><command_n>;<br>**end**]<br>**else** __L_i;<br>**label** __L_i+1;<br><do_command_1>;<br><do_command_2>;<br>...<br><do_command_k>;<br>**next** <ptrName> [**suchthat**<br>**begin**<br><command_1>;<br><command_2>;<br>...<br><command_n>;<br>**end**]<br>**else** __L_i;<br>**goto** __L_i+1;<br>**label** __L_i; |

Again, labels in form "__L_i" are not supposed to be found in any L1 program written by the user.

The question about the language L3 – is every branching construction writable in L2? The answer is even more obvious in this case than in the previous ones – every branching construction can be simulated using "goto" and "label" commands in a standard way (Table 7). As shown above, "else" branches are added here to intercept the cases when the "if" condition (*begin-end* expression) fails.

**Table 7.** The principle of the implementation of a branching command

| L3 | L2 |
|---|---|
| **if** | <if_command_1> [**else** _L_i]; |
| **begin** | <if_command_2> [**else** _L_i]; |
|   <if_command_1>; | ... |
|   <if_command_2>; | <if_command_n> [**else** _L_i]; |
|   ... | <then_command_1>; |
|   <if_command_n>; | <then_command_2>; |
| **end** | ... |
| **then** | <then_command_k>; |
| **begin** | **goto** _L_i+1; |
|   <then_command_1>; | **label** _L_i; |
|   <then_command_2>; | [<else_command_1>; |
|   ... | <else_command_2>; |
|   <then_command_k>; | ... |
| **end** | <else_command_l>;] |
| [**else** | **label** _L_i+1; |
| **begin** | |
|   <else_command_1>; | |
|   <else_command_2>; | |
|   ... | |
|   <else_command_l>; | |
| **end**]; | |

Again, labels in form "_L_i" are not supposed to be found in any L2 program written by user.

## 4.2 Main Problems of the Implementation of L0' Until L3

When compiling a program from one language to another, new "label" commands may appear that must be unique in the scope of the whole particular procedure/function. Therefore, some limitations of naming conventions must exist. In the case of languages L0' until L3 these limitations are as follows – the user cannot make labels in L1 starting with three underscores ("_"), in L2 – starting with two underscores, and in L3 – starting with at least one underscore. If so, compilers of L1, L2, and L3 can make new labels starting with some underscores (so many that the user is allowed to make such a label in the target language, but is not allowed in the source language) and following by the symbol "L" and an integer uniquely generated for every new label. In the case of the L0' compiler, no labels are made, so no action needs to be performed.

In the case of L0’ compilation, some new variables may appear that must be unique in the scope of the whole particular procedure/function. Therefore, some limitations of naming conventions must exist. The limitation offered is similar to the case of labels – to forbid declaring variables starting with an underscore in languages L0’, L1, L2, and L3. So the compiler of L0’ will make variables starting with an underscore and followed by a string “var” and an integer.

When compiling an L2 program to an L1 program, two instance searching commands (“first” and “next”) arise from each loop command. If the loop command contains a pattern definition block, both instance searching commands will contain this block as well. In terms of models, this means that there is one command block assigned to two “FNCom” commands. When passing to the next step – the compilation of L1 – a problem arises – a compiler processes the first of two instance searching commands mentioned above and then deletes the pattern definition block from the metamodel (in order to make the model respective to L0’ program). So the information about the fact that the other command had this block as well is lost. Hence to solve this problem, it is forbidden to have one command block attached to more than one command. Therefore, in the compilation of L2, a copy of the command block must be made. It means that a copy of each command of the particular block must be made. There is no problem in regard to other commands, but a problem arises when “label” commands come in place – how to preserve the uniqueness of labels? The solution here is to make a new unique label from the old unique label by simply concatenating the label name with itself. For example, if the old label name was “__L17”, the new one is “__L17__L17”; if the old one was “myLabel”, the new one is “myLabelmyLabel”. The same conventions relate to “goto” commands and “else” branches as well.

### 4.3 The Whole Compilation Process – From L3 Up To L0

Before turning to the compilation process, one more concept needs to be explained, and that concept is the lexem of the transformation language L3. The metamodel of lexems – basic syntactical elements of a program – is very simple (Fig. 8). Lexems are one of the intermediate steps in the full compilation process from L3 up to L0.

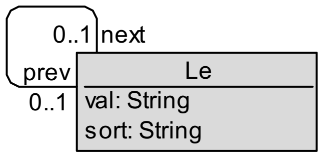


**Fig. 8.** The metamodel of lexems

The full compilation process transforming one text file to another consists of the following components:

1) initialization tasks – the deletion of any old models of metamodels of L3 and lexems, and the generation of the new model of lexems from the input text file specified by the user;

2) transformation of the model of lexems into an L3 model;
3) transformation of the L3 model into an L2 model;
4) transformation of the L2 model into an L1 model;
5) transformation of the L1 model into an L0' model;
6) transformation of the L0' model into an L0 model;
7) printing of the L0 model to an output text file specified by the name of the transformation.

Owing to the fact that pre- and post-conditions of each component are precisely clear, it is possible to easily use just a subset of all components instead of using the full compiler as well. It can be useful, for example, in cases when L3 is used as an intermediate language in the compilation of some higher-level language as it is done in the compilation of the graphical model transformation language MOLA [15] – there is no need to perform neither initialization tasks nor the "lexems to L3" transformation because the L3 model has got into the metamodel of L3 in some other way.

The full L3 to L0 compiler is available in the Lx homepage [16].

## 5 Conclusions and future work

It has been evidenced that the bootstrapping method justifies itself in the use of compiler building if operating with model transformation languages. With this method, it is possible to build higher and higher-level model transformation languages that are easy to compile to some lower-level language. The sequence of such languages – the Lx language family – has been stopped at the language L3 which is of sufficiently high level to be used in practical model transformation tasks. As a proof of this, a transformation-based graphical tool-building platform GrTP is being developed using the language L3 as its base language [17]. The other use case of the language L3 is the implementation of even higher-level languages. The graphical model transformation language MOLA [6,7] has been implemented by bootstrapping method, using the language L3 as an intermediate language in this process [15].

The further work relating to the Lx language family includes, but is not limited to supplementing languages L0', L1, L2, and L3 with the features of the language L0+ – the extended version of L0 [1].

## References

1. S. Rikacovs, *The base transformation language L0+ and its implementation*, Scientific Papers, University of Latvia, "Computer Science and Information Technologies", 2008.
2. *MDA Guide Version 1.0.1.* OMG, document omg/03-06-01, 2003.
3. E.D.Willink, *UMLX - A Graphical Transformation Language for MDA*, 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture , OOPSLA'2003, Anaheim, 2003.
4. T. Clark, A. Evans, P. Sammut, J. Willans. *Language Driven Development and MDA*, BPTrends, MDA Journal, Oct 2004.

5. J. Bezivin, E. Breton, G. Dupe, P. Valduriez. *The ATL Transformation-based Model Management Framework*, Research Report No 03.08, 2003, IRIN, Universite de Nantes.
6. A. Kalnins, J. Barzdins, E. Celms, *Model Transformation Language MOLA*. Proceedings of MDAFA 2004, Vol. 3599, Springer LNCS, 2005, pp. 62-76.
7. MOLA project, http://mola.mii.lu.lv
8. J. Barzdins, G. Barzdins, R. Balodis, K. Cerans, A. Kalnins, M. Opmanis, K. Podnieks, *Towards Semantic Latvia*. Communications of the 7th International Baltic Conference on Databases and Information Systems (Baltic DB&IS'2006), Vilnius, 2006, pp. 203-218.
9. The Base Transformation Language L0, http://lx.mii.lu.lv/L0_plus_CurrVers_2_4.pdf
10. J. Barzdins, A. Kalnins, E. Rencis, S. Rikacovs, *Model Transformation Languages and their Implementation by Bootstrapping Method.* Pillars of Computer Science, Vol. 4800, Springer LNCS, 2008, pp. 130-145.
11. J. Gallier, *Logic for Computer Science: Foundations of Automatic Theorem Proving*, Wiley, 1986.
12. L.M. Garshol, *BNF and EBNF: What are they and how do they work?*, http://www.garshol.priv.no/download/text/bnf.html
13. B. Efron, R.J. Tibshirani, *An Introduction to the Bootstrap*, Chapman & Hall/CRC, 1994, p. 436.
14. A.V. Aho, R. Sethi, J.D. Ullman, *Compilers - Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, 1986, p. 796.
15. A. Sostaks, A. Kalnins, *The implementation of MOLA to L3 compiler*, Scientific Papers, University of Latvia, "Computer Science and Information Technologies", 2008.
16. The Lx transformation language set home page, http://Lx.mii.lu.lv
17. J. Barzdins, A. Zarins, K. Cerans, A. Kalnins, E. Rencis, L. Lace, R. Liepins, A. Sprogis, *GrTP: Transformation Based Graphical Tool Building Platform*, MODELS 2007, Workshop on Model Driven Engineering Languages and Systems, 2007.

## Appendix

### A. L3 syntax definition based on Backus-Naur notation

```
<L3Program>       ::= <transformation> [ <L3Program> ]
<transformation>  ::= transformation <identifier> ;
<transfPartList> endTransformation;
<identifier>      ::= <letter> [ <string> ]
<specialID>       ::= <specialLetter> [ <string> ]
<string>          ::= <letter> [ <string> ] | <digit> [
<string> ]
<letter>          ::= <specialLetter> | _
<specialLetter>   ::= a | b | c | d | e | f | g | h | i |
j | k | l | m | n | o | p | q | r | s | t | u | v | w | x
```

```
| y | z | A | B | C | D | E | F | G | H | I | J | K | L |
M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
<digit>          ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
9
<transfPartList>  ::= <transfPart> [ <transfPartList> ]
<transfPart>      ::= <nativeProc> | <nativeFunct> |
<directive> | <debug> | <varDeclaration> | <procedure> |
<function>
<nativeProc>      ::= native procedure <identifier> ( [
<paramList> ] );
<paramList>       ::= <parameter> [ , <paramList> ]
<simpleParamList> ::= <identifier> [ , <simpleParamList>
]
<parameter>       ::= <identifier> : <primTypeOrMMElem>
<nativeFunct>     ::= native function <identifier> ( [
<paramList> ] ): <primTypeOrMMElem> ;
<directive>       ::= <dirType> " <fileName> ";
<dirType>         ::= useMM | include | useLib | useUnit
<fileName>        ::= [ <specialLetter> :\] [
<folderList> ] <string> [ . <string> ]
<folderList>      ::= <string> \ [ <folderList> ]
<debug>           ::= DEBUG_ON; | DEBUG_OFF;
<varDeclaration>  ::= <primitiveVarDecl> | <pointerDecl>
<primitiveVarDecl>::= var <specialID> : <primTypeName> ;
<primTypeName>    ::= Integer | Real | String | Boolean
<pointerDecl>     ::= pointer <identifier> :
<metaModelElement> ;
<metaModelElement>::= <letter> [ <stringPlus> ]
<stringPlus>      ::= <stringPlusElem> [ <stringPlus> ]
<stringPlusElem>  ::= <letter> | <digit> | # | ::
<procedure>       ::= [ main ] procedure <identifier> ( [
<paramList> ] ); [ <varList> ] begin; [ <L3CommandList> ]
end;
<function>        ::= function <identifier> ( [
<paramList> ] ): <primTypeOrMMElem> ; [ <varList> ]
begin; [ <L3CommandList> ] end;
<primTypeOrMMElem>::= <primTypeName> | <metaModelElement>
<varList>         ::= <varDeclaration> [ <varList> ]
<L3CommandList>   ::= <L3Command> [ <L3CommandList> ]
<L3Command>       ::= <call> | <return> | <first> |
<next> | <goto> | <label> | <addObj> | <addLink> |
<deleteObj> | <deleteLink> | <setPointer> | <setPointerF>
| <setVar> | <setVarF> | <setAttr> | <type> | <var> |
<pointer> | <attr> | <link> | <noLink> | <debug> |
<foreach> | <if>
<call>            ::= call <identifier> ( [
<simpleParamList> ] );
<return>          ::= return [ <identifier> ] ;
```

```
<first>           ::= first <identifier> :
<metaModelElement> [ from <identifier> by
<metaModelElement> ] [ suchthat begin [ <L3CommandList> ]
end ] [ else <specialID> ] ;
<next>            ::= next <identifier> [ suchthat begin
[ <L3CommandList> ] end ] [ else <specialID> ] ;
<goto>            ::= goto [ <specialID> ] ;
<label>           ::= label <specialID> ;
<addObj>          ::= addObj <identifier> :
<metaModelElement> ;
<addLink>         ::= addLink <identifier> .
<metaModelElement> . <identifier> ;
<deleteObj>       ::= deleteObj <identifier> ;
<deleteLink>      ::= deleteLink <identifier> .
<metaModelElement> . <identifier> ;
<setPointer>      ::= setPointer <identifier> =
<identifier> ;
<setPointerF>     ::= setPointerF <identifier> =
<identifier> ( [ <simpleParamList> ] );
<setVar>          ::= setVar <specialID> = <expression> ;
<setVarF>         ::= setVarF <specialID> = <identifier>
( [ <simpleParamList> ] );
<setAttr>         ::= setAttr <identifier> .
<metaModelElement> = <expression> ;
<type>            ::= <identifier> <pointerRelOp>
<metaModelElement> [ else <specialID> ] ;
<var>             ::= <identifier> <relationOperator>
<expression> [ else <specialID> ] ;
<pointer>         ::= <identifier> <pointerRelOp>
<identifier> [ else <specialID> ];
<attr>            ::= attr <identifier> .
<metaModelElement> <relationalOperator> <expression> [
else <specialID> ];
<link>            ::= link <identifier> .
<metaModelElement> . <identifier> [ else <specialID> ] ;
<noLink>          ::= noLink <identifier> .
<metaModelElement> . <identifier> [ else <specialID> ] ;
<foreach>         ::= foreach <identifier> :
<metaModelElement> [ from <identifier> by
<metaModelElement> ] [ suchthat begin [ <L3CommandList> ]
end ] do begin [ <L3CommandList> ] end;
<if>              ::= if begin [ <L3CommandList> ] end
then begin [ <L3CommandList> ] end [ else begin [
<L3CommandList> ] end ] ;
<expression>      ::= <boolExpr> | <notBoolExpr>
<boolExpr>        ::= true | false
<notBoolExpr>     ::= <exprPart> [ <arithmeticOper>
<notBoolExpr> ]
```

```
<exprPart>          ::= <const> | <identifier> |
<attribute> | <functionCall>
<arithmeticOper>  ::= + | - | * | /
<const>             ::= <integerConst> | <realConst> |
<stringConst>
<integerConst>    ::= <positiveNum> | (- <positiveNum> )
<positiveNum>     ::= <digit> [ <positiveNum> ]
<realConst>        ::= <positiveNum> . <positiveNum> | (-
<positiveNum> . <positiveNum> )
<stringConst>     ::= ″ <extendedString> ″
<extendedString>  ::= <symbol> [ <extendedString> ]
<symbol>            ::= <letter> | <digit> |
<relationOperator> | ~ | ` | ! | @ | # | $ | % | ^ | & |
( | ) | { | } | : | < | > | ? | [ | ] | ; | ' | \ | , | .
| <space>
<attribute>        ::= <identifier> . <metaModelElement> :
<primTypeName>
<functionCall>    ::= <identifier> ( [ <simpleParamList>
] )
<relationOperator>::= <pointerRelOp> | < | > | <= | >=
<pointerRelOp>    ::= == | !=
<space>             ::=
```

Note – the non-terminal symbol <space> is considered to be a space symbol (32th symbol in the ASCII code table).

LATVIJAS UNIVERSITĀTES RAKSTI. 2008, 733. sēj.:
DATORZINĀTNE UN INFORMĀCIJAS TEHNOLOĢIJAS 140.–178. lpp.

# The Implementation of MOLA to L3 Compiler

Agris Sostaks[1], Audris Kalnins[2]
[1,2] University of Latvia, Institute of Mathematics and Computer Science,
Raina blvd 29,LV-1459 Riga, Latvia
[1]Agris.Shostaks@gmail.com, [2]Audris.Kalnins@mii.lu.lv

**Abstract.** The implementation of the model transformation language MOLA compiler to the L3 language is described in the paper. It is shown that L3 is a suitable low-level model transformation language for efficient implementation of pattern matching in MOLA. A rationale for the chosen compiler architecture is offered. The detailed description of mappings from MOLA to L3 is also given. Some general approach to the graphical language compiler development, such as model-driven compiling and debugging, is also sketched.

**Keywords:** Graphical model transformation language, MOLA, L3, Lx, compiler, model-driven compiling.

## 1 Introduction

**Model transformations** play an important role in the Model-Driven Software Development (MDSD) [1]. The main idea of MDSD is a systematic use of **models** as primary software engineering artefacts throughout the software development lifecycle. Model-Driven Development refers to a range of development approaches that are based on the use of software modelling. A model expresses a particular aspect of a software system in a certain level of detail. A code of the software system is generated from models built by a system developer. The generated code varies ranging from a system skeleton to a complete product. It depends on the abstraction level of models used as a source for the generator. If the created models are at high level of abstraction, then **model transformations** are applied to create more detailed models that can be used for code generation. The model transformation is the automatic generation of a target model from a source model, according to a transformation definition [2]. Model transformation languages are used to define model transformations. Models that are used by model transformations must conform to **metamodels**. A metamodel defines a language which specifies a model. A model transformation language uses metamodels to define the model transformation. A meta-language specifies the metamodels. The general architecture of model transformations is shown in Fig.1.
The best known Model-Driven Software Development initiative is the Object Management Group (OMG) [3] Model-Driven Architecture (MDA) [4], which is a registered trademark of OMG. The OMG has developed the set of standards related to

[1] Partially supported by ESF (European Social Fund),
project 2004/0001/VPD1/ESF/PIAA/04/NP/3.2.3.1/0001/0001/0063

MDA, including the Meta-Object Facility (MOF) [5] (a meta-language), Object Constraint Language (OCL) [6], Unified Modelling Language (UML) [7] (a software development language), and MOF Queries/Views/Transformations (MOF-QVT) [8] (a model transformation language).

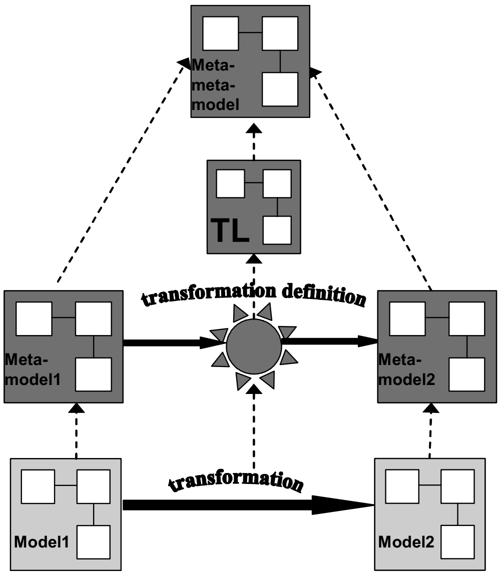


**Fig. 1** Model transformation

The MDA approach defines system functionality using a platform-independent model (PIM) that is written in an appropriate modelling language (for example, UML). Then the PIM is transformed to one or more platform-specific models (PSMs), which include platform- or language-specific details. For example, the UML Profile for Java [9] can be used to specify the PSM. Then the PSM is translated to the code written in the language appropriate to the PSM.

Today the application area for model transformation languages is much broader. One such area is generic meta-model-based modelling tool building. The model transformation languages can be used (and are used [10, 11, 12]) as a much more effective domain specific substitute for the general purpose languages that are used for tool building up to now. This paper shows that model transformation languages also become appropriate facilities for compiler building. Thus, domains for applications of model transformation languages are quite different, but the typical language constructs used for model processing in all these domains are quite similar.

The OMG was the first to state precisely the requirements what should be a model transformation language [13]. The MOF-QVT language, which is an answer by OMG itself to these requirements, becomes the OMG standard for model transformations [8]. In MOF-QVT source and target meta-models conform to the MOF. There are two variants of MOF defined – the EMOF (Essential MOF) and the CMOF (Complete MOF). The MOF can be viewed as a general standard to write metamodels, but, more specifically, EMOF is used for metamodel definition in MOF-QVT. The MOF-QVT standard defines two languages of transformation development – the *Relations* and the

*Operational Mappings*. The *Relations* language is at the highest level of abstraction and uses patterns and a declarative transformation definition style whenever possible. This language has two semantically equivalent concrete syntaxes – a graphical and a textual one. The *Operational Mappings* language is an imperative textual language. The syntax of the *Operational Mappings* provides constructs commonly found in imperative languages (loops, conditions, etc), while the management of model elements is based on extended OCL constructs. Actually, the MOF-QVT specification [8] also contains the third language – the *Core*. The role of this language is to serve for semantic definition of the first two OMG languages and also for possible implementation of these languages. There are several realizations of the MOF-QVT language. The *Relations* textual language is implemented in the *medini QVT* [14]. The *Operational Mappings* language is implemented in the *SmartQVT* [15], several less complete implementations are also available.

There are many other model transformation languages which also satisfy the OMG requirements. There are textual model transformation languages – ATL [16], VIATRA2 [17], the Lx language family (L0-L3) [18] and also graphical model transformation languages – Fujaba [19], GReAT [20], MOLA [21]. In fact, model transformation languages existed even before the OMG coined this concept. These were the graph transformation languages, which were used to transform a source graph to a target graph in a rule-based manner. The structure of both graphs was defined by means of graph grammars which, in fact, are the same metamodels. There are several such graph transformation languages that are now being used as the model transformation languages, for example, AGG [22] and PROGRES [23].

Most of the model transformation languages rely on an EMOF-compatible meta-language for defining metamodels. For example, Fujaba and GReAT use class diagram notations close to EMOF, and ATL uses KM3 [24] (a certain extension of EMOF). Sometimes meta-languages are used that are much more expressive than EMOF, for example, VTML [25] for the VIATRA2 language. An implementation of a metamodelling language is closely related to the specific repository used for storing models.

An efficient implementation of model transformation languages is still a topical issue. There are several possibilities of implementation. A direct compilation to a general purpose programming language is a common approach (AGG, Fujaba, GReAT). The result of the compilation contains invocations of the API of the repository used to manage models and the corresponding metamodel. Another possibility is a compilation to an intermediate "very low-level" transformation language, for example, ATL uses the so called ATL byte-code [26]. It is also possible to build a direct interpreter of a model transformation language, as it is done for the VIATRA2 language.

The model transformation language MOLA is developed by the University of Latvia, Institute of Mathematics and Computer Science. This paper describes the implementation of the MOLA compiler. The MOLA compiler uses a different approach by compiling MOLA to L3, which is a lower-level textual model transformation language, but still has features typical of a transformation language. The L3 language is an imperative language which also includes imperative facilities for pattern definition; therefore, the compilation of declarative patterns in MOLA is the only complicated part of MOLA to L3 compiler realization. The L3 language is efficient regarding implementation [27], and it is also developed by UL, IMCS. The L3 language is also used for the development of MOLA compiler. In other words, the compiler itself is built as a model transformation. Therefore, the chosen implementation is relatively simple and at the same time guarantees efficiency of implementation.

A brief introduction to the MOLA language is given in chapter 2. The experience gained in building the previous MOLA realizations is described in chapter 3. The language family Lx is introduced in chapter 4. The general architecture of the MOLA compiler and a brief overview of the model-driven compiling are given in chapter 5. Mappings from MOLA to L3 are described in details in chapter 6. Chapter 7 contains MOLA environment problem descriptions and possible solutions that are not directly related to the compiling process.

## 2 MOLA Language

MOLA is a graphical model transformation language, which is used for transforming an instance of a source metamodel (the source model) into an instance of the target metamodel (the target model). A transformation definition in MOLA consists of the source and target metamodel definitions and one or more MOLA procedures.

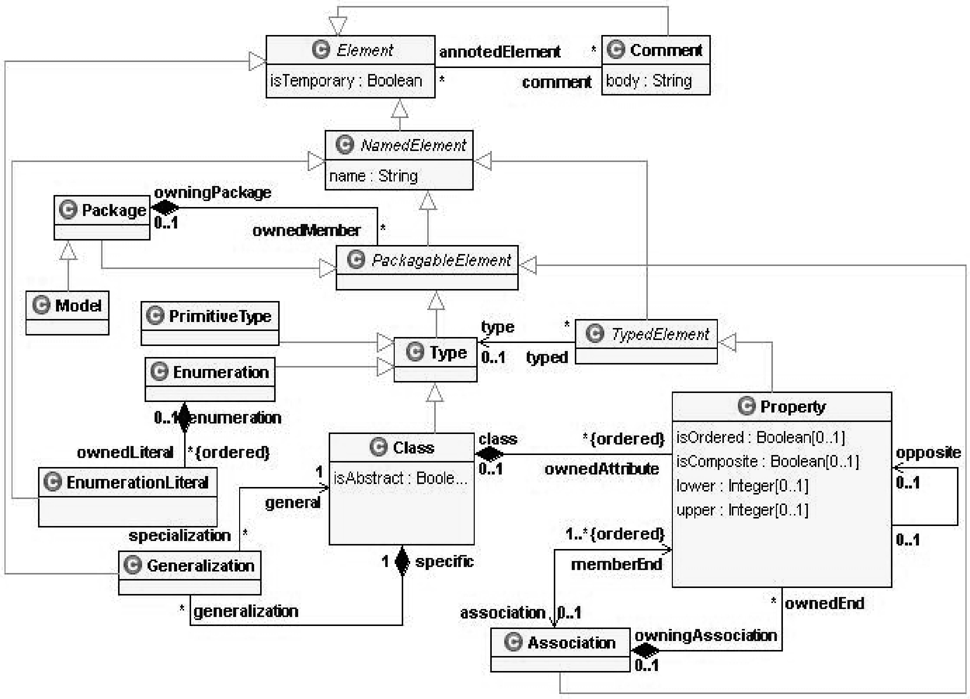


**Fig. 2**. The metamodel of the MOLA metamodelling language

Source and target metamodels are jointly defined in the MOLA metamodelling language, which is quite close to the OMG EMOF specification [8]. These metamodels are defined by means of one or more class diagrams, packages may be used in a standard way to group the metamodel classes. Actually, the division into source and target parts of the metamodel is quite semantic, as they are not separated syntactically (the complete metamodel may be used in transformation procedures in a uniform way). Typically, additional mapping associations link the corresponding classes from source and target metamodels; they facilitate the building of natural transformation procedures and document the performed transformations. The source and target metamodel may be the same – that is the case for in-place model update transformations. The MOLA metamodelling language is defined formally in the Kernel package of the MOLA metamodel (see Fig. 2).

MOLA procedures form the executable part of a MOLA transformation. One of these procedures is the main one, which starts the whole transformation. MOLA procedure is built as a traditional structured program, but in a graphical form. Similarly to UML activity diagrams (and conventional flowcharts), control flow arrows determine the order of execution of MOLA statements. Call statements are used to invoke sub-procedures. However, the basic language statement of MOLA procedures is specific to the model transformation domain – it is the **rule**. Rules embody the pattern match paradigm, which is typical of model transformation languages. Each rule in MOLA has the pattern and the action part. Both are defined by means of **class-elements** and **-links**. A class-element is a metamodel class, prefixed by the element ("role") name (graphically shown in a way similar to UML instance). An association-link connecting two class-elements corresponds to an association linking the respective classes in the metamodel. A pattern is a set of class-elements and -links which are compatible to the metamodel for this transformation. A pattern may simply be a metamodel fragment, but a more complicated situation is also possible – several class-elements may reference the same metamodel class – certainly, their element names must differ (these elements play different roles in the pattern, e.g., the start and end node of an edge). A class-element may also contain a constraint – a Boolean expression in a simplified subset of OCL. The main semantics of a rule is in its pattern match – an instance set in the model must be found, where an instance of the appropriate class is allocated to each class-element so that all required links are present in this set and all constraints evaluate to true. If such a match is found, the action part of the rule is executed. The action part also consists of class-elements and links, but typically these are create-actions – the relevant instances and links must be created. An end of a create-link may also be attached to a class-element included in pattern. Assignments in class-elements may be used to set the attribute values of the instances. Instances may also be deleted and modified in the action part. Thus a rule in MOLA typically is used to locate some construct in the source model and build a required equivalent construct in the target model. If several instance sets in the model satisfy the rule pattern, the rule is executed only once (on an arbitrarily chosen match). Such a situation should be addressed by another related construct in MOLA – the loop construct. In addition, the reference mechanism (a class-element may be a reference to an already matched or created instance in a previous rule) is used to restrict the available match set. Thus, rules are typically used in MOLA in situations where at most one match is possible. Certainly, there may be a situation when no match exists – then the rule is not executed at all. To distinguish this situation, a rule may have a special *ELSE*-exit (a control flow labelled *ELSE*), which is traversed namely in this situation. Thus, a rule plays in MOLA the role of an *if-then-else* construct as well.

Another essential construct in MOLA is the **loop** (more concretely, for-each loop). The loop is a rectangular frame, which contains one special rule – the **loophead**. The loophead is a rule which contains one specially marked (by a bold border) element – the loop variable. The semantics of a for-each loop is that it is executed for all possible matches for the loophead, which differ by instances allocated to the loop variable (possible variations for other loop head elements are not taken into account). In fact, a for-each loop is an iterator which iterates through all possible instances of the loop variable class that satisfy the constraint imposed by the pattern in the loophead. With respect to other elements of the pattern in the loop head, the "existential semantics" is in use – there must be a match for these elements, but it does not matter whether there are one or several such matches. Thus a for-each loop is the main MOLA construct, which is used to code a situation: "for each instance of . . . which satisfies . . . perform the following transformation. . . ". Namely such situations in informal descriptions of model transformations are frequently called transformation rules, but in MOLA they must be formalised as for-each loops. In addition to the loophead, a loop typically contains the loop body – other MOLA statements whose execution order is organised by control flows. The loop body is executed for each iteration of the loop. Since the loop head is a rule, it may also contain create actions, thus simple transformations of source model elements may be coded in MOLA by loops consisting of the loop head only. For nested loops the main organising feature is the possibility to reference the loop variable (and other elements) of the main loop in the pattern of the nested loop head, thus specifying an iteration over all related instances (to the current instance in the main loop).

There also are other available constructs in MOLA procedures. Procedures may have **parameters** (of type of a metamodel class or a primitive type) and local **variables** (also of both types). These elements may be used in MOLA rules, in addition, **text-statements** (consisting of a constraint and assignments) may be used to process these elements more directly. For primitive-typed variables the text statement is the only option. A text statement containing a constraint (a Boolean expression) may also have an *ELSE*-exit and serve as an *if-then-else* construct (in addition to rule). Besides MOLA procedures, external (coded in an OOPL) procedures can also be invoked; this feature is used for low-level data processing (e.g., model data import). It should be noted that MOLA has no built-in UI support (MOLA is oriented towards behind-the-scenes transformations), therefore diagnostic messages and similar situations should be addressed via a library of external procedures. All MOLA procedure elements are defined formally in the MOLA package of the MOLA metamodel (see Fig. 3).

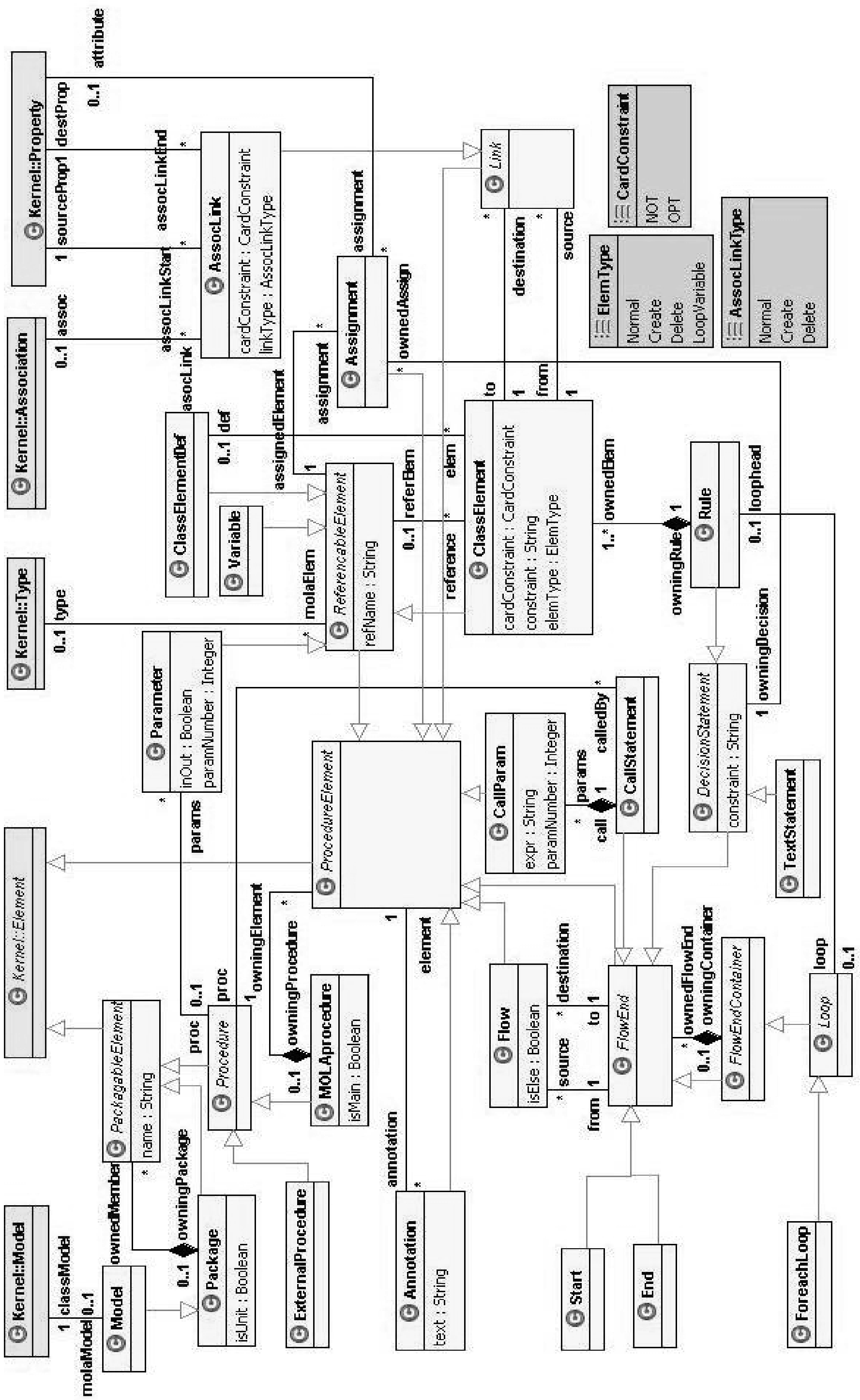


**Fig. 3**. The metamodel of the MOLA procedure elements

The execution of a MOLA transformation on a source model starts from the main procedure. A loop is executed while there are instances to iterate over, then the next construct according to the control flow is executed. If a rule without a valid match is to be executed, and this rule has no *ELSE*-exit, then the current procedure is terminated (if this occurs outside a loop) or the next iteration of the loop is started (within a loop body). When the main procedure reaches its end, the transformation is completed.

## 3 Previous Realizations of MOLA

The most critical part of the implementation of a pattern-based transformation language is the implementation of the pattern matching. It has been already shown [28] that an efficient MOLA pattern matching implementation is possible. This realization is based on only few specific low-level operations needed to iterate over a model. They are:

- `getNext(Class Cl)` – returns the next instance of a metaclass `Cl` upon each call. There is also an initialization for it – `initializeGetNext(Class Cl)`
- `getNextByLink(Association as, Cl1 inst, Class Cl2)` – returns one by one instances of a metaclass `Cl2` that can be reached by links corresponding to association `as` from a fixed instance `inst`. There is also an initialization for it, with similar parameters – `initializeGetNextByLink(Association as, Cl1 inst, Class Cl2)`
- `checkLink(Cl1 inst1, Cl2 inst2, Association as)` – checks whether a link of the required type is between these instances
- `eval(Cl inst, Expr exp)` – evaluates a local constraint on attributes

Thus, the target language of the MOLA compiler or the API of a repository that is used for realization of the MOLA interpreter (Virtual Machine) must contain similar operations. This approach requires the implementation of the pattern matching algorithm using such low-level constructs. That is a sufficiently complicated task.
Another approach that can be used for pattern matching is to rely on some powerful high-level pattern matching language and build mappings from MOLA to it. An appropriate model repository must also be chosen.

The previous realization of MOLA [29] used SQL queries as a pattern matching language and a relational database as the model repository. A fixed database schema had been defined in the most natural way by storing the metamodel in tables which correspond to the EMOF metamodel classes. The storage of model elements – instances of metamodel classes, associations, and attributes was completely straightforward in the corresponding tables. A MOLA program was also naturally stored in tables according to the MOLA metamodel. The main idea was to map a MOLA pattern to a single SQL statement. SQL queries generated by this realization were large self-join queries that are non-typical of standard database applications. The database engines were performing efficiently for queries if the number of class elements in a MOLA pattern did not exceed a certain number. Experiments and benchmark tests had shown that the implemented MOLA Virtual Machine performed satisfactorily and MOLA is a suitable transformation language for typical MDSD tasks. However, for an industrial usage of MOLA a special in-memory repository and a compiler/interpreter that implements the principles described in [28] is required.

The next step in the realization of the model transformation language MOLA was to search for a solution which satisfies the requirements mentioned above.

## 4 Lx Language Family

The search for a suitable solution for the MOLA realization revealed that an appropriate language and also a repository could be found nearby. The model transformation languages Lx [18] (the so called Lx language family) fulfil the requirements mentioned in the previous chapter. Textual model transformation languages Lx contain the base transformation language L0 and its related transformation languages L0', L1, L2 and L3. Each of these languages is based on the previous language of this family by adding some extra features.

The model transformation language L3 has been chosen as a target language for the MOLA compiler. A more detailed description of the Lx language family is available in [32] and [27]; however, a brief overview of all these languages is given in this chapter in order to make this paper understandable without reading the papers mentioned above.

### 4.1 Lx Metamodelling Facilities

The Lx language family, as any other model transformation language, uses some sort of metamodelling language. It is quite close to the OMG EMOF specifications. The main difference is that multiple generalization is not allowed and there are no packages in this metamodelling language. The metamodel of this language is shown in Fig. 4.

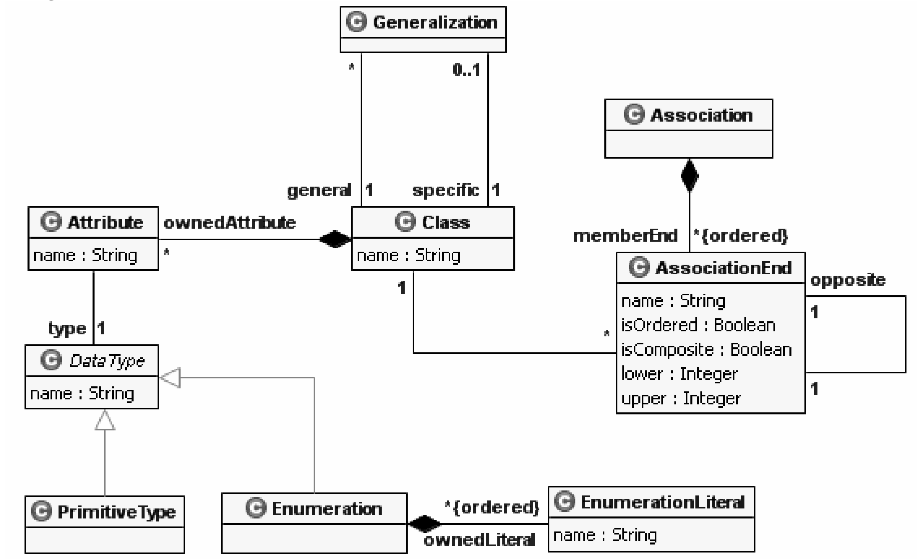


**Fig. 4.** The metamodel of Lx metamodelling language

Classes and binary associations are core elements of this language. Classes can have attributes which can be primitive or enumeration-typed. There are four pre-defined primitive types – *String*, *Integer*, *Boolean*, and *Real*. There are no possibilities to define new ones.

The basic commands (constructs for a textual definition of a metamodel) of the Lx family metamodelling language are the following:

- **class** <className>**;** – defines class with a given name.
- **attr** <className>**.**<attrName>**:**<ElementaryTypeName>**;** – defines attribute with a given name and type.
- **assoc** <className>**.{ordered}**<card><roleName>/<roleName><card>**{ordered}.** <className>**;** – defines association with corresponding properties.
- **compos** <compositeClassName>**.{ordered}**<card><roleName>/<roleName><card> **{ordered}.**<partClassName>; **–** defines compositions with corresponding properties.
- **rel** <subClassName>**.subClassOf.**<superClassName>**; –** defines a generalization relationship between given classes.
- **enum** <enumName>**:{** <enumLiteral1> , < enumLiteral2>, … **}; –** defines enumeration with given elements.

## 4.2 Language L0

An elementary unit of L0 transformation is a **command** (an imperative statement). L0 transformation contains several parts:

- global variable definition part;
- native subprogram (function or procedure) declaration part (used C++ library function headers);
- L0 subprogram definition part. Exactly one subprogram in this part is the **main**. The main subprogram defines the entry point of the transformation. An L0 subprogram definition also consists of several parts:
  - Subprogram header
    - **procedure** <procName>**(**<paramList>**);** Subprogram header, the (formal) parameter list can be empty. Parameter list consists of formal parameter definitions separated by “,”. A parameter definition consists of its name, the parameter type (the type can be an elementary type or a class from the metamodel), and the passing method (parameters can be passed by reference or by value). If the parameter is passed by reference, its type name is preceded by the **&** character.
    - **function** <funcName>**(**<paramList>**):** <returnType>; **–** return type name can be an elementary type name or class name.
  - Local variable definitions

        - **pointer** <pointerName> **:** <className>**;** – defines a pointer to objects of class <className>.
        - **var** <varName> **:** <ElementaryTypeName>**;** – defines a variable of elementary type. <ElementaryTypeName> is one of elementary types.
    - Keyword **begin** – starts subprogram body definition
    - Subprogram body definition
    - Keyword **end -** ends subprogram body definition.

The subprogram body definition may contain the following commands:

1. **return;** – returns execution control to caller procedure or function.
2. **call** <subProgName>**(**<actPrmList>**);** – calls a subprogram. Actual parameters list can be empty. Actual parameter list consists of binary expressions separated by "**,**".
3. **label** <labelName>**;** – defines a label with the given name.
4. **goto** <label>**;** – unconditionally transfers control to <label>. <label> should be located in the current subprogram.
5. **first** <pointer> **:** <className> **else** <label>**;** – positions <pointer> to an arbitrary object of <className>. Typically, this command in combination with the **next** command is used to traverse all objects of the given class (including subclass objects). If <className> does not have objects, <pointer> becomes **null**, and execution control is transferred to the <label>. The <className> in this command must be the same as (or a subclass of) the class used in pointer definition. If it is a subclass, then the pointer value set is narrowed (for the subsequent executions of **next**).
6. **first** <pointer1> **:** <className> **from** <pointer2> **by** <roleName> **else** <label>**;** – similar to the previous command. The difference is that it positions <pointer1> to an arbitrary class object, which is reachable from <pointer2> by the link <roleName>. Similarly, this command in combination with the **next** command is used to traverse all objects linked to an object by the given link type.
7. **next** <pointer> **else** <label>**;** – gets the next object, which satisfies conditions, formulated during the execution of the corresponding **first** and which has not been visited (iterated) with this variable yet. If there is no such object, the <pointer> becomes **null**, and execution control is transferred to <label>.
8. **addObj** <pointer>**:**<className>**;** – creates a new object of the class <className>.
9. **addLink** <pointer1>**.**<roleName>**.**<pointer2>**;** – creates a new link (of type specified by <roleName>) between the objects pointed to by the <pointer1> and <pointer2> , respectively.
10. **deleteObj** <pointer>**;** – deletes the object, which is pointed to by <pointer>.
11. **deleteLink** <pointer1>**.**<roleName>**.**<pointer2>**;** – deletes link whose type is specified by <roleName> between objects pointed to by <pointer1> and <pointer2>, respectively.

12. **setPointer** <pointer1>=<pointer2>**;** – sets <pointer1> to the object which is pointed to by <pointer2>. Instead of <pointer2> the *null* constant can be used.
13. **setVar** <variable> = <binExpr>**;** – sets <variable> to <binExpr> value. <binExpr> is a *binary* expression consisting of the following elements: *elementary variables, subprogram parameters (of elementary types), literals, object attributes*, and *standard operators* (*+,-,*,/,&&,||,!*).
14. **setAttr** <pointer>**.**<attrName>=<binExpr>**;** – sets the value of attribute <attrName> (of the object, pointed to by <pointer>) to the <binExpr> value.
15. **type** <pointer> == <className> **else** <label>**;** – if the type of the pointed object is identical to the <className>, then control is transferred to the next command, else control is transferred to <label>. Instead of the equality symbol == an inequality symbol != can be used. This command is used for determining the exact subclass of an object.
16. **var** <variable>==<binExpr> **else** <label>**;** – if the condition is *true*, then control is transferred to the next command, else control is transferred to <label>. Instead of equality symbol other (<, <=, >, >=, !=) relational operators compatible with argument types can be used.
17. **attr** <pointer>**.**<attrName> == <binExpr> **else** <label>**;** – if the condition is *true,* then control is transferred to the next command, else control is transferred to <label>. Other relational operators (<, <=, >, >=, !=) can be used too.
18. **link** <pointer1>**.**<roleName>**.**<pointer2> **else** <label>**;** – checks whether there is a link (with the type specified by <roleName>) between the objects pointed to by <pointer1> and <pointer2>, respectively.
19. **pointer** <pointer1>==<pointer2> **else** <label>**;** – checks whether the objects pointed to by <pointer1> and <pointer2> are identical. Instead of <pointer2> *null* constant can be used. The inequality symbol (!=) can be used too.

It is easy to see that the language L0 contains only the very basic facilities for defining transformations [32].

## 4.3 Languages L0' – L3

**Language L0'** – model transformation language L0' is based on the language L0. The new feature of L0' is the possibility to make long arithmetic expressions (in L0, only unary and binary expressions were allowed).
**Language L1** – is supplemented with an imperative pattern matching feature, so that it is possible to search for instances that match some condition. Any L1 pattern can contain conditions on values of variables or attributes, links between instances and other. In fact, all L1 commands can be used to specify pattern condition.
The textual syntax for the pattern (*such-that* block) is as follows:

```
suchthat
begin
<L1Commands>
end;
```

The condition holds if it is possible to successfully [27] reach the end of the block (i.e., successfully execute its last command). The "conditional" commands in L0 (commands that have an **else** branch) may be used without the **else** branch in the *such-that* block. If in such a command the undefined **else** branch is to be executed, then the condition defined by the pattern fails.
The *such-that* block may be used with **first** and **next** commands.
**Language L2 –** has the possibility to make loops. A special command exists in L2 with which it is possible either to visit all instances of the specified class or just those instances of the class that match the given pattern. The textual syntax for the loop is as follows:

```
foreach <pointerName1> : <className> [ from
<pointerName2> by <roleName> ] [ suchthat
  begin
      <L2Commands>
  end ]
  do
  begin
      <L2Commands>
  end;
```

**Language L3 –** has the branching command – a standard *if-then-else* construct can be used. The textual syntax of the branching command is as follows:

```
  if
  begin
      <L3Commands>
  end
  then
  begin
      <L3Commands>
  end
  [ else
  begin
      <L3Commands>
  end ];
```

The L3 metamodel (the Lx language family metamodel) is shown in Fig. 5.

## 4.4 MOLA and L3

The main reasons why the Lx model transformation language family and the L3 language, particularly, have been chosen are described in this section.

One of the main requirements that must be met is the compatibility of metamodeling languages. In our case metamodelling languages are EMOF-based for both MOLA and Lx language family. There are no significant differences between both languages, but such minor issues like absence of packages in Lx family metamodeling language can be resolved using name prefixes for class names. Thus, we can claim that MOLA and Lx metamodeling languages are fully compatible.

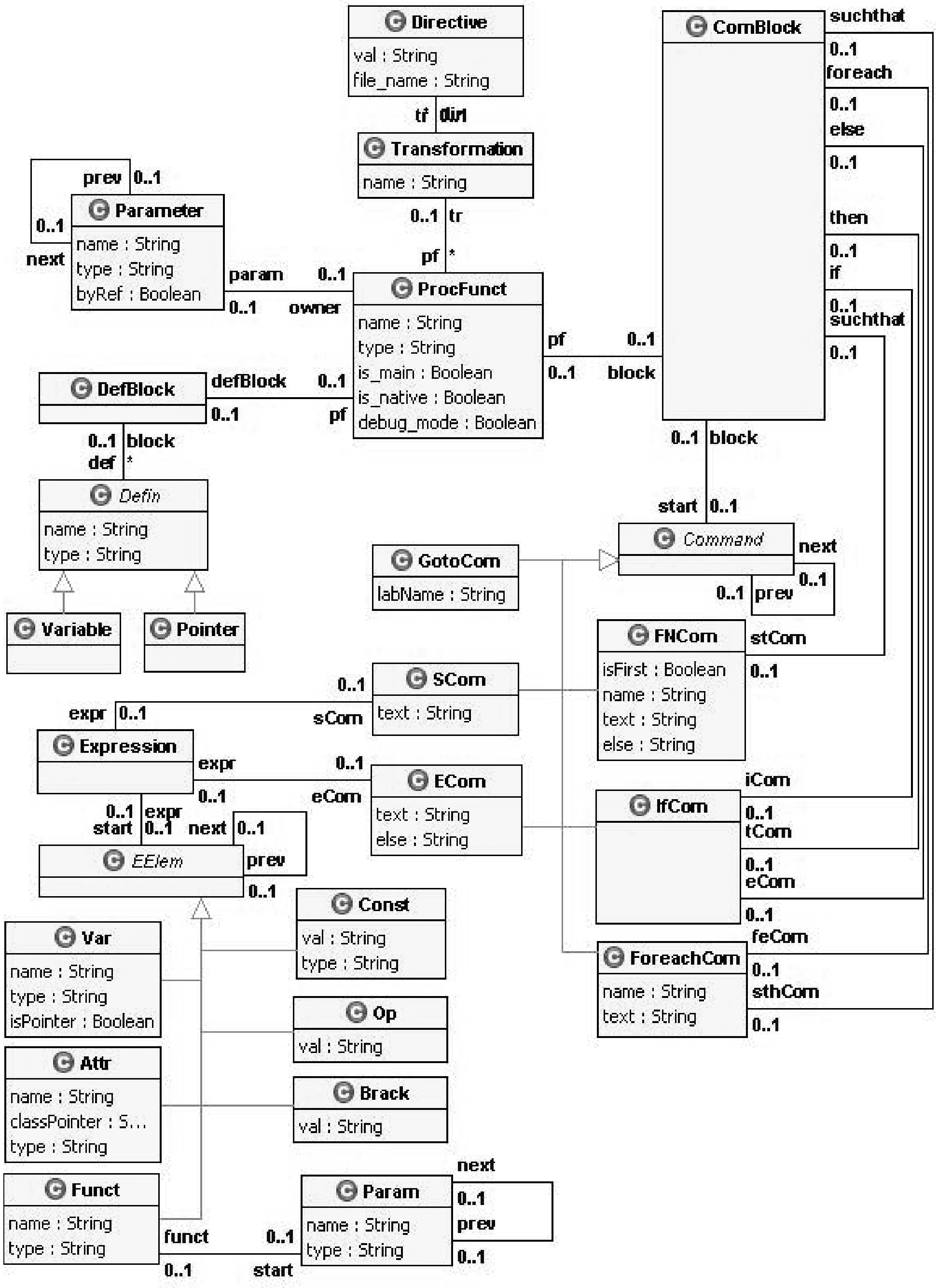


**Fig. 5**. The metamodel of L3 language

It has already been shown [28] that MOLA language can be implemented efficiently using a set of low-level operations for patterns. There is a direct mapping from the required operations to the commands of Lx model transformation family.

- `initializeGetNext(Class Cl)`and `getNext(Class Cl)` operations can be mapped to **first** *c:Cl* and **next** *c* commands. These commands return all instances of a given meta-class. In the beginning the **first** *c:Cl* command must be called to initialize the iteration through required instances and afterwards the **next** *c* must be called to iterate through.
- `initializeGetNextByLink(Association as, Cl1 inst, Class Cl2)` and `getNextByLink(Association as, Cl1 inst, Class Cl2)` operations can be mapped to the **first** *c:Cl2* **from** *inst* **by** *as* and **next** *c* commands. These commands return all instances of a given meta-class navigable by links of the given type from a fixed instance. The iteration must be done similarly to the previous case.
- `checkLink(Cl1 inst1, Cl2 inst2, Association as)` operation can be mapped to the **link** *inst1.as_rolename.inst2* command. The semantics of this command is the same as the semantics of this operation – check the existence of a link of the given type between two fixed instances.
- `eval(Cl inst, Expr exp)` operation is an expression interpreter and the MOLA realization to L3 must implement a generator of sequences of L3 commands that interprets the given expression. The core elements of such expressions are attribute or variable value checks. These operations can be mapped to **attr** *inst.<attrname><relation><expression>* and **var** *<varname><relation><expression>* commands accordingly. Arithmetic expressions can be mapped to expressions introduced by the L0' language. Constraints that are complex (Boolean) expressions where conjunction, disjunction and negation are used can be mapped to a sequence of commands which interprets the given expression.

MOLA operations that create update and delete instances and links can be mapped to **addObj**, **addLink**, **setAttr**, **deleteObj**, **deleteLink** commands. The control flows in MOLA can be mapped to **label** and **goto** commands in L3 language. L3 language as well as MOLA has such concepts as *procedure*, *parameter*, *variable*, *sub-procedure call*. These concepts can be mapped directly from MOLA to L3 language. Thus L3 language provides all necessary features that allow us to build an efficient MOLA compiler.

These basic features are included in the L0' language, but commands introduced in the following languages L1-L3 (pattern matching, looping, and branching commands) allow much easier implementation of the MOLA compiler. That is possible because these commands are at an abstraction layer much closer to MOLA concepts, such as for-each loop and rule, than basic, L0 and L0', commands.

A detailed description of the mapping from MOLA to L3 is given in chapter 6 of this paper.

# 5 Architecture of MOLA Compiler

This chapter describes the general architecture of the MOLA compiler. It includes the chain of compilers from MOLA to L3, L3 to L0, L0 to C++, and C++ to executable code. An introduction to the model-driven compiling is also included in this chapter.

## 5.1 Implementation of the Lx Language Family

An efficient compiler has been already built [18] for the Lx language family. Actually, an efficient realization of the L0 language has been built, and a compiler for each next language is built using the bootstrapping method [30]. It means that the previous language in the family is used to build the compiler for the next one (L0 for L0' compiler, L0' for L1 compiler, and so on).

The metamodel-based in-memory repository [31] developed by the UL IMCS has been chosen to store metamodel and its instances for the implementation of L0 language. This repository has an appropriate low-level API implemented as a C++ function library. Therefore, the intermediate result of the L0 compilation is a C++ program. The final result of the L0 compilation is a dynamic link library (DLL file) that can be executed over a repository instance which contains the appropriate metamodel and model and must be loaded into memory. The experiments have shown that the repository itself and the selected way of compilation to the API [32] are efficient for the implementation of a model transformation language.

The bootstrapping method used to build compilers for the rest of the Lx family languages requires that programs written in L0' to L3 must be stored in the repository that is used by L0 language. Thus, the metamodel of the L3 language is required. All other languages of the Lx family are described by the same metamodel because each next language is derived from the previous one by adding some new features; therefore, the metamodel of the last language in the chain (L3) also describes all the previous languages.

The first step in the compilation of an L3 program is to obtain a model – an instance of the L3 metamodel. It is a representation of the L3 program in the metamodel-based repository. This step is a separate step in the whole process of the compilation which requires parsing of the text file and building a model. It is implemented using a traditional programming language (C++). Obtained lexemes [33, chapter 3] are stored in the repository as a very simple lexeme model [27]. Next, the transformation language L0 is used to obtain the L3 program model from the lexeme model.

When a program model has been built, the actual compilation is being performed. The L3 (also L2, L1, L0') compiler actually is a model transformation. In this case, an in-place transformation is used – the L3 program model is overwritten by the semantically equivalent L2 program model (also L2 by L1, etc.). The final result of the chain of compilation steps is an L0 program model which is semantically equivalent to the initial L3 program given as the input file. The chain of compilation steps (from L3 to L0) can be treated as one step (the corresponding transformations are invoked one after another).

The last step in the compilation process is the code generation (a model to text transformation). An L0 language text file is generated. This step is also carried out using the L0 language extended with native functions for file handling written in C++. Actually, only one write to file function is needed.

## 5.2 MOLA Compiler

Since the whole L3 compilation process has been divided into three separate steps, there is a possibility to start with any step if the appropriate model has been prepared. This fact is used by MOLA to L3 compiler – MOLA program is being compiled directly to an L3 model. This allows to decrease significantly the complexity of the implementation of MOLA to L3 compiler. Actually, it allows to use transformation language L3 to build MOLA to L3 compiler.

The first MOLA Transformation Definition Environment (MOLA Editor) [34] was built on the basis of Generic Modelling Framework [35] – a domain-specific modelling framework, developed by the UL IMCS together with the company *Exigen Services DATI*. The models (MOLA program and metamodel) were stored in a compatible format to the repository used by the L0 language. Thus, the input for the MOLA to L3 compiler, a model of a MOLA transformation, already could be obtained. In fact, no other natural representation of a MOLA program than a model can be obtained because MOLA is a graphical transformation language. The most appropriate way to implement MOLA compiler to any suitable language is by using model transformations. Thus, the first MOLA compiler was implemented using L3 language.

Since the MOLA Editor required more sophisticated features than the GMF domain specific modelling framework could offer, the next MOLA Editor – MOLA2 Tool – has been built. MOLA2 Tool uses the METAclipse framework [10], which is based on Eclipse platform [36] and model transformations. It should be noted that METAclipse uses the same repository as the L0 realization. Therefore it was possible to develop transformations for MOLA2 Tool using MOLA itself and the first MOLA compiler. The second version of MOLA to L3 compiler has been built for MOLA2 Tool, also using L3 language.

Although there are two implementations of MOLA to L3 compiler, there are no significant differences in the architecture and general ideas of the implementations of both compilers. The main difference between these two implementations is the MOLA metamodel. The MOLA metamodel for MOLA2 Tool was improved by eliminating metamodel restrictions enforced by GMF and making it more suitable for compilation. The experience and a significant part of the code from the first version of MOLA to L3 compiler is reused in the second version. This paper is based on the second version of MOLA to L3 compiler.

Compilation of a MOLA transformation is divided into four steps. Each of them is performed by a separate component – compiler. These components are:

- MOLA to L3 compiler,
- L3 to L0 compiler,
- L0 to C++,
- C++ to executable file.

The general architecture of MOLA compiler is shown in Fig. 6.

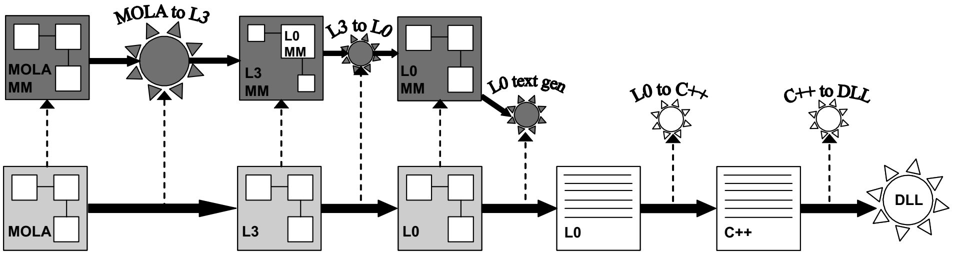


**Fig. 6.** The general architecture of MOLA compiler

A question may arise – why such a large number of compilers are used? Why do not use direct compilation from MOLA to C++? The answer is in the low complexity and reusability of each step. Each compiler transforms a higher-level language to a lower-level language. It is much easier to build compiler to a language that is at a closer abstraction level to the source language. Especially it is so if the general concepts of both languages are similar. This is the reason why L3 (and not L0) is used as the target language for MOLA. Another issue is the reusability. The compiler of L3 language was already built and this implementation was efficient. The efficiency of the generated code does not suffer if MOLA compiler is built on top of the compiler chain. In addition, if we will decide to implement MOLA on another EMOF compatible repository, for example, EMF [37] or Gralab [38], then only L0 compiler must be rewritten. Even less, only the actual code generator in L0 compiler must be rewritten – the lexical and syntax analyzers can be reused. The last compiler (L0 to code) is dependent on the programming language that implements the API of the model repository, but for most programming languages it is already built and free, or open-source versions are available. For example, there are free compilers for Java [39] and C++ [40]. The only disadvantage of a long compiler chain is longer compilation time, but it is not a significant problem in areas where transformation languages are used.

## 5.3 Model-Driven Compiling

The usage of models and transformation languages in the process of compilation is not new. The ATL model transformation language [16] has already been used to compile CPL to SPL [41] and FIACRE to LOTOS [42]. The ATL language itself is also compiled using a domain-specific language created only for this purpose – ACG (ATL Code Generation language) [43]. All of these are textual languages and the model-to-model transformation is used for actual compilation similarly to the way it was used in the example of the L3 to L0 compilation [27]. A similar idea is also used in the SmartQVT [15] implementation. The QVT code is parsed to obtain the model representation of a QVT transformation, and the actual compilation to the Java file is performed using this model.

A similar pattern of compilation is used in all examples. Three basic steps are performed:

- parse an input program and obtain the model of it,
- compile the model of the input program to the model of an output program,

- generate the code of the output program from the model.

This approach may be called **model-driven compiling** – models are used as core elements of the compilation process (see Fig. 7).

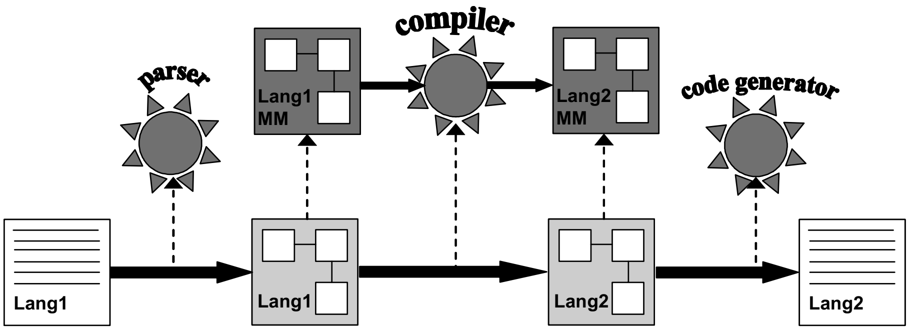


**Fig. 7.** Model-driven compiling – general architecture

These steps are similar to the phases of compilation in the traditional compilation technique [33, chapter 1]. The lexical and syntactical analyses are performed by the parser. The semantic analysis, intermediate code generation (target program model), and optimization are performed by compiler (model transformation). The code generation is done in the last step. The model of a source program is stored according to the language metamodel. Actually, the parse trees used in the traditional compilation technique can be treated as sort of models. Thus, the similarity is obvious.

All three steps of the model-driven compiling require appropriate metamodels already built for both input and output languages and transformation written using a model transformation language suitable for the compilation tasks. Actually, text-to-model (T2M), model-to-model (M2M), and model-to-text (M2T) languages are needed. An exporter or importer written in the general purpose programming language can be used instead of the T2M and M2T transformations. Certainly, the choice of the programming language depends on the repository used to store models.

The model-driven compiling is even more appropriate for graphical languages such as MOLA. Since programs of graphical languages are stored as models, the first step can be omitted – the model-to-model transformation that implements a compiler can be applied directly.

The main advantages of using model-driven compiling:

- The higher level of abstraction that is provided by model transformation languages allows reducing the complexity of compiler implementation.
- This is the most appropriate way to compile graphical languages because they are mostly implemented using some metamodel [37] or graph-based [38] repository. Actually, programs (diagrams) of such languages are models and the usage of a model transformation language is the most natural approach.

- If the concrete syntax of the input language is based on some general "coding" language, like XML [44], then model transformations can be applied to obtain the model of the program from its "coding". In this case, a standard parser can be used to obtain the model of the "coding". Next, the model transformation can be used to obtain the model conforming to the input language metamodel. A similar approach is also applicable for the output language.
- Since attribute grammars have been used to specify the semantics of programming languages [45], a precise definition of the model transformation between source language and target languages can be used to define the semantics of the source language in even more readable way.

The first experience in using **model-driven compiling** was quite promising. The MOLA to L3 and L3 to L0 [27] compilers have been developed. The implementation of both compilers has shown that using transformation language for compilation tasks reduces the complexity of the implementation. However, the best practice of model-driven compiling has yet to be developed, and comparison to the traditional compilation techniques [33] must be drawn.

## 6 Mapping from MOLA to L3

This chapter contains detailed description of the mapping from MOLA to L3. That includes mapping of metamodeling language constructs and mapping of MOLA procedure and its elements to constructs of the L3 language.

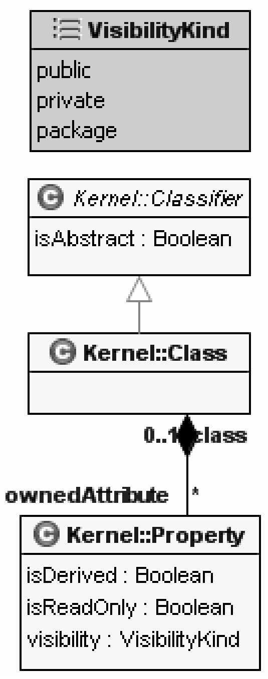
### 6.1 Mapping of Metamodelling Languages

Both MOLA metamodelling language and the Lx family metamodelling language are based on EMOF. So the mapping is straightforward. For the description of this mapping, we will use the meta-class names from MOLA and Lx family metamodelling language metamodels shown in Fig. 2 and Fig. 4. The MOLA related meta-class names are prefixed by the *Kernel* prefix, but the Lx related meta-class names are prefixed by the *Lx* prefix.

- Each *Kernel::Class* instance is transformed to *Lx::Class* with the same name, but since there are no packages in Lx, the *Lx::Class* name is prefixed by all parent package names. For example, the *Kernel::Class* "Lifeline", which is owned by the package "Interactions", which is in package "UML", is transformed to *Lx::Class* named "UML::Interactions::Lifeline"
- Both languages have pre-defined primitive types. All the primitive types that are in MOLA – *String*, *Integer, Boolean* – are also in Lx.
- Each *Kernel::Enumeration* instance is transformed to *Lx::Enumeration* instance and each *Kernel::EnumerationLiteral* instance is transformed to *Lx::EnumerationLiteral* instance owned by the appropriate enumeration.
- Each *Kernel::Generalization* instance is transformed to *Lx::Generalization* instance. Of course, *general* and *specific* links are set to the appropriate classes. This implementation of the L0 does not allow multiple generalization; thus, it cannot be used in MOLA either.

- Each *Kernel::Association* instance is transformed to *Lx::Association*, and appropriate association ends that are represented as *Kernel::Property* instances linked by *memberEnd* link to the association are transformed to *Lx::AssociationEnd* instances. They are linked to the appropriate class instances. Multiplicity, ordering and composition information of association ends are also transformed directly to Lx.
- Each *Kernel::Property* instance that is an attribute is transformed to an *Lx::Attribute* instance. Since MOLA allows only primitive or enumeration-typed attributes, the correspondence is direct.

An example of the transformation is given in Fig. 8.

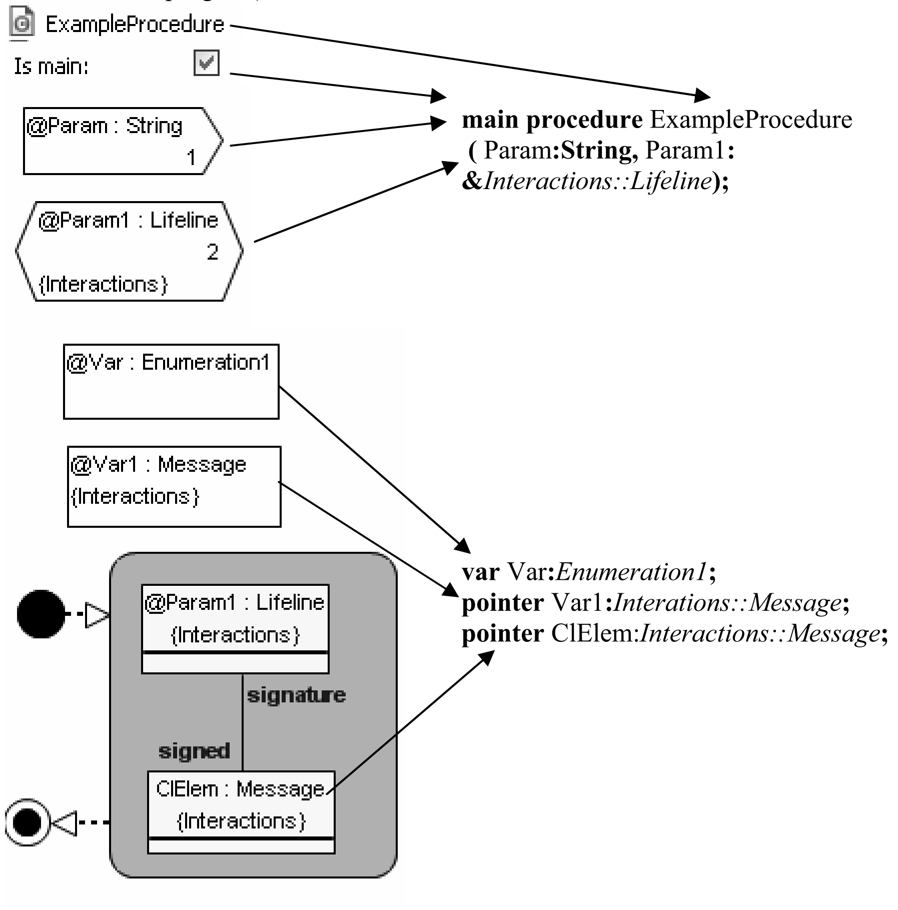


```
class Kernel::Classifier;
class Kernel::Class;
class Kernel::Property;
enum VisibilityKind : {public,private,package};
 compos Kernel::Class.[0..1]class/
     ownedAttribute[*].Kernel::Property;
attr Kernel::Classifier.isAbstract:Boolean;
attr Kernel::Property.isDerived:Boolean;
attr Kernel::Property.isReadOnly:Boolean;
attr Kernel::
     Property.AggregationKind:AggregationKind;
attr Kernel::Property.VisibilityKind:VisibilityKind;
rel Kernel::Class.subClassOf.Kernel::Classifier;
```

**Fig. 8.** An example of MOLA to Lx metamodelling language

## 6.2 Mapping of the Procedure Headers

MOLA procedures form the executable part of a MOLA transformation. The L3 language also has procedures. Both MOLA and L3 procedures may have parameters that may be *in* (passed by value) or *in-out* (passed by reference). Both languages may have variables declared. In L3, the class-typed variables and parameters are called *pointers* and have a different syntax, so compiler must distinguish class-typed variables from enumeration and primitive-typed variables. Each non-reference class-element that is used in rules in a MOLA procedure is transformed to a pointer declaration. Actually, the transformation of procedure header is straightforward and does not need detailed description. An example of the transformation of a MOLA procedure header is shown in Fig. 9 (the L3 code in all examples is used to better illustrate the result of compilation. Actually, the compiler produces instances of the model of an L3 program)



**main procedure** ExampleProcedure
**(** Param**:String,** Param1**:**
**&***Interactions::Lifeline***);**

**var** Var:*Enumeration1***;**
**pointer** Var1**:***Interations::Message***;**
**pointer** ClElem:*Interactions::Message***;**

**Fig. 9.** Procedure header to L3

## 6.3 Mapping of the Execution Control Flows

The basic statements of MOLA are rule and for-each loop. There also are other MOLA statements – text-statement, call-statement, etc. Control flows are used to determine the order of execution of MOLA statements within one MOLA procedure.

There is exactly one start-statement in a MOLA procedure. It defines the entry point of the MOLA procedure. Other statements may pass the execution control to another statement or terminate the execution of the procedure. End-statements are used to terminate the execution of the procedure. They define the exit points of the MOLA procedure. The execution of the procedure may also be terminated by a text-statement or a rule if the corresponding control flow is not present. Actually, a text-statement and a rule are used as traditional branching constructs (they may have two outgoing control flows, one of them labelled *ELSE*). A for-each loop contains nested MOLA statements (loop-body) that are executed during each iteration. It has a special statement – loop header (rule-based loophead), which defines the entry point to the loop-body. There may be any other MOLA statement in the loop (except start-statement) – nested loops are also allowed. A statement that has no outgoing control flow terminates the current iteration of the loop. A branching statement may also terminate the current iteration of the loop if one of outgoing control flows is not present. Other statements (call-statement, etc) just pass the execution control to the next statement. Control flows in MOLA procedure may connect statements in an almost arbitrary way, there are only few restrictions. Incoming control flows are not allowed to the start-statement and loophead. Outgoing control flows are not allowed from end-statements. It is not allowed to "jump" into a loop from an outside statement either (it is allowed to "jump" out).

Control flows and MOLA statements form a directed graph, where some nodes (loops) may contain a nested graph. This graph is the control flow graph (CFG) of a MOLA procedure. The control flow graph is a data structure used by traditional compilers for analysis and optimization of program execution [33, chapter 10].

The most natural way to code the control flow graph in a textual language is to use a labelled block of code for every node and a "jump" command for every edge. Thus each node of the MOLA control flow graph will compile to the block of L3 code. The block of code must start with a **label** command that unambiguously identifies the block. The execution control is passed to another code block using a **goto** command. If the execution of the MOLA procedure must be terminated, then a **return** command is used.

According to the different types of statements described above, we can distinguish five types of nodes in the control flow graph of the MOLA procedure and define the mapping to L3 language for these types:

- Entry node (start-statement) is a unique and mandatory node. Here we do a little optimization – no L3 code block is created for start-statement. The outgoing control flow determines the first MOLA statement that in turn determines the first code block of the procedure.
- Exit node (end-element) is compiled to the following code block (in what follows, a simple template language is used – L3 keywords are bolded, other parts of code are shown in angular braces containing an intuitive description):

  **label** <label name>**;**
  **return;**
- Simple node (call-statement) may not have an outgoing *ELSE* control flow. It is compiled to a simple code block – a sequence of commands depending on the actual type of MOLA statement and the **goto** command to the **label** command of the code block that is created from the MOLA statement connected by the outgoing control flow.

  **label** <label name>**;**
  <sequence of commands>**;**
  **goto** <next label name>**;**
- Branching node (rule, text-statement) may have two outgoing control flows, where one of them may be an *ELSE* control flow. It is compiled to an **if-then-else** command. The *if-block* contains the condition, *then-block* contains the action part of the MOLA rule or text-statement and *else-block* contains a

**goto** command to the **label** command of the code block that is created from the MOLA statement connected by the outgoing *ELSE* control flow. The last command in the main code block is the **goto** command to the **label** command of the code block that is created from the MOLA statement connected by the other (*non-ELSE*) outgoing control flow.

```
label <label name>;
if
        begin
                <condition commands>;
        end
then
        begin
                <action commands>;
        end
else
        begin
                goto <next else label name>;
        end;
goto <next label name>;
```

- Loop node (for-each loop) contains a nested control flow graph. Since a loop and its loophead can not be used separately, a common L3 code block is created for both nodes. A loop is compiled to a **foreach** command. The *such-that* block contains the condition, the *do* block contains the action part of the loophead. The *do* block also contains a **goto** command to the **label** command of the code block that is created from the MOLA statement connected by the outgoing from the loophead control flow. The last command in the *do* block is a **label** command. This label is used to receive back the execution control from the code blocks that terminate an iteration of the loop. Thus, a MOLA statement which terminates the execution of the current iteration of the loop passes the execution control to this **label** command instead of terminating the execution of the whole procedure. In fact, the execution control is passed away from the *do* block of a **foreach** command, but it is received back just at the end of an iteration. Thus, the code blocks that are created from MOLA statements within the loop body are included in the corresponding L3 loop body indirectly – using **goto** and **label** commands. The last command in the main code block is a **goto** command to the **label** command of the code block that is created from the MOLA statement connected by the outgoing control flow of the loop.

```
label <label name>;
foreach <loop variable name> suchthat
        begin
                <loophead condition commands>;
        end
do
        begin
                label <loophead label name>;
                <loophead action commands>;
                goto <loophead next label name>;
                label <loop iteration end label name>;
        end
goto <next label name>;
```

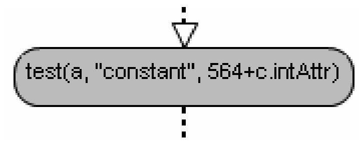
The complete code of the procedure is assembled using code blocks obtained in the way just described. The first code block is determined by the start-statement. All other code blocks may be added to the procedure in an arbitrary order because the order of execution is determined only by **label** and **goto** commands – not by the order in which command blocks are added to the procedure.
The result will be likely a sort of "spaghetti code" [46], but this causes no danger because the L3 code is just an intermediate code which is compiled further. This code is not read by a transformation developer. The wide usage of the **goto** commands does not cause any loss in the overall performance.

## 6.4 Mapping of MOLA Statements

The control structure aspect of the mapping of MOLA statements to L3 commands has already been described in the previous section. This section contains a detailed description of the mapping for each MOLA statement including data processing and pattern matching aspects.
The mapping for start and end statements has already been described. The start-statement is used to determine the first MOLA statement and end-statement is transformed to the **return** command.

### 6.4.1 Call-Statement

The **call-statement** is transformed to the **call** command. Since the mapping from a MOLA procedure to L3 procedure is one-to-one, the called L3 procedure is the same that is mapped from the MOLA procedure called by the MOLA call-statement. The L3 language allows only binary expressions to be used as actual parameters of the **call** command. MOLA allows arbitrary expressions (of appropriate type) to be used as actual parameters (the same problem is for calling functions in an expression). Our solution is to use temporary variables or pointers (depending on the actual type of a parameter) and **setVar** or **setPointer** commands to calculate the values of expressions. These commands must be executed before the **call** command. If the actual parameter is a MOLA variable, parameter, or class element identifier, then a temporary variable is not used. An example of the compilation is shown in Fig. 10.



**var** temp_var1:*String*;
**var** temp_var2:*Integer*;
**begin**
**...**
**label** id_lab1;
**setVar** temp_var1="constant";
**setVar** temp_var2=**564**+c.intAttr:*Integer*;
**call** test(a,temp_var1,temp_var2);
**goto** id_labx;
**...**

**Fig. 10.** The compilation of the call-statement

### 6.4.2 Text-Statement

As it was described before, the **text-statement** is transformed to the **if-then-else** command. MOLA text-statement has two main parts – a condition (constraint), which is expressed using OCL-style expression, and a list of assignments. The condition holds if the expression evaluates to *true*. The condition is compiled to the *if* block of the **if-then-else** command. Assignments are compiled to the *then* block of the **if-then-else** command.

Assignments are used in the text statement to assign values to elementary variables and pointers. The L3 commands that are used for this task are **setVar** and **setPointer**. In MOLA the value that is being assigned is expressed using a *simple expression* of an appropriate type. A simple expression of *Integer* type may contain *Integer-typed* variable, parameter or attribute specifications, *Integer* constants, pre-defined functions (*size, indexOf, toInteger*) and arithmetic operations (addition, subtraction, multiplication). A simple expression of *String* type may contain *String-typed* variable, parameter or attribute specifications, *String* constants, pre-defined functions (*toLower, toUpper, substring, toString*), and a concatenation operation. A simple expression of *Boolean* type may contain *Boolean-typed* variable, parameter or attribute specifications, *Boolean* constants (*true* and *false*), or pre-defined function (*isTypeOf, isKindOf, toBoolean*). A simple expression of *enumeration* type may contain *enumeration-typed* variable, parameter or attribute specification, *enumeration* literals or a pre-defined function *toEnum*. A simple expression of *class* type may contain a *class-typed* variable or parameter specification (pointer), *null* constant or typecast.

In L3 similar expressions are allowed, but there are a few differences: there is no direct typecast of a pointer, actual parameters in a function call may be only a binary expression of an appropriate type. The list of pre-defined functions in L3 does not match all the pre-defined functions of the MOLA language either. The solutions to these problems are rather simple. In addition, some kinds of expressions in L3 allow more features than in MOLA, but these features are not relevant for MOLA compiler. The complete table of correspondence is shown in Table 1.

**Table 1.** Correspondence of elements used in expressions in MOLA and L3

| **MOLA** | **L3** |
|---|---|
| *String, Integer, Boolean,* enumeration-typed constants, NULL constant | + |
| elementary variables, pointers | + |
| attribute specification | + |
| +,-,*,concatenation | + |
| direct typecast (class-typed) | temporary variable and extra **setPointer** command used |
| function call | temporary variables and extra **setVar** commands for complex parameters used |

| pre-defined functions | extended library of native functions used |
|---|---|
| toEnum, toInteger, toString, toBoolean | + |
| indexOf, toLower, toUpper | extended library used |
| size, substring | + |
| isTypeOf, isKindOf | temporary variable and **type** command used |

The left column describes features used in MOLA expressions and the right column shows the correspondence in L3. The plus sign (+) means that the mapping is direct. If there is no direct mapping, the basic principles of a solution are shown. It may be the usage of a temporary variable (typecast and function call) or the usage of an extended library of native functions (*indexOf, toLower, toUpper* functions).

Though L3 expressions allow Boolean operations, they cannot be used with relations. Relational operators (<, >, etc) may be used only in **var** and **pointer** commands. That makes the compilation of *Boolean* expressions used in MOLA more difficult.

In MOLA the simplest condition is a simple expression of the *Boolean* type. Then it is compiled using a temporary variable and a **var** command in the following way:

| **Condition:** | **if** |
|---|---|
| | **begin** |
| <simple boolean expression> | [<extra commands>] |
| | **setVar** temp_var=<simple boolean expression>**;** |
| | **var** temp_var**==true**; |
| | **end…** |

Usually a condition also contains a relation (>, <, >=, <=, =, <> operators can be used). Since the left and the right operands may be arbitrary expressions of the same type, the value of each expression is computed and stored in a temporary variable. Then these variables are compared using a **var** or **pointer** command depending on the type of expressions.

| **Condition:** | **if** |
|---|---|
| | **begin** |
| <expression1><relation> <expression2> | [<extra commands>] |
| | **setVar/setPointer** temp_var1=<expression1>**;** |
| | [<extra commands>] |
| | **setVar/setPointer** temp_var2=<expression2>**;** |
| | **var/pointer** temp_var1<relation>temp_var2**;** |
| | **end** |
| | **...** |

A condition in MOLA may also contain Boolean operations – conjunction (**and**), disjunction (**or**), and negation (**not**) – together with relational operators. The L3 has no such features, but it is shown [18, chapter 4] that it is possible to construct L3 code that implements the Boolean operations. The algorithm implemented in MOLA to L3 compiler uses the same principles.

Our template language will be used to explain this algorithm. An extension of the template language is required – let us define a function

*PrintBooleanExpression(variable_name,boolexpression)* that returns **the block of L3 code** that calculates the value of the Boolean expression *boolexpression* and stores it in the variable whose name is passed by the parameter *variable_name*. The use of this function means that the code block returned by the function replaces the function call. We will also need an auxiliary procedure *CreateBooleanVariable(varname)*, which adds the declaration of a new *Boolean* variable whose name is passed by the parameter *varname*. Variable and label names having a prefix *unique* are considered to be unique within the procedure.

If the parameter *boolexpression* is a simple expression of type *Boolean* or a relation, then the function *PrintBooleanExpression* will return the following code:

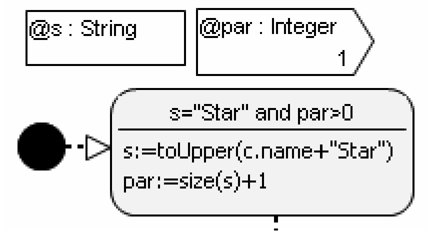
| | |
|---|---|
| *boolexpression*=<simple boolean expression> | [<extra commands>]<br>**setVar** *variable_name* =<br>    <simple boolean expression>**;** |
| *boolexpression*=<br>*<expression1><relation>*<br>*<expression2>* | **setVar** *variable_name* =**false**;<br>[<extra commands>]<br>**setVar** unique_temp_var1=<expression1>;<br>[<extra commands>]<br>**setVar** unique_temp_var2=<expression2>;<br>**var** unique_temp_var1<relation><br>unique_temp_var2 **else** unique_label;<br>**setVar** *variable_name* =**true**;<br>**label** unique_label; |

If the parameter *boolexpression* contains Boolean operators **and, or, not,** then the function will return the following code

| | |
|---|---|
| *boolexpression*=<br>*boolexpression*1 **or**<br>*boolexpression2* | *CreateBooleanVariable ("unique_temp_var1")*<br>*CreateBooleanVariable ("unique_temp_var2")*<br>*PrintBooleanExpression("unique_temp_var1",*<br>    *boolexpression*1*)*<br>*PrintBooleanExpression("unique_temp_var2",*<br>    *boolexpression2)*<br>**setVar** *variable_name*=**true*;***<br>**var** *unique_temp_var1*==**false else** unique_label**;**<br>**var** *unique_temp_var2*==**false else** unique_label;<br>**setVar** *variable_name*=**false*;***<br>**label** unique_label**;** |
| *boolexpression*=<br>*boolexpression*1 **and**<br>*boolexpression2* | *CreateBooleanVariable ("unique_temp_var1")*<br>*CreateBooleanVariable ("unique_temp_var2")*<br>*PrintBooleanExpression("unique_temp_var1",*<br>    *boolexpression*1*)*<br>*PrintBooleanExpression("unique_temp_var2",*<br>    *boolexpression2)*<br>**setVar** *variable_name*=**false*;***<br>**var** *unique_temp_var1*==**true else** unique_label;<br>**var** *unique_temp_var2*==**true else** unique_label**;**<br>**setVar** *variable_name*=**true*;***<br>**label** unique_label**;** |

| | |
|---|---|
| *boolexpression=* **not** *boolexpression1* | *CreateBooleanVariable (“unique_temp_var1”)* *PrintBooleanExpression(“unique_temp_var1”, boolexpression1)* **setVar** *variable_name*=**true***;* **var** *unique_temp_var1*==true **else** unique_label; **setVar** *variable_name*=**false**; **label** unique_label; |

An example of the compilation of a MOLA text-statement is shown in picture Fig. 11.



```
s="Star" and par>0
s:=toUpper(c.name+"Star")
par:=size(s)+1
```

```
if begin
   setVar _mvar_6=false;
   setVar _mvar_9=s;
   setVar _mvar_10="Star";
   var _mvar_9==_mvar_10 else
   _mlabel_8;
   setVar _mvar_6=true;
   label _mlabel_8;
   setVar _mvar_7=false;
   setVar _mvar_12=par;
   setVar _mvar_13=0;
   var _mvar_12>_mvar_13 else
   _mlabel_11;
   setVar _mvar_7=true;
   label _mlabel_11;
   setVar _mvar_4=false;
   var _mvar_6==true else _mlabel_5;
   var _mvar_7==true else _mlabel_5;
   setVar _mvar_4=true;
   label _mlabel_5;
   var _mvar_4==true;
end then begin
   setVar _mvar_14=
   c.name:String+"Star";
   setVar s= toUpper(_mvar_14);
   setVar par= Length(s)+1;
end else begin
   return;
end;
```

**Fig. 11.** The compilation of the text-statement

### 6.4.3 Rule

Another, and the most important, decision statement in MOLA is a **rule**. It is also compiled to the **if-then-else** command. The condition of the rule is expressed using a pattern. The implementation of pattern matching typically is the most demanding component to implement and also the key factor determining the implementation efficiency. The efficiency of the implementation of the pattern matching is not the central theme of this paper. The chosen realization of the pattern matching implements some ideas that have been already described in [28]. This approach guarantees sufficient efficiency of the pattern matching for typical MOLA use cases.

The basic elements of the pattern are class-elements and association-links. A class-element represents the instance of the particular class. There are several types of class-elements, but only *normal* and *delete* class-elements are used to specify a pattern. Let us call them *pattern elements*. In addition, only *normal* and *delete* association-links are used to specify a pattern. Let us call them *pattern links*. Pattern elements and pattern links form the *pattern graph*. Pattern elements that are linked by pattern links form the *pattern fragment* (connected component of the pattern graph). A pattern may contain several pattern fragments that can be treated as separate patterns. All pattern fragments must match for the whole pattern to match. The main goal of the pattern matching is to find particular instances that match the given pattern. The sought instances are represented by non-reference pattern elements. The pattern links, reference class elements, and constraints on class elements form the *pattern constraint*. Actually, *such* a set of instances is sought *that* matches the pattern constraint.

The pattern is compiled to a block of L3 code which is placed in the *if* block of the **if-then-else** command. Several pattern fragments are compiled to separate L3 code blocks following each other. Natural constructs in L3 language that implement patterns are **first-suchthat** and **first-from-by-suchthat** commands. A pattern fragment is thus compiled to a nested **first-suchthat** or **first-from-by-suchthat** command.

To achieve this goal, the pattern graph must be traversed and appropriate commands built. The classical graph traversing techniques are used – a recursive algorithm that marks already traversed nodes and edges [47].

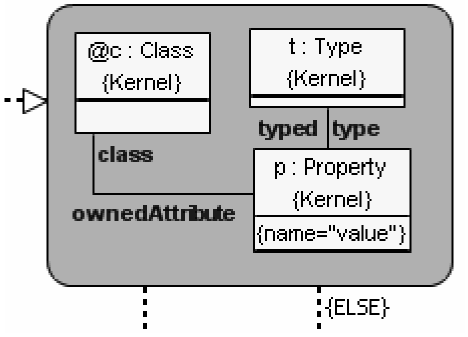
The first task is to decide which pattern element will be processed first – let us call it a *root node*. This is an important task because this decision affects the overall performance of the pattern matching. The main idea is to reduce the number of instances that must be examined to match or fail the pattern. If the pattern fragment contains a reference element, then the traversing of the pattern graph must be started from this element. This version of MOLA language also allows to denote the root element manually, using special compiler-related annotations.

The algorithm starts the processing of the graph with the root node:

- *root node* – is marked as *traversed.*
    - o If it is a non-referenced class-element, then the **first-suchthat** command is created. The *such-that* command block of the command is selected as the *current command block*. L3 commands that are obtained from the local constraint of the class-element are placed in the *such-that* block of the created command.
    - o If it is a referenced class element, then L3 commands that are obtained from the local constraint of the class element are placed in the *if* block of the **if-then-else** command.
    - o All nodes connected by adjacent edges (pattern links that have not yet been traversed) are processed.
- Other (non-root) *nodes* are processed in the following way – the edge which is used to reach this node is marked as *traversed.*

- If the node has been already traversed, then a **link** command is added to the *current command block*.
- If the node has not been traversed, then it is marked as *traversed*.
    - If it is a reference class-element, then a **link** command is added to the *current command block*. L3 commands that are obtained from the local constraint of the class element are placed in the *if* block of the **if-then-else** command.
    - If it is a non-reference class-element, then the **first-from-by-suchthat** command is added to the *current command block*. The *such-that* command block of the this command is selected as the *current command block*. L3 commands that are obtained from the local constraint of the class element are placed in the *such-that* block of the created command.
    - All nodes connected by adjacent edges that have not yet been *traversed* are processed.

The local constraints of pattern elements are processed in the same way as the condition of the text-statement.
An example of the compilation of a pattern is given in Fig. 12.



```
if begin
    first p:Kernel::Property from c
    by ownedAttribute suchthat
    begin
      setVar _mvar3=p.name:String;
      setVar _mvar4="value";
      var _mvar3==_mvar4;
      first t:Kernel::Type from p by
      type;
    end;
end
then
....
```

**Fig. 12.** The compilation of the rule-pattern

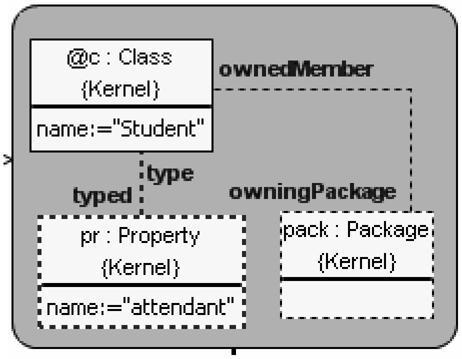
Actually, the algorithm described above realizes the principles of MOLA Virtual Machine described in [28]. This algorithm builds an efficient L3 code if MOLA language constructs are used in a natural way. The practical usage of MOLA compiler has also shown that the natural use of MOLA constructs leads to an efficient pattern matching. Thus, the current implementation is sufficient enough for typical tasks (MDA, tool building). However, the algorithm can be enhanced in order to achieve a better performance in less typical situations. For example, if the pattern does not contain a reference pattern-element or annotated pattern-element, then a more detailed analysis of the pattern graph should be performed. The multiplicities of the associations that correspond to the association-links used in the pattern could be analyzed. The direction of traversing the graph should be chosen so that the "going" along an association in the direction of '*' multiplicity is minimized. More complicated algorithms for the pattern matching have been used typically in rule-based transformation languages, for example, VIATRA [48]. This problem (the pattern matching efficiency) is not the main topic of this paper; therefore, it is not discussed in-depth.

The action part of a rule consists of class-elements, association links, and attribute assignments that are included in class elements. The *create* and *delete* class-elements are used to create and delete particular instances. The *create* and *delete* association-links are used to create and delete links. The assignment is used to assign the value of the attribute of a particular instance. The value is specified by using expressions that have been already described in previous sections. The correspondence between MOLA and L3 constructs is shown in Table 2.

**Table 2.** Correspondence of constructions used in the action part of the rule

| MOLA | L3 |
|---|---|
| *create, delete* class-elements | **addObj**, **deleteObj** commands |
| *create*, *delete* association-links | **addLink**, **deleteLink** commands |
| attribute value assignments | **setAttr** commands |

The L3 code that is created for the action part of the rule is placed in the *then* block of the **if-then-else** command. An example of the compilation of the action part of a rule is shown in Fig. 13.



```
If begin …end
then begin
    addObj pr:Kernel::Property;
    addLink pr.type.c;
    setAttr c.name="Student";
    setAttr pr.name="attendant";
    deleteLink
    c.owningPackage.pack;
    deleteObj pack;
end else
…
```

**Fig. 13.** The compilation of the rule – action part

### 6.4.4 For-each loop

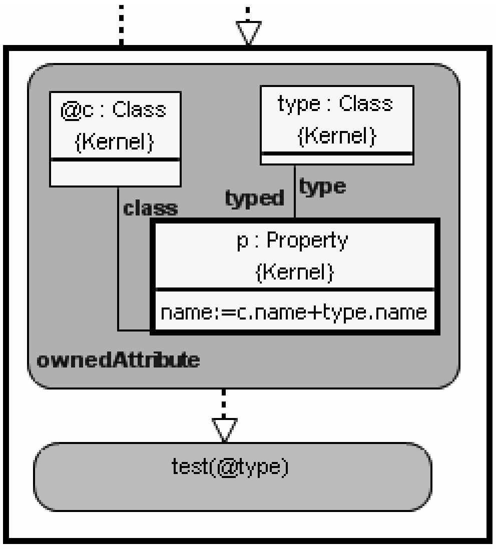
The last MOLA statement described in this chapter is the **for-each loop**. The implementation of a loop is one of the crucial issues in the implementation of the MOLA compiler. An incorrectly chosen search structure may cause serious efficiency problems.

The condition of a loop is expressed by using the pattern of the loophead, which contains a special class-element – the *loop variable*. The iteration is performed over all instances that correspond to the loop variable.

The loop is compiled to the **foreach** command. The condition of the loop is compiled to the *such-that* block of the **foreach** command. The compilation of the loophead pattern is similar to the compilation of the rule pattern. The pattern match starts from the loop variable (it is chosen as the *root node*). Usually there is a *restriction-path* – a path from a referenced class element to the loop variable where all multiplicities of the corresponding associations are **'0..1'** or **'1'.** Then for this path, **first-from-suchthat** commands are created and added to the code block before the **foreach** command. The loop variable is used as the loop variable in the **foreach** command. All nodes and edges that have been already processed (appropriate commands built for the loop variable and class-elements in the restriction path) are marked *traversed*, and the algorithm used for the compilation of a rule is executed.

This algorithm is not the most optimal either, but it is suitable for most of typical examples – usually there is a restriction path. Further optimization of the algorithm is not addressed in this paper.

The action part of the loophead is compiled in the same way as the action part of a rule. The created code is added to the *do* block of the **foreach** command. Fig. 14 illustrates an example of the compilation of a loop.



```
foreach p:Kernel::Property from c
by ownedAttribute suchthat
begin
    first type:Class from p by
    type;
end do begin
    setAttr p.name=c.name:String
    + type.name:String;
    goto _mlabel_10;
    label _mlabel_9;
end;
goto _mlabel_23;
...
label _mlabel_10;
call test(type);
goto _mlabel_9;
...
```

**Fig. 14.** The compilation of the loop

The mapping of the most important MOLA constructs to L3 has been defined in this chapter.

# 7 The surrounding of the MOLA compiler

This chapter introduces the problems that have been discovered during the implementation of the MOLA compiler. The compiler is the most important part of the implementation of a programming or transformation language. However, there are other parts needed in a proper development environment.

## 7.1 Error handling in MOLA

The compiler detects syntax errors in a program. Usually a development environment of a textual programming language provides the possibility to navigate to errors in a code. A list of errors is shown and the appropriate "problematic" line of code is highlighted. Similar requirements can also be applied to the MOLA development environment. Since MOLA is a graphical language, there are no "lines of code", as it is in textual languages. Each element that has a visual representation (MOLA statement, class-element, etc) can be treated as a "line of code". The MOLA compiler must detect errors in a program and point to the appropriate element. Actually, MOLA compiler does not "know" anything about the visual representation of a MOLA element. Thus, the visualization of an error is done by the development environment.

Our solution is to store the error information in the *error model*. The *error metamodel* is very simple (see Fig. 15).

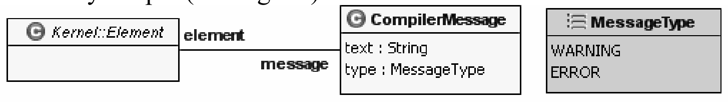
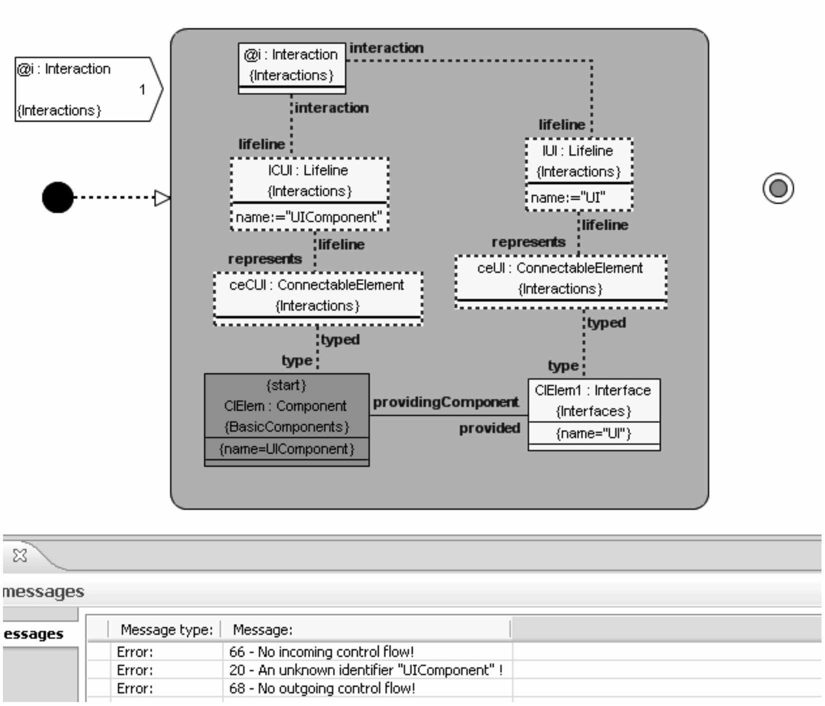


**Fig. 15.** The error metamodel

In fact, there is only one class (*ErrorMessage*). It represents a particular error. There are two attributes – the attribute *text* contains the textual information and *type* determines whether it is a warning or an error. The association *element* represents an "error pointer" to the appropriate element in a MOLA transformation (any MOLA element inherits from the *Kernel::Element*, see Fig. 3). The MOLA compiler deletes the existing error model and creates a new one in the process of compilation. The MOLA2 Tool reads the error model and visualizes it. An example of the error visualization is shown in Fig. 16.
The list of errors is shown in the properties tab. It is possible to navigate to the corresponding MOLA procedure from there. The elements pointed by the compiler are highlighted. This is an adequate way to treat the error handling problem in a graphical language.



```
Message type:   Message:
Error:          66 - No incoming control flow!
Error:          20 - An unknown identifier "UIComponent" !
Error:          68 - No outgoing control flow!
```

**Fig. 16.** The visualization of errors in a MOLA procedure.

## 7.2 Structuring a program in MOLA

Another feature provided by modern development environments is the possibility to compile only part of the code if the whole program has already been compiled. This is needed for large programs, when a compilation takes a significant amount of time. To achieve this goal, the program has to be structured. The most common approach is to use code units. Each unit is compiled to a separate object. Next, a linker is used to obtain a single executable.

A similar idea is also used in the MOLA2 Tool. Packages are used to structure a MOLA program. A package may be defined as a *MOLA unit*. It means that all MOLA procedures that are contained by the unit are compiled to a separate L0 unit. This allows using L0 compiler as a linker that assembles all L0 units into one C++ project. Thus, model transformations (MOLA and L3-L0'compilers) can work with smaller models that helps to improve the overall performance of the compilation process.

## 7.3 Debugging in MOLA

If a program is successfully compiled, it means that it is syntactically correct, but it does not mean that the program is semantically correct. Testing is a common approach used by a program developer. If a bug is found, then it must be fixed. This process is called *debugging*. The debugging process requires a tool support to ease this process. Tools used for debugging are called *debuggers*.

Typically, debuggers offer functions such as running a program step by step and pausing the program to examine the current state of the program to track the values of some variables. Some debuggers have the ability to modify the state of the program while it is running. The importance of a good debugger is very high. The existence of such a tool can often be the deciding factor in the use of a language, even if another language is more suited to the task.

However, a debugger for the MOLA2 Tool has not yet been developed. There are examples of a debugger of a graphical language, for example, the UML Model Debugger [49]. There are differences between the debugger of a textual language and the debugger of a graphical language. The main difference is in the representation of the single-stepping approach. Since graphical languages are usually represented in diagrams, an animation of the program execution is required. Other representations could also be used, but they would be rather far from the concepts of the language.

An interpreter or instrumentation by an additional code in the compilation result may be used for the debugging purposes. The execution of a single MOLA statement could be considered as one step in the step-by-step debugging process. The result of the compilation of a MOLA program is L3 code. Since this code consists of code blocks that correspond to one MOLA statement, these blocks could be supplemented with a debugging code in a rather simple way.

There is another widely used but not so sophisticated way of the debugging. The trace (log) files can be used to trace the execution of a program. The current version of the MOLA compiler uses the L0 debugging feature – the L0 trace file. It logs an execution of every L0 command. However, the L0 tracing operates with L0 concepts. Therefore, a tracing that is at a closer abstraction level to the MOLA is needed.

## 8 Conclusions and Future Work

A sufficiently efficient implementation of the MOLA to L3 compiler has been described in this paper. The MOLA compiler has already been used practically in the area of tool building. The transformations that are used for implementation of the MOLA2 Tool within the METAclipse framework are developed using the MOLA to L3 compiler. The MOLA2 Tool that includes the second version of the MOLA compiler is successfully being used in the European IST 6th framework project ReDSeeDS [50]. Traditional MDA tasks are being implemented in MOLA there. These tasks include transformations from formalized software requirements to an architecture model of the system to be built and then to a detailed design model. Thus, the efficiency of the chosen architecture has been approved by practical usage. In both cases, non-trivial MOLA transformations have been developed and applied to sufficiently large models.

On the one hand, the future work is related to the problems discussed in chapter 7. The practical usage of MOLA has shown that the problem of debugging is quite significant. It should be noted that building both a user-friendly and sufficiently high-level debugger for model transformation languages, especially for graphical ones, is quite a challenging task. On the other hand, improvements in the implementation of the MOLA compiler are also expected – a more advanced algorithm of pattern matching for MOLA will be developed. These improvements should ensure more efficient execution for less typical MOLA transformations. In addition, the model-driven compiling briefly sketched in this paper also deserves a more detailed research.

## References

1. Volter M. and Stahl T., Model-Driven Software Development. John Wiley & Sons, 2006.
2. A.G. Kleppe, J.B. Warmer, & W. Bast, MDA explained: The model driven architecture: Practice and promise (Boston: Addison-Wesley, 2003)
3. The Object Management Group (OMG) URL: http://www.omg.org/
4. OMG Model-Driven Architecture URL: http://www.omg.org/mda/
5. Meta Object Facility (MOF) 2.0 Core Specification URL: http://www.omg.org/docs/ptc/04-10-15.pdf
6. OCL 2.0 Specification Version 2.0 URL: http://www.omg.org/docs/ptc/05-06-06.pdf
7. OMG Unified Modelling Language (UML), version 2.1.1 URL: http://www.omg.org/technology/documents/formal/uml.htm
8. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification URL: http://www.omg.org/docs/ptc/07-07-07.pdf
9. Metamodel and UML Profile for Java and EJB Specification URL: http://www.omg.org/docs/formal/04-02-02.pdf
10. Kalnins, A., Vilitis, O., Celms, E., Kalnina, E., Sostaks, A., Barzdins, J.: Building Tools by Model Transformations in Eclipse. Proceedings of DSM'07 workshop of OOPSLA 2007, Montreal, Canada, Jyvaskyla University Printing House, 2007, pp. 194–207.
11. I. Rath, D. Varro. Challenges for advanced domain-specific modelling frameworks. Proc. of Workshop on Domain-Specific Program Development (DSPD), ECOOP 2006, France.
12. Ermel, C., Ehrig, K., Taentzer, G., Weiss, E.: Object Oriented and Rule-based Design of Visual Languages Using Tiger. Proceedings of GraBaTs'06, 2006, pp. 12
13. Request for Proposal: MOF 2.0 Query / Views / Transformations RFP URL: http://www.omg.org/docs/ad/02-04-10.pdf
14. ikv++ - mediniQVT URL: http://www.ikv.de/index.php?option=com_content&task=view&id=75&Itemid=77
15. SmartQVT URL: http://smartqvt.elibel.tm.fr/index.html
16. ATL. URL: http://www.eclipse.org/m2m/atl/
17. VIATRA2 URL: http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/VIATRA2/index.html
18. J. Barzdins, A. Kalnins, E. Rencis, S. Rikacovs, Model Transformation Languages and Their Implementation by Bootstrapping Method, Pillars of Computer Science: Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday, Arnon Avron, Nachum Dershowitz, and Alexander Rabinovich, editors, Lecture Notes in Computer Science, vol. 4800, Springer-Verlag, Berlin, 2008.
19. T. Fischer, J. Niere, L. Torunski, and A. Zundorf. Story diagrams: A new graph rewrite language based on the Unified Modelling Language. In G. Engels and G. Rozenberg, editors, Proc. of the 6th International Workshop on Theory and Application of Graph Transformation, volume 1764 of LNCS, pages 296–309. Springer Verlag, 1998.
20. Agrawal A., Karsai G., Shi F. Graph Transformations on Domain-Specific Models. Technical report, Institute for Software Integrated Systems, Vanderbilt University, ISIS-03-403, 2003

21. Kalnins, A., Barzdins, J., Celms, E.: Model Transformation Language MOLA. Proceedings of MDAFA 2004, Vol. 3599, Springer LNCS, 2005, pp. 62–76.
22. C. Ermel, M. Rudolf, and G. Taentzer. The AGG Approach: Language and Tool Environment. In H. Ehrig, G. Engels, H. J. Kreowski, and G. Rozenberg, editors. Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools, pages 551–603. World Scientific, 1999
23. Schürr, A., Winter, A., Zündorf, A.: The PROGRES approach: Language and environment. In Ehrig, H., Engels, G.,Kreowski, H.J., Rozenberg, G., eds.: Handbook on Graph Grammars and Computing by Graph Transformation: Application, Languages, and Tools. Volume 2. World Scientific (1999) pp. 487–550
24. F. Jouault and J. Bézivin. KM3: a DSL for Metamodel Specification. In Procs. FMOOD'06, volume 4037 of LNCS, pages 17–185
25. Balogh, A., Varro, D. Advanced Model Transformation Language Constructs in the VIATRA2 Framework, ACM SAC2006, Dijon, France, 2006
26. ATL: Atlas Transformation Language Specification of the ATL Virtual Machine URL: http://www.eclipse.org/m2m/atl/doc/ATL_VMSpecification%5Bv00.01%5D.pdf
27. E. Rencis, Model Transformation Languages L1, L2, L3 and Their Implementation, Articles of the University of Latvia, "Computer Science and Information Technologies" 2008.
28. Kalnins A., J. Barzdins, E. Celms. Efficiency Problems in MOLA Implementation. 19th International Conference, OOPSLA'2004 (Workshop "Best Practices for Model-Driven Software Development"), Vancouver, Canada, October 2004
29. A. Kalnins, E. Celms, A. Sostaks. Simple and Efficient Implementation of Pattern Matching in MOLA Tool. Proceedings of the 7th International Baltic Conference on Databases and Information Systems (Baltic DB&IS'2006), Vilnius, Lithuania, July 3–6, 2006, pp. 159–167.
30. B. Efron, R.J. Tibshirani, "An Introduction to the Bootstrap", Chapman & Hall/CRC, 1994, 436 p
31. Barzdins, J., Barzdins, G., Balodis, R., Cerans, K., Kalnins, A., Opmanis, M., Podnieks, K.: Towards Semantic Latvia. Proceedings of Seventh International Baltic Conference on Databases and Information Systems, Communications, Vilnius, Lithuania, O. Vasileckas, J. Eder, A. Caplinskas (Eds.), Vilnius, Technika, 2006, pp. 203–218.
32. S. Rikacovs, The base transformation language L0+ and its implementation, Articles of the University of Latvia , "Computer Science and Information Technologies", 2008
33. A. Aho, R. Sethi, J. Ullman, Compilers: Principles, Techniques, and Tools. Bell Laboratories, 1986
34. A. Kalnins, E. Celms, A. Sostaks. Tool support for MOLA. Fourth International Conference on Generative Programming and Component Engineering (GPCE'05). Proceedings of the Workshop on Graph and Model Transformation (GraMoT), Tallinn, Estonia, September 2005, pp. 162–173
35. Celms E., A. Kalnins, L. Lace. "Diagram definition facilities based on metamodel mappings". Proceedings of the 18th International Conference, OOPSLA'2003 (Workshop on Domain-Specific Modeling), Anaheim, California, USA, October 2003, pp. 23–32
36. Eclipse – an open development platform. URL: http://www.eclipse.org/
37. Eclipse Modelling Framework (EMF, Eclipse Modelling subproject), http://www.eclipse.org/emf/
38. Peter Dahm and Friedbert Widmann. GraLab - Das Graphenlabor. Projektbericht 4.3.0, University of Koblenz-Landau, Institute for Software Technology, 07 2003.

39. Java Technology URL: http://java.sun.com/
40. GCC, the GNU Compiler Collection URL: http://gcc.gnu.org/
41. Jouault, F., Bezivin, J., Consel, C., Kurtev, I., Latry, F. Building DSLs with AMMA/ATL, a Case Study on SPL and CPL Telephony Languages. In: Proceedings of the 1st ECOOPWorkshop on Domain-Specific Program Development (DSPD), July 3rd, Nantes, France. (2006)
42. ATL Use Case – Compiling a new formal verification language to LOTOS (ISO 8807) URL: http://www.eclipse.org/m2m/atl/usecases/FIACRE2LOTOS/
43. F. Jouault, and F. Allilaire, An introduction to the ATL Virtual MachineV1.0 draft URL: http://www.eclipse.org/m2m/atl/doc/ATL_VM_Presentation_%5B1.0%5D.pdf
44. Extensible Markup Language (XML) 1.1 (Second Edition) URL: http://www.w3.org/TR/xml11/
45. Slonneger, K. and B. Kurtz. Formal Syntax and Semantics of Programming Languages. A Laboratory Based Approach, Addison-Wesley Publishing Company, 1995.
46. E. W. Dijkstra, GOTO Statement Considered Harmful, Letter of the Editor, Communications of the ACM, March 1968, pp. 147–148.
47. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L. (1990). Introduction to Algorithms, first edition, MIT Press and McGraw-Hill.
48. G. Varro, D. Varro and K. Friedl. Adaptive graph pattern matching for model transformations using model-sensitive search plans. In G. Karsai and G. Taentzer editors, Proc. of Int. Workshop on Graph and Model Transformation (GraMoT'05), volume 152 of ENTCS, pages 191–205, Tallinn, Estonia, September 2005.
49. D. Dotan, A. Kirshin, Debugging and Testing Behavioral UML Models, Proceedings of OOPSLA 2007, Montreal, Canada
50. ReDSeeDS. Requirements Driven Software Development System. European FP6 IST project. http://www.redseeds.eu/, 2007.

LATVIJAS UNIVERSITĀTES RAKSTI. 2008, 733. sēj.:
DATORZINĀTNE UN INFORMĀCIJAS TEHNOLOĢIJAS 179.–212. lpp.

# Technical Solutions for the Transformation-Driven Graphical Tool Building Platform METAclipse

Oskars Vilitis[1], Audris Kalnins

Institute of Mathematics and Computer Science, University of Latvia, 29 Raiņa blvd., Rīga, LV-1459, Latvia, ph.: (+371) 6 7224 363

Oskars.Vilitis@gmail.com, Audris.Kalnins@mii.lu.lv

**Abstract.** The paper gives a detailed description of technical solutions developed for the implementation of a metamodel-based graphical tool building platform whose main area of application is the development of DSL editors. As opposed to the well-known static-mapping-driven approach, the implementation described here provides more flexible means for the definition of the correspondence between the domain and presentation metamodels, using model transformations. The solutions described in the paper form the basis of a newly developed Eclipse plugin METAclipse that allows an easy use of transformations and materializes the ideas of the transformation-driven tool building platform. METAclipse has proven its flexibility and efficiency in the development of a new generation graphical editor for the model transformation language MOLA.

**Keywords.** DSL Editors, model transformations, metamodel-based graphical tool building platform, transformation-driven, Eclipse

## 1 Introduction

Due to the increasing interest in the MDA approach and the growing popularity of various domain-specific languages (DSLs), various graphical tool building environments have gained continuously increasing attention in recent years. The first simple generic metamodel-based tool environments, such as MetaEdit [1], Kogge [2] and early versions of Dome [3] and [4], appeared already in the mid-nineties, but their capabilities were quite limited.

The second generation of such metamodel-based environments with much wider possibilities, such as MetaEdit+ [5], GME [6], and ATOM3 [7], appeared around 2000 (the first version of MetaEdit+ actually appeared much earlier [8]). They already had domain metamodeling facilities close to MOF [9] and more advanced graphical capabilities. Therefore the popular tool paradigm of a visual language being based on a presentation-independent domain (as it is e.g., for UML [10]) could be supported. But the presentation metamodel (the description of graphical elements) still had to be close to the domain metamodel, only relatively simple mappings between them were

[1] supported partially by ESF (European Social Fund), project 2004/0001/VPD1/ESF/PIAA/04/NP/3.2.3.1/0001/0001/0063

permitted, and everything else had to be defined by OOPL code (e.g., C++ in GME). The previous tool framework by UL IMCS, the Generic Modeling Tool environment [11], also belongs to this category.

A completely new generation of tool frameworks has emerged in recent years in response to the need of the MDA community to make DSLs an everyday software development practice. One such group of environments is based on the open-source Eclipse platform. Eclipse, together with its EMF plugin [12], is a broadly used metamodeling environment, close to MOF. In addition, the GEF plugin [13] is a basic "diagram drawing engine." Only something linking the two was required for a complete tool building environment. The first and the most popular solution is the static metamodel mapping-driven GMF platform [14]. Alternative solutions are provided by the Pounamu/Marama [15] environment and the coming GEMS project [16].

A popular alternative to Eclipse on a commercial basis is offered by Microsoft DSL Tools [17] in Visual Studio 2005; however, the logical capabilities there are quite close to GMF. The already mentioned MetaEdit+ has significantly evolved and has also become a key player in this area.

The above-mentioned solutions are quite appropriate for relatively simple cases, where the domain and presentation metamodels are close and no complicated mapping logic is required. However, DSL support frequently requires much more complicated and flexible mapping logic. Therefore a new approach has appeared: to define this mapping by model transformation languages. Model mappings in tools actually lie very close to the traditional MDA tasks, for which model transformation languages were invented. Therefore they can be considered very appropriate DSLs for metamodel-based tool building, yielding development efficiency that is an order of magnitude higher when compared to that of OOPL.

The first frameworks using this approach to a degree are the Tiger project [18] and the ViatraDSM framework [19]. Both are based on Eclipse and use GEF as a drawing engine. The Tiger project is based on the graph transformation language AGG [20]. However, a specific domain modeling notation is used there, which still forces the domain metamodel of a language to be close to the presentation metamodel. Standard editing actions (create, delete, etc.) are specified by graph transformations, which act on the domain model, and the presentation model is updated accordingly. The ViatraDSM framework is based on the Viatra2 transformation language [21]. In this framework, the domain metamodel must be close to the presentation metamodel too, but larger freedom is allowed, and the transformation approach can, to a degree, be combined with the static mapping approach. There are also plans to use the Fujaba [22] transformation language in the MOFLON framework [23].

A detailed analysis of the two approaches and their strengths and weaknesses has been done in the paper "Building Tools by Model Transformations in Eclipse" [24]. This paper concentrates on a thorough description of the technical solutions developed in order to implement the fully transformation-driven tool building platform METAclipse. METAclipse is partly being developed within the project "New Generation Modeling Tool Framework," which is funded by ERDF (2006–2008). Within this project, another tool implementing similar ideas, GrTP [25], is also being developed, however with a different profile: its aim is to handle various tasks related to the semantic web.

In METAclipse there are no restrictions on the correspondence between the domain and presentation metamodels. The mappings are defined dynamically by transformations in the model transformation language MOLA [26]. METAclipse is implemented as an Eclipse plugin and reuses the basic Eclipse components such as EMF and GEF, as well as parts of the GMF runtime [14]. METAclipse obeys traditional Eclipse style and behavior guidelines and therefore can be integrated in other eclipse-based development environments. Also, it is possible to integrate other Eclipse technologies like model-to-text generation. An overview of the platforms architecture and the rationale behind the METAclipse framework will be presented in section 2.

The main distinguishing feature of METAclipse is an appropriately built presentation metamodel, which is discussed in detail in section 4. It enables a clear separation of responsibilities between the METAclipse presentation engines, which handle all the low-level presentation and layout-related tasks, and transformations, which create and maintain only the domain and the logical structure of presentation.

Section 5 provides a brief sketch of transformations in METAclipse, however, they are not the main topic of this paper. The emphasis of this paper is on the structure and functionality of the METAclipse framework itself.

METAclipse is already proven to be useful in practice by creating an editor for the MOLA language itself (MOLA is a graphical model transformation language, thus being a remarkable example of a DSL). This editor is successfully being used in the European IST 6th framework project ReDSeeDS [27]. All figures containing class diagrams in the paper have been created with the MOLA metamodel editor.

## 2 Overall METAclipse Architecture

A graphical modeling tool must deal with many complex tasks, such as proper domain element representation; intuitive and standardized element editing; correct model modifications in response to the graphical editing events; providing a convenient way of navigating through models and a clear way of model element property representation; etc. The most complex and time consuming tasks are the ones concerned with the graphical representation and user interface handling. Luckily, a number of these tasks are common to all graphical tools (i.e., they are domain-independent) and can be handled at the tool-building platform framework level.

### 2.1 Basic Principles of the METAclipse Framework

In METAclipse, a well-defined framework is provided for the tool builders. The top-level view of the METAclipse architecture is very simple (see Fig. 1). METAclipse itself consists of a set of Eclipse plugins that define the framework of the tool building platform and comprise several so-called presentation engines, each of which deals with a particular set of graphical editor tasks (project tree engine, element property engine, etc.). Each of these engines will be discussed later in this paper in section 4.

METAclipse plugins contain all the common functionality needed for the tools and relieves the creator of the tool from the need to worry about many technical user interface issues. The part that defines a concrete tool and that must be written by the toolsmith is the transformation library containing all the necessary model transformations that change the model according to the user actions in the tool.

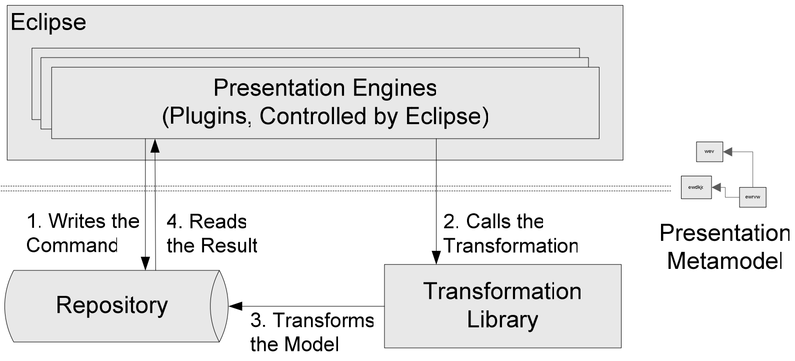


**Fig. 1.** High-Level view of the METAclipse architecture

In METAclipse the toolsmith must start with the creation of the domain metamodel and proceed with wiring the domain metamodel to the presentation metamodel through writing the model transformations. Thus the only items the toolsmith builds for a concrete tool are the domain metamodel and the transformation library defining the functionality. In the paper the combined metamodel of presentation and domain metamodels will be referred to as METAclipse metamodel. Accordingly, the combination of domain and presentation models will be called simply model. Manipulations with the domain model are completely the responsibility of the transformation writer. METAclipse framework provides no support for the domain model modifications.

Every framework engine exposes its features to the transformations through a strictly defined metamodel that serve as an interface between the transformations and editors. Metamodels of the engines will be discussed in more detail in later sections of this paper. Part of each engine's metamodel is also the available set of commands that could occur as a result of user actions. Commands are used to trigger the transformations and a single command instance represents one atomic user action, which constitutes the smallest piece of work in the framework. All actions that make purely graphical changes are handled directly by METAclipse framework. Only semantic actions (actions causing domain model changes or any changes in the presentation model that are specific to a concrete tool) are transformed into the commands and passed to the transformations for execution.

Together metamodels of all engines form the presentation metamodel of METAclipse. Each element displayed in the tool, created using METAclipse, corresponds to a presentation model element (an instance of some presentation metamodel class). Presentation model as well as domain model (model on which the tool actually operates) are stored in the model repository and are changed by the transformations as a reaction to the user triggered events. Every semantic user action in METAclipse results in the following sequence of actions:

- The presentation engine that gets some user action writes the command corresponding to the action taken (right click on a project tree node, creation of an element, drawing a link between elements, etc.) to the model repository and invokes the main transformation (steps 1 and 2 in Fig. 1);
- The main model transformation recognizes the command written and makes the necessary changes to the presentation and/or domain models (step 3 in Fig. 1);
- Presentation engines read the model changes and react accordingly: show context menu, show newly created element or edge, etc. (step 4 in Fig. 1).

Such top-level view of METAclipse architecture can be compared to the traditional MVC approach: the role of the controller is played by transformations; the repository serves as the model, and the presentation engines act as the view. It must be noted that METAclipse leverages the abstraction level of the MVC approach: the controller (transformations) receives only the semantic actions.

In order to make the METAclipse architecture and functionality more clear, an example state of the project tree engine is given in Fig. 2. A visual representation of the project tree engine is given on the left. In the middle, a part of the simplified project tree engine metamodel is shown. Here one can see how the visual editor elements are represented to the transformations: ProjectTreeNode class represents one node in the project tree. The ShowMenuCommand class represents a right-click event on the tree node and expresses user request to show the context menu.

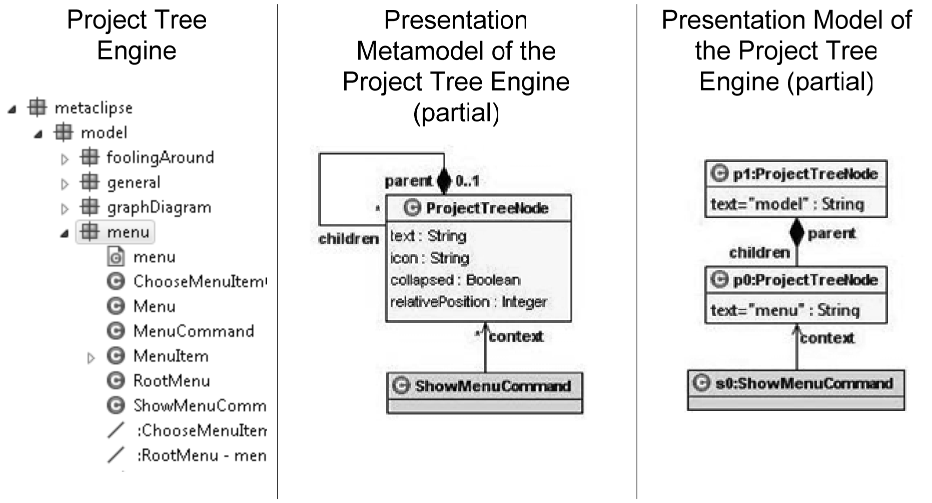


**Fig. 2.** Example of a project tree engine and its metamodel and model states

Let us imagine that one has right-clicked the node called "menu" in the tree and the project tree engine has written the ShowMenuCommand instance to the repository (step 1 in Fig. 1). At the right side of Fig. 2 the presentation model part is given, showing the instances involved in the handling of the right-click event. As the next step in event processing, the engine will invoke the transformation (step 2 in Fig. 1).

The transformation will find that ShowMenuCommand has been written in the repository and will create presentation metamodel instances (not shown in the Fig. 2) comprising the needed context menu (step 3 in Fig. 1). No domain model changes are needed in this example. At last, Eclipse will get back the control and presentation engines will be notified of the model elements changed. The menu engine will see that a menu has been created, so it will show the context menu for the project tree node called "menu."

## 2.2 Solutions Chosen for the METAclipse Implementation

METAclipse is built on top of Eclipse technologies and is packaged in the form of several Eclipse plugins. Eclipse was chosen as a mature and widely appreciated platform, providing a large number of frameworks covering many needs of the tool developers. Eclipse is also a very popular choice of a wide variety of leading production-quality software development platforms that could potentially gain from integration of modeling and DSL editor tools.

The transformation language MOLA [26, 28], developed by LU IMCS, was chosen for the implementation of transformations. MOLA has a rich set of language elements and it had already proven its performance and stability in practice, so it was a natural choice. The current implementation of MOLA is compiled to a Windows DLL file and works against the repository MIIREP (codenamed "OUR" in the paper "Towards Semantic Latvia" [29]), also developed by LU IMCS. Therefore, the choice of the repository was also clear. However, to make METAclipse more flexible, it was decided to make the access to transformations and the repository transparent so that it would be possible to switch to other transformation languages and/or repositories. The repository access solution will be described in Section 3.

As discussed in the previous section, every METAclipse presentation engine exposes its features to the transformations through its metamodel. What is actually displayed in the editor is a visual representation of the engine metamodel instances, i.e., models. In Eclipse, Java code needs to access this model information. To accomplish this, physical in-memory model storage is needed. The framework fitting these purposes already exists and is called EMF [12]—Eclipse Modeling Framework. EMF is being used in many Eclipse-based tool building platforms as the model repository.

EMF was also chosen for implementation of the METAclipse model repository, as it has several features that fit the framework needs. EMF provides a generator for the creation of Java classes that correspond to the model elements. This eases the creation of the runtime model classes. Another important feature of EMF is the model change notification mechanism implemented through model listeners that allow easy and dynamic model change transfer to various presentation engine parts. There are also some aspects of the EMF that are currently less important for METAclipse, which however could turn useful in time: XMI import/export, OCL implementation, etc.

This leads to the presentation model in METAclipse being stored in the EMF repository. Transformations, however, also need to operate on this model. As transformations work on an external repository, a challenge rises to synchronize the

EMF model instances with information in the MIIREP. More details on the non-trivial solution will be given in Section 3.

The EMF framework is not the only Eclipse framework used in METAclipse. For various METAclipse needs, others are used as well:

- The property engine uses the tabbed properties framework for dynamic generation of the element property sheets (see Section 4.5 for detailed description of the property engine);
- The project tree engine (described in Section 4.3) uses the navigator framework;
- The graph diagram engine (described in Section 4.6) uses the Graphical Editing Framework GEF [13] and parts of the Graphical Modeling Framework GMF [14] runtime.

## 3 Interaction with the Repository and Transformations

As already stated before, editor interaction with the repository and transformation invocation was intended to be made as generic as possible in order to maintain the possibility to change the implementation of repository or transformations if necessary. To achieve such independence, two problems had to be solved. First of all, an interface to the set of external repository operations used in METAclipse (such as find object, store object, change object property etc.) had to be defined. Transformation invocation is also part of this interface, as transformations are always related to a particular repository. Secondly, a generic mechanism to transfer the repository data to EMF object instances had to be developed in order to allow the handling of repository objects in Eclipse as if they were normal EMF objects, thus giving the access to the entire infrastructure provided by EMF.

### 3.1 Repository Interface

The repository interface itself is nothing special; it is a regular Java interface containing all the operations required by METAclipse. The interface contains the following sets of operations:

- Metamodel (object type) manipulations, such as creating a class, adding a class attribute, finding classes, creating associations, etc.
- Model (object) manipulations, such as finding an object of a certain class, creating objects and setting object attributes, etc.
- Transformation invocation. Only one function for this is required, as transformations have just one entry point in the METAclipse architecture.

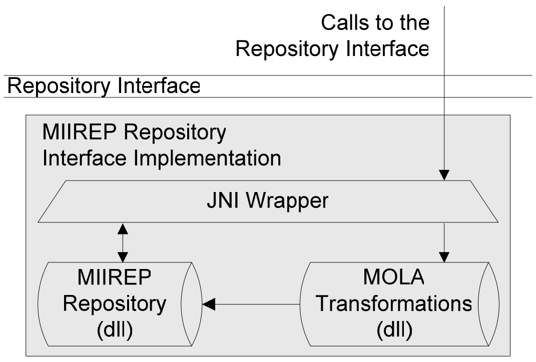


**Fig. 3.** MIIREP repository interface implementation

MOLA transformations currently are compiled against the MIIREP repository, which is developed in C++ and released as a Windows DLL file. MOLA transformations themselves are also compiled to a DLL file, which directly accesses the MIIREP DLL loaded in memory. This implies that the MIIREP repository interface implementation currently used in METAclipse (see Fig. 3) uses a JNI (Java Native Interface) wrapper for the repository operations (see [30] for information on JNI). The wrapper delegates all repository access operations (model and metamodel manipulations) to the appropriate MIIREP repository API functions and the invocation of transformations to the transformation library.

## 3.2 The Link Between Eclipse and the Repository: "Wise" Objects

As stated before, all presentation engines (Eclipse plugins) developed work with EMF runtime objects in order to gain all the benefits the EMF framework is offering. Transformations, on the other hand, work with the external repository, so synchronization between the repository and EMF is required.

The task of integrating the external repository seamlessly into the Eclipse EMF framework was quite challenging. Simple interface did not satisfy the requirement to keep Java-side code unaware that anything other than EMF is used, which is why the "wise" object mechanism was created. The main reason for such a requirement was the wish to keep the possibility to switch to a clean EMF implementation in the future (meaning that no external repository would be used, with EMF itself serving as the repository), as well as to be able to use clean EMF infrastructure.

Another aspect that had to be taken into account was performance. As every little action in the editor results in changes in the repository through the invocation of the transformation, a complete re-read of all repository data after each operation is unacceptable. Only the "dirty" or changed information has to be transferred back to EMF object instances.

To comply with the given requirements, a special mechanism was developed, consisting of alternative EMF runtime objects that conform to the EMF interfaces and externally look like normal EMF objects, but internally do all the synchronization with the repository. These objects were named "wise" objects, as they show certain "intelligence": though from the interface perspective they look like normal EMF objects and support all EMF framework operations, internally they know when and how it is necessary to read or write some information to the repository. The standard EMF notification mechanism is used to notify any changes occurring in the repository. "Wise" objects can be considered the second level of repository abstraction, which introduces the caching mechanism, conforms to the EMF object interfaces and uses first level abstraction—repository interface—to read and write data to the repository.

ECore, the core metamodel in EMF, is very similar to the EMOF (Essential MOF), a subset of the MOF model [9]. In fact, there are just some small, mostly notational, differences between these two. According to the MOF hierarchy, ECore is at the M3 layer, the same as MOF itself. The code generation facility provided by EMF can be used to generate Java runtime classes for a particular metamodel (M2 layer) defined by ECore. Instances of the generated runtime classes then correspond to the M1 layer in MOF.

ECore metamodel classes (ECore base classes) define the class hierarchy that forms the basis for the Java runtime. All EMF runtime classes generated for a particular metamodel extend these base classes. ECore base classes provide all the functionality to the generated classes and allow using them in EMF infrastructure by providing all the EMF framework features. Therefore, base classes are the best place where the repository synchronization should be implemented.

## "Wise" Objects as an EMF Extension

Base ECore classes were extended and a set of "wise" object base classes was defined (see Fig. 4). By analogy to ECore classes, base "wise" object classes, together with some helper classes comprising the whole "wise" object concept, were called WCore. In WCore, the methods inherited from ECore for getting and setting the properties are extended with functionality of reading and writing data from and to the repository through the repository interface described in the previous section. For performance considerations, "Wise" objects keep track of the state of every object property and cache the data from the repository in the object instance, so the consequent reads of the same property will result only in one read of the property from the repository.

The fact that the parent of all ECore classes is a single class—EObject (see [12] for complete ECore structure)—simplified the extension of ECore. For "wise" object needs it was enough to extend just two ECore classes, EObject and EFactory, with the corresponding WObject and WFactory classes. WObject contains all the caching and synchronization logic and, as it is the superclass of all the other framework classes, the logic is available all across the framework. The WFactory extension of the factory class was needed, as some initialization of the "wise" object on its creation was required.

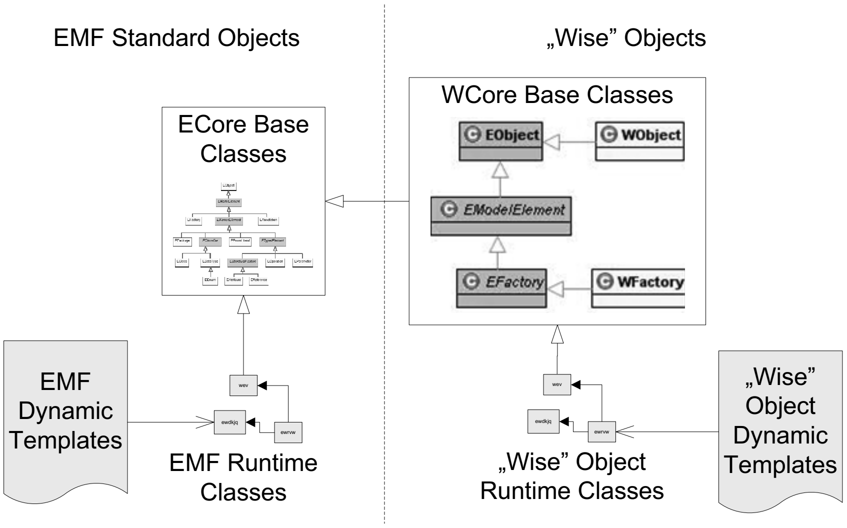


**Fig. 4.** "Wise" object dependencies

To put the WCore classes in action, the EMF generator also had to be extended so that it produced "wise" objects extending WCore base classes. The EMF framework uses the so-called dynamic code templates (using another Eclipse framework for the code generation—JET [31]) during the generation process of the runtime classes. The EMF generator reads the serialized form of the metamodel and then, using the set of templates, generates the runtime classes (see Fig. 4). Default templates producing EMF runtime classes were extended so that they would generate the code using WCore instead of ECore.

The complete set of classes comprising the WCore can be seen in Fig. 5. The above-mentioned extension of getter and setter methods of ECore is divided into two classes. Reading of the attributes from the repository was easiest to implement in the WObjectImpl class itself, in the inherited getter methods. Writing the attributes, however, was easier to move to a separate class WObjectChangeObserver, which implements the EMF change listener and is attached to every instance of WObject. The change observer listens to any changes done to the WObject from the engine side and if any occurs, writes the data to the repository.

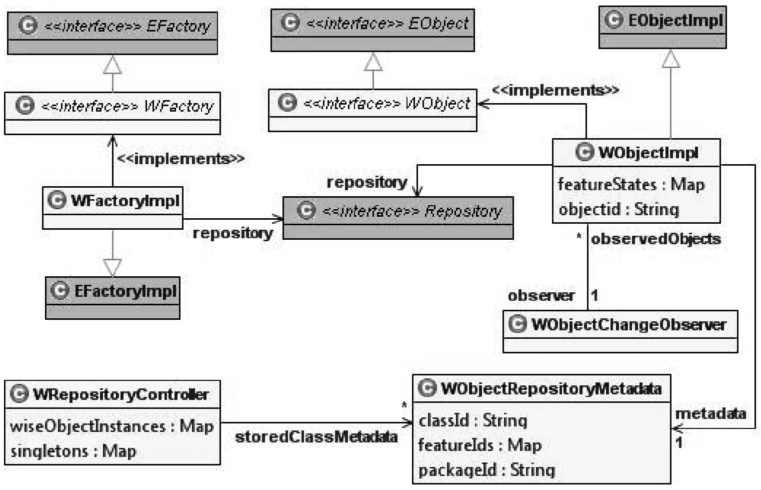


**Fig. 5.** WCore class diagram

To be able to read and write the repository data, “wise” objects need to have a possibility to map the classes, attributes and associations to the corresponding repository objects. Such mapping can be defined only at M2 level and thus it is necessary to have the WCore class and feature mapping to the repository metadata at the M2 layer. As it is inefficient to read these mappings every time any object is accessed, class metadata mappings are cached. The WRepositoryMetadata object represents the class metadata. The map of WCore class to repository metadata mappings is held in the WRepositoryController object and the mappings are attached to every WObject instance for convenience when instantiating it (as a reference to the cached mappings).

## Repository Change Notification in METAclipse

Extending the ECore base classes covers the synchronization needs only from the METAclipse perspective, i.e., if changes to the model are done from the engines. However, the most intense model changes happen on the other side—in the transformations. Therefore, another missing piece is a change notifier back from the transformation, which would trigger the EMF change events for all objects that have been changed in the repository. In WCore, the WRepositoryController class takes care of this. There, a special method is defined for change detection, which has to be invoked after each transformation execution.

Transformation change notification is not a trivial task, as it is also constrained with tight performance requirements. It is very inefficient to detect the changes already after transformation execution, as it means inspection of all object instances in the repository. This means that a support from the side of the transformations is required in order to make an efficient implementation of the change notification. This is why WRepositoryController change notification method is designed in a way that it calls special functions of the repository interface in order to get the lists of the changed or deleted objects. Functionality of tracking changes is left to the implementation of the interface. When changed or deleted object lists are read from the repository, WRepositoryController issues the corresponding EMF notifications and the changed features of the object instances that have changed are set “dirty,” so that they are once again read from the repository instead of using the cached values from the WObject instances. The concept of “wise” objects is not trivial and is best understood on a concrete example. One such example is provided in Section 4.2.

In case of the repository and transformations currently used in METAclipse, it was very easy to track object deletions, as the MIIREP repository itself has the functionality to track such changes. However, the tracking of the changes to the existing objects had to be incorporated in the transformations. For this reason, a special class “Changes” was introduced in the presentation metamodel. Each transformation is responsible not only for making the actual changes, but also for adding a link from the “Changes” singleton object to the objects actually changed. See Section 4.1 for more information on the “Changes” object and the singleton concept.

Of course, it would be more convenient to have also detection of changes to the existing objects automated and incorporated at the repository level, but unfortunately MIIREP does not provide such a possibility. In case of MOLA, as its transformations are compiled, it is also possible to add special functionality in the MOLA compiler that automatically adds the “Changes” link. However, at the moment such functionality is not implemented.

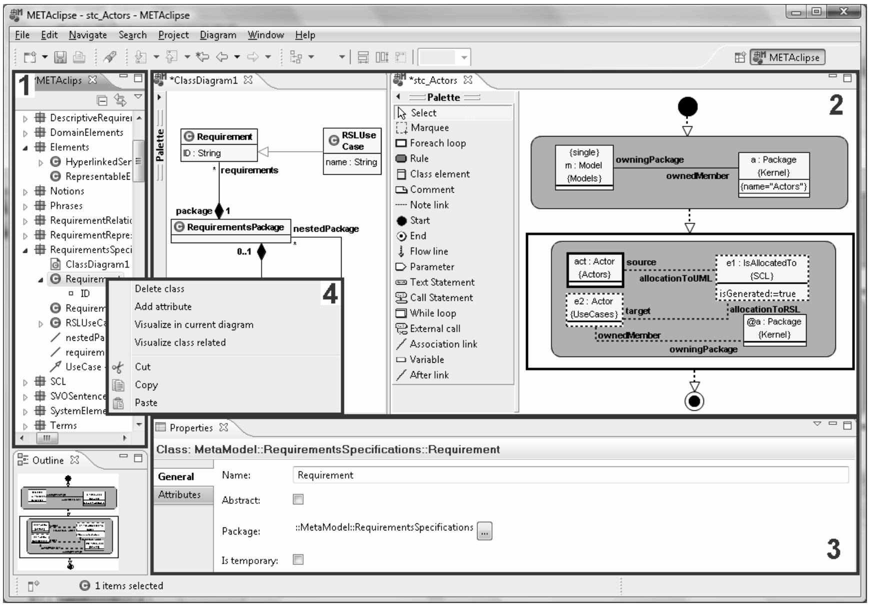
## 4 Presentation Engines

As already stated before, METAclipse consists of several presentation engines. Although there are some additional smaller helper parts in METAclipse, four main presentation engines can be named that together comprise the whole tool building platform (in Fig. 6 all of them can be seen in action).

1. Project tree engine, responsible for organization of projects, models and model elements in a hierarchical tree structure;
2. Graph diagram engine: the main engine of METAclipse, providing editing capabilities to the graph diagrams;
3. Property engine: provides property editing capabilities for other engines (like properties for a selected item in the project tree or a selected diagram element);
4. Menu engine: used by other engines for the displaying of context menus (like by project tree engine for showing context menus of the tree nodes or by graph diagram engine for showing context menus on the diagram elements).

Besides these four engines, additionally there are some less important components in METAclipse responsible for common functionality like drag-and-drop, clipboard, METAclipse perspective; utility functions; transformation control etc. These will not be discussed here. In the following sections the focus will be put on the interaction between the engines and transformations, and special attention will be paid to the description of all the presentation metamodels, as they form one of the most important aspects describing the METAclipse functionality.

The look and feel and general operation principles in METAclipse engines were adopted from Eclipse standard editors so that the editors would fit smoothly in the Eclipse environment. This means that some eclipse standards were obeyed. For example, METAclipse does not use dialogs for the diagram element creation. Instead, all element properties are assigned default values, which can later be changed to the desired values through the properties view. Properties are displayed in one single view for all editors, implying that just one editor is in focus at all times.



Class: MetaModel::RequirementsSpecifications::Requirement

Name: Requirement

Abstract: ☐

Package: ::MetaModel::RequirementsSpecifications

Is temporary: ☐

**Fig. 6.** METAclipse presentation engines in action

In the development of the presentation engines, one simple rule drove the splitting of functionality between the engine and transformations:

- Every task that needs any information read from the domain model, i.e., that is domain-specific, has to be done by transformation;
- All tasks that do not require any knowledge of the domain have to be done by the engines.

So, for example, the right click on the project tree node for showing the context menu needs the knowledge of what kind of node it is in order to know what menu options to offer. This means that this is a task for a transformation. Another example—the move of a diagram element within the borders of the same parent—does not require any knowledge of the domain. Such operation requires only changing of some presentation model attributes, thus it can be carried out by the engine itself. If, in contrast, the diagram element was dragged out of the borders of the parent element (for example, dragged from one sub-diagram to another), this again would require some domain model changes and thus it is a semantic operation that has to be performed by transformations.

## 4.1 Presentation Metamodel Structure

The transformation library is the changing part in METAclipse from one tool implementation to other. That is why transformation creation must be made as easy as possible in order to make METAclipse useful and convenient for the toolsmiths. In order to accomplish this there are several prerequisites to be met:

- A well-established set of base transformations common to all or at least most editors must be provided. This would form the base framework for transformations to be created by the toolsmith. This would allow the toolsmith to concentrate on semantic tasks for mapping of domain elements to presentation elements and would remove the need to worry about some tasks that could be done by the framework (for example, handling of the element styles, parts of copy and paste logic, building of standard menus, etc.);
- A set of helper transformations must be provided, so that the transformation creator has decent artillery at hand for handling of different kind of tasks (utility functions);
- It is very important to create a good interface to the presentation engines. In this case engine metamodels compose this interface. A proper presentation metamodel is extremely important for the transformation creators to make work with the editors easy and convenient.

A very short overview on the solutions provided by METAclipse for the first two will be given in Section 5. The focus in this paper however is on the last—proper design of the presentation metamodel. A large amount of effort and time was invested in the design of this metamodel to make it best usable from transformations. The following few sections will give a thorough description of various parts of it, i.e., of various presentation engine metamodels.

The presentation engines rely heavily on various Eclipse frameworks. Therefore, the metamodels of the engines could be partially extracted from them. It must be noted, however, that none of the used Eclipse frameworks had a metamodel already defined. Metamodel of every engine had to be synthesized from the corresponding framework API. Then it had to be amended with the METAclipse-specific classes needed for the engine.

As the metamodel is an interface between two parties, transformations and Java code, it has to be conveniently usable from both sides. However, more importance must be given to the transformation requirements for the metamodel. It was decided to adopt the naming and structuring standards of classes from the Java coding standards, keeping in mind not to make any transformation tasks complicated. As it turned out, it is very convenient for both sides if the metamodel is structured in strictly hierarchical and logically split packages. The whole presentation model contains the following packages:

- the *general* package contents include the base classes used by the presentation metamodel, classes common to all engines and various types used across the presentation metamodel;

- the *project* package contains all the classes needed for project handling in METAclipse and classes for steering the project tree engine (see section 4.3 for the description);
- the *menu* package contains classes for steering the menu engine (see section 4.4 for the description);
- the *properties* package contains classes for steering the properties engine (see section 4.5 for the description);
- the *graphDiagram* package contains classes for steering the graph diagram engine, excluding the classes for palette organization (see section 4.6 for the description);
- the *palette* package contains classes for creation of the editor palettes. This was created as a separate package, because palettes may be required not only by graph diagrams. Palette elements could be reused also if another kind of editor engine were created.

## The Common Part of the Presentation Metamodel (*general* Package)

The *general* package defines the core classes of the METAclipse presentation metamodel (see Fig. 7). In this and following figures a special color-coding will be used. Normal metamodel classes will be shown in white. Light gray color will represent the command classes. For more information on what a command is, see Sections 2.1 and 4.1, as well as descriptions of METAclipse presentation engines. The dark gray classes will denote the singletons. The description of the term "singleton" is given below.

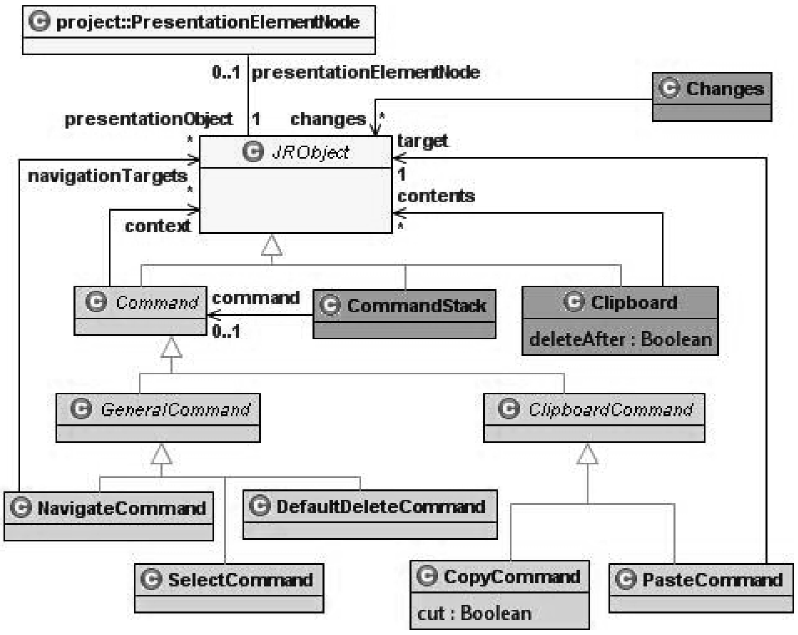


**Fig. 7.** The general part of the presentation metamodel

As the metamodeling practice shows, and also as the preliminary experience of METAclipse technology evaluation proved, it is very convenient to have one superclass for all classes in the metamodel and to organize all classes in strict hierarchies. Just as Java has a superclass of all classes, "Object", the METAclipse presentation metamodel also includes such a superclass, JRObject. One example of how the introduction of such a superclass helps is the case when there is a need to define a very general association to any kind of object. This can be done only if there is a superclass for every object needed to be referenced. In the *general* package this is used to model the concept that any presentation model element can be displayed in the project tree engine as a node: association between PresentationElementNode and JRObject (see Fig. 7).

A concept used across all metamodels by engines for finding the starting points of various parts of models is singletons. Singletons are classes that have exactly one instance. This fact is used by the presentation engines to find the only instance just by knowing the class name. Singleton classes are used in METAclipse engines everywhere where there is a need for an entry point in the model. In the *general* package one example of singletons is the Changes class. This class is an important singleton, which is used to find all the changed or deleted objects after the execution of a transformation.

As discussed in Section 3.2, for wise objects to work there is a need for change tracking after each transformation invocation. Current implementation of the MIIREP repository and MOLA transformations does allow automatic tracking of deletions; however changes must be tracked by each transformation manually. The Changes singleton instance must be linked through "changes" association to every presentation model object changed by the transformation. Engines will then use the singleton nature of the Changes class to find the only instance and read the list of the changed model objects.

The *general* package also contains the supporting and base classes for one of the backbones of METAclipse, namely, the command infrastructure. Commands have already been discussed before. A command in a presentation metamodel corresponds to a possible user action in the editor that requires some reaction from the engine, i.e., the invocation of a transformation. Command class in the metamodel is the superclass for all the command classes. Command base class defines the "context" association: every command can have links to some JRObject instances that form the context of the command. All commands are structured in a strict class hierarchy: for every logical set of commands, an additional superclass is defined (as GeneralCommand and ClipboardCommand in Fig. 7). This opens diverse command parsing possibilities in transformations.

The sequence of command execution in METAclipse is described in Section 2.1. After any user action, a corresponding command is written to the repository. CommandStack singleton instance is linked to the written command. Transformations then seek the command to execute by querying the "command" link of the CommandStack singleton. Currently this link points to at most one instance of a command. After execution, the transformation may write back some results to the executed command by setting some attributes or links. Finally, engines read the command after the transformation execution in order to get the transformation results, if needed.

The rest of the *general* package classes shown in Fig. 7 are common classes used by many presentation engines. This includes some common command classes and the clipboard-supporting classes. NavigateCommand is used as a response to double-clicking on some project tree node or diagram element. Such action would result in opening a diagram in the editor and possibly selecting some diagram element (or multiple elements), if the element under the cursor were a diagram or diagram element. Transformations must return the diagram to open or diagram elements to select by setting the navigationTargets link. It will be queried by the engines after the execution of the transformation to find the objects to open / select.

SelectCommand is executed if any object is selected. It must be used by transformations to generate the property sheets corresponding to the selected object. See section 4.5 for more information about the properties engine. Command DefaultDeleteCommand is executed if the delete button is pressed on any of the selected objects. As the name suggests, transformations should carry out the default delete action when processing this command. Such a command is especially useful for diagrams—usually it is possible to delete an element from the diagram while retaining the domain element or to delete both the diagram and the model element. Different tools require different default logic on such operation.

For clipboard operations, the Clipboard singleton and two commands for copying and pasting are defined. The Clipboard singleton contains links to the copied or cut objects (through “contents” association); the deleteAfter flag is used to distinguish the copy and cut operations. Copy command is executed when the selection is copied. Selected objects are linked to the command through the “context” association. Paste command is executed when users executes the paste operation in the engines.

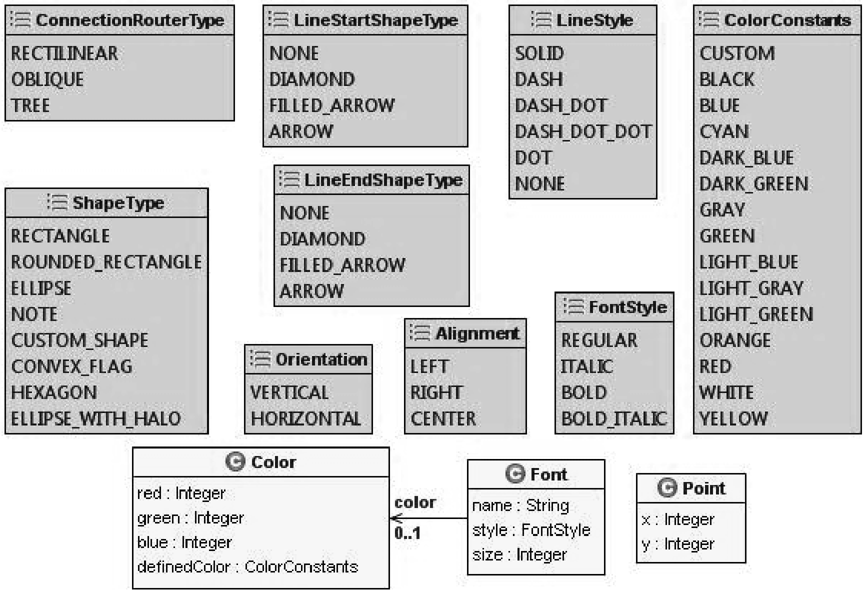


**Fig. 8.** General type part of the presentation metamodel

Finally, the last set of classes found in the *general* package consists of the various types used across the entire METAclipse presentation metamodel. These include definitions of enumerations like Alignment, ShapeType, Orientation, etc., as well as some type classes like Color, Font and Point.

## 4.2 Interaction between the Transformations and Engines

The mechanism of the interaction between the engines and transformations has already been outlined. Now, as all the concepts of the components involved in METAclipse (engines, wise objects, repository, transformations and presentation metamodel) have been introduced, it is time to put it all together. This section will give an example of how all of the METAclipse components fit together before proceeding to the detailed descriptions of the separate engines in the sections to follow. See Fig. 9 for a detailed operation schema of the opening of a new diagram from the project tree. Solid lines in the figure represent the control flow; dashed lines, simple operations like creation of objects.

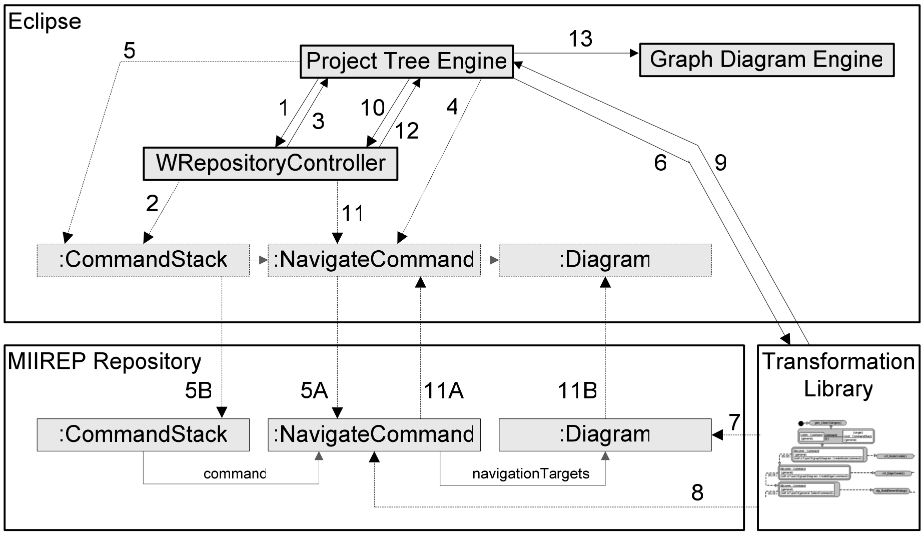


**Fig. 9.** Opening a new diagram from the project tree: an example of the METAclipse component interaction

Let us imagine that a user has double-clicked a node in the project tree that represents a graph diagram. This results in invocation of the project tree engine (discussed in more detail in section 4.3). This engine must react so that a corresponding diagram is opened. Such operation includes the following steps:

- 1: The project tree engine asks WRepositoryController to find the singleton instance of the CommandStack class (see previous section for information about singletons, repository controller, and command stack).

- 2: If this is the first time CommandStack singleton is used, WRepositoryController searches the repository for the single instance of the class with the name “CommandStack.” As it is a singleton, there will be exactly one instance. The repository controller loads the CommandStack wise object instance and caches it, so that the next time the CommandStack is queried, it would be retrieved from the cache.
- 3: The CommandStack wise object is returned to the project tree engine.
- 4: The project tree engine creates a new instance of NavigateCommand wise object and links it to the project tree node wise object, on which the double-click was performed (not shown in the figure). As the NavigateCommand has not been yet saved to the repository, for the time being no synchronization with repository is carried out.
- 5: The project tree engine links the newly created command to the CommandStack. At this moment CommandStack wise object notices that a new link has occurred. As the linked object is not yet saved to the repository, it asks the NavigateCommand instance to save itself to the repository (5A). Then the CommandStack wise object links the repository instance of CommandStack to the newly created instance of NavigateCommand (5B).
- 6: Now, when the command is written to the repository, the transformation library is invoked.
- 7: Transformation detects the NavigateCommand instance linked to the CommandStack and finds which project tree node was double-clicked. Then it searches for the corresponding diagram to be opened.
- 8: Transformation links the Diagram instance found to the NavigateCommand as a result of the execution. Additionally, it puts a link from the Changes singleton (see previous section) to the NavigateCommand in order to signal that NavigateCommand instance has changed.
- 9: Control is given back to the project tree engine.
- 10: The project tree engine calls the WRepositoryController in order to invoke the repository change notification process and synchronize the wise object state with the repository.
- 11: WRepositoryController reads the Changes singleton to detect that the wise object instance of NavigateCommand has changed. It then notifies the NavigateCommand wise object that it must read its contents from the repository instead of its cached data (11A). This also causes the instantiation of the linked Diagram object (11B).
- 12: Control is given back to the project tree engine.
- 13: Finally, the project tree engine delegates control to the graph diagram engine and passes the Diagram wise object to be displayed. Graph diagram engine then uses the Diagram object as the root for reading all the contents to be displayed on the diagram.

All engines operate similarly and the wise object technology is used throughout all METAclipse for synchronization with the repository. This ensures consistent interaction with the transformations. It must be noted that only one transformation at a time can be executed. This, however, does not cause any problems, because in the graphical editors the user makes just one action at a time and actions are sequential.

We could continue describing property generation for the element that is currently selected. However, the operations for that would be very similar to the ones already described. The only additional operation for building of the properties would be the querying and modification of the domain model. This, however, is hidden from the METAclipse framework, as only transformations are responsible for the operations with it and only transformations can access the domain model.

## 4.3 Project Tree Engine

Every graphical tool needs some means of organizing the model objects in a hierarchical tree structure to enable the navigation through models—similarly to how files and folders are organized on the computer hard drive. At the minimum, it is required to display the diagrams as a list, so that the user could choose the one he/she desires to edit.

Eclipse defines the notion of "project" as the highest level of organization. Different tools built on Eclipse provide different kinds of projects: for example, Java, C++, GMF and others. METAclipse also defines a separate kind of project, the METAclipse project. A METAclipse project corresponds to one repository instance, which is created together with the project. All elements of the project model are stored in this repository, e.g., if there are several diagrams in one METAclipse project, they all will be stored in the same repository instance.

For organization of project artifacts, Eclipse provides the so-called navigator framework, which provides a view for displaying of items in a tree. The METAclipse project tree engine is built using this framework and implements its own view (see Fig. 6, part 1). The Eclipse navigator framework already provides all the functionality required to manage the project tree. The only thing needed to implement a specific project tree is the implementation of Navigator interfaces for the retrieving of the model data (or the so-called provider-interfaces, which is a concept used also in other Eclipse frameworks). This is an easy task, as the interfaces require an implementation of a few very simple methods like one for getting the children of a given node and another for getting the parent of a given node. METAclipse provides the implementations of these interfaces for reading the project tree data from the repository. This implementation was very easy to create: just about 100 LOC was required, which was clearly less than would be needed if all functionality had to be created from scratch.

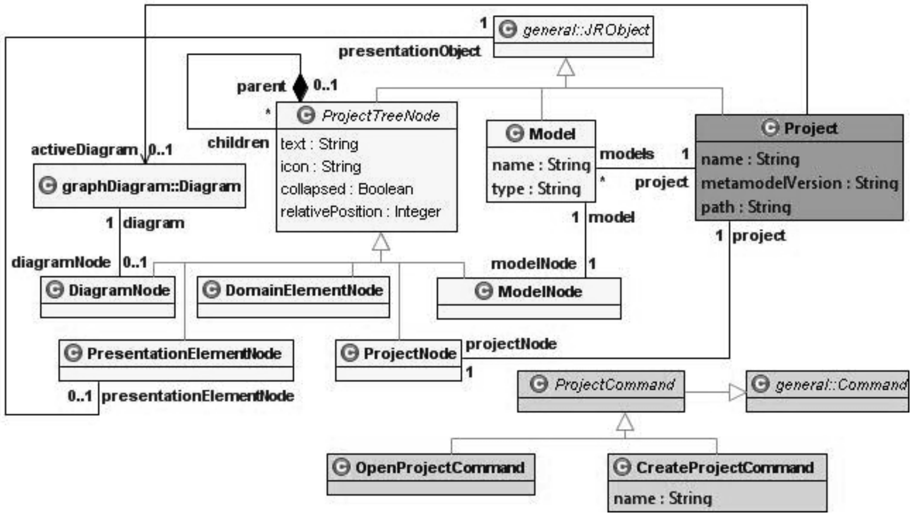


**Fig. 10.** Project part of the presentation metamodel

Fig. 10 shows the metamodel of the project tree engine. When a METAclipse project is opened, first the Project singleton is used to find the ProjectNode instance, which then is interpreted as the root of the project tree. Every METAclipse project will always have exactly one ProjectNode. Project is a singleton that represents the METAclipse project opened in the platform (recall that there is one-to-one correspondence between a METAclipse project and a repository instance).

The ProjectTreeNode class is the superclass of all kinds of tree nodes, ProjectNode included. This class allows defining the hierarchical structure of the tree through the parent-children association. Every instance of one of its subclasses will appear in the project tree engine as a separate node with the given text and icon and ordered by the relativePosition. Transformations are free to define any kind of project tree structures, using the ProjectTreeNode building blocks. There are five kinds of nodes at their disposal, each with a slightly different support from the engine's side:

- *ProjectNode*. Interpreted by the engine as the root project node;
- *ModelNode*. Interpreted as the node defining the boundaries of one model. The model term is introduced to allow further grouping of project items in smaller pieces of work. On possible use of the ModelNode and Model classes could be for the demarcation of the nodes that correspond to the packages in the domain or, if the domain metamodel provides the term of model (like UML domain model [10]), to the models;
- *DiagramNode*. Interpreted as a node that can be opened and represents a diagram. Transformations must make sure that tree nodes of this kind are linked to a corresponding Diagram instance;
- *PresentationElementNode***.** Interpreted as a node that represents some diagram presentation element. Can be used for navigation;
- *DomainElementNode*. Interpreted as a node that corresponds to an element from the domain model.

ProjectTreeNode is the only class that represents the original metamodel of the Navigator framework according to its API. METAclipse project tree engine also does not really need all the various subclasses of the ProjectTreeNode. The subclasses have been introduced in order to ease the creation of the transformations.

There are only two commands specific to the project tree engine that can occur. One is CreateProjectCommand, which is invoked when a METAclipse project is created. It must be interpreted by transformations to initialize the models with some startup data—for example, to initialize the singletons, to set up the default context menus and property editors, to initialize styles, etc. Second is OpenProjectCommand, which is invoked when the project is opened in METAclipse. It can be interpreted by the transformations to carry out some initialization routines required for the opening of the project.

## 4.4 Menu Engine

The menu engine is the simplest engine of all and provides just the functionality needed for the creation of context menus (see Fig. 6, part 2). It uses the standard Eclipse infrastructure for the generation of the menus. Therefore the implementation of the menu engine in METAclipse was even easier than the implementation of the project tree engine.

The menu engine metamodel defines one singleton class, RootMenu (see Fig. 11), which points to the root of the active menu through the "menu" association. If the RootMenu instance does not have this property set, it means that no menu will be displayed. Menu structure is defined by the Menu and MenuItem classes. The Menu class is interpreted by the engine as a menu container (like the root of the context menu or any submenu popping out when an item containing the submenu is selected). Menu consists of menu MenuItem classes, which correspond to the items displayed in the menu. Submenus are shown by the engine only for those MenuItem instances that have the submenu property set.

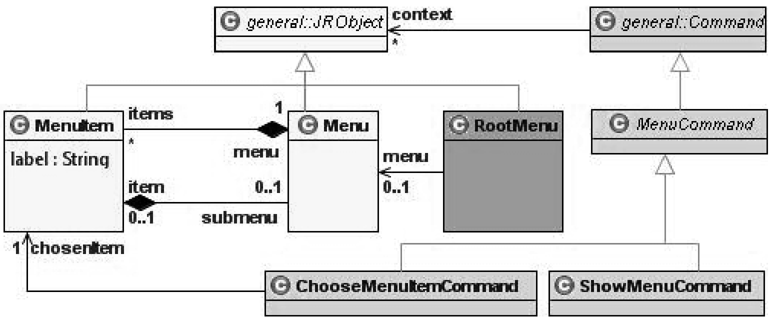


**Fig. 11.** Menu part of the presentation metamodel

Only two specific commands can occur in the menu engine. ShowMenuCommand is invoked when the user right-clicks any node in the project tree or any element in the diagram. Selected JRObject instances (whether tree nodes or diagram elements) will be linked to the ShowMenuCommand through the context association defined in the general Command class. Transformations must react to this command by building the context-sensitive menu (using the context information from the context association) and setting the RootMenu singleton “menu” association to it. The menu engine then will consult the RootMenu singleton to read the menu to be shown.

ChooseMenuItemCommand is written to the repository if the user chooses an item from the menu. Then transformations must carry out the corresponding action. Action can be anything necessary for the chosen menu item, starting from creation of some element up to very complicated tasks like model simplification, compiler invocation for visual DSL languages and so on.

## 4.5 Properties Engine

A very important part of the tools is the properties editor. This editor is used to display and edit various properties of elements displayed in editors. For example, in the UML class diagram editor there is a need to edit the properties of a class or association. In Eclipse property editing is done through a special properties view, which is common to all editors and can be seen at all times (see Fig. 6, part 3). Any time the selection in Eclipse changes, the contents of the properties view are also updated to reflect the properties of the currently selected item. Properties can be arranged in the so-called tabs for better structuring.

The properties view is driven by yet another Eclipse framework, the tabbed properties framework [32], which is used by the properties engine of METAclipse. When the development of METAclipse began, the tabbed properties framework did not provide all the capabilities needed for the tool building platform. Particularly, it was not possible to define the structure of the property sheets at runtime. The framework allowed only definition of what should be displayed in the property sheets during the time of development, and this information had to be compiled in the released plugins.

Because of this, in the beginning the tabbed properties framework was extended to add this functionality. Later, however, the functionality of the framework was widened to include the possibility to define the property sheets dynamically at runtime. This allowed switching to a clean tabbed properties framework without the need to extend its classes. Tabbed properties with dynamic property support will be released in Eclipse 3.4 M3, which is not yet available at the time of this writing. However, Eclipse 3.4 M2 nightly builds already include the new dynamic tabbed property capabilities, so this is what is being used for the time being.

**General Part of the Properties metamodel**

In METAclipse transformations are responsible for building of the property sheets. The select command is issued by editors so that transformations could carry out this task (already introduced in Section 4.1 and shown in Fig. 7). The main part of the property engine metamodel can be seen in Fig. 12.

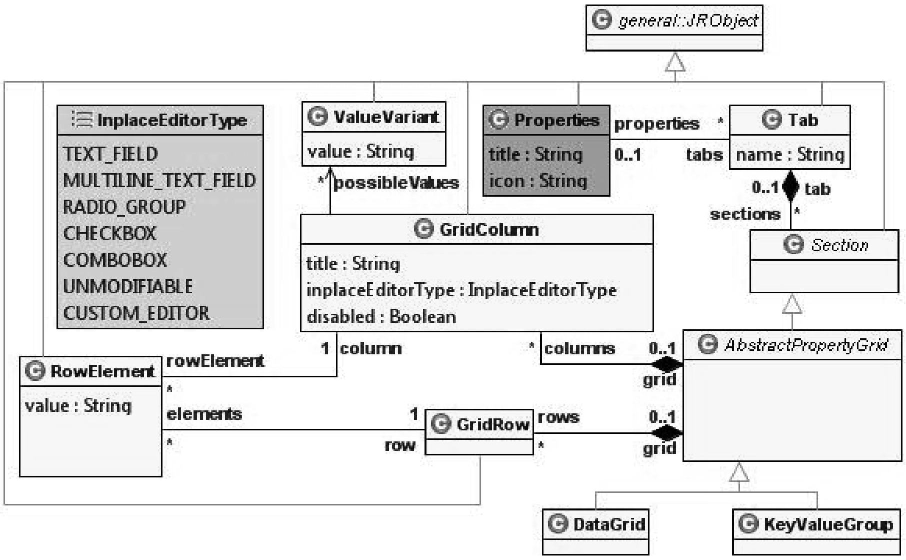
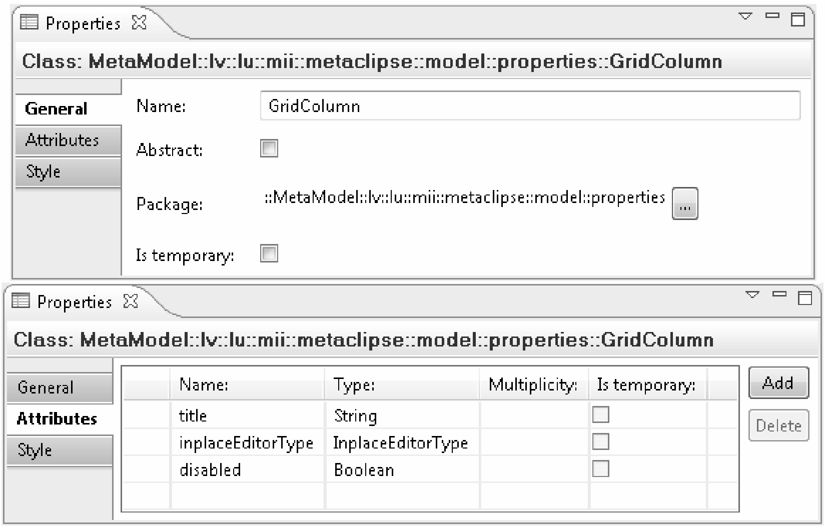


**Fig. 12.** Property part of the presentation metamodel: main classes

The properties singleton is queried every time after the selection of any element and execution of the SelectCommand to read the current state of the properties view. Through this singleton the whole structure describing the contents of the property page can be read. The title and icon attributes of the Properties singleton are used for the title of the properties view. The class Tab represents one property sheet tab and is linked to the Properties singleton through the "tabs" link. The attribute name is the title shown on the tab and is used to name the contents of the tab. For example, both properties views in Fig. 13 consist of three tabs: "General," "Attributes," and "Style."

Every tab in the tabbed properties framework consists of the so-called sections. Sections group the properties shown in the tab in logical groups. The corresponding class in the metamodel is the abstract Section class. The Tab class has a composite association with Section. As many section implementations as necessary could be provided in Eclipse. Two implementations turned out to be most useful in practice:

- A data grid that shows the properties in the form of a table with headers. Such a section can be used for the representation of properties that have one-to-many relationship with the element owning them. An example could be the list of attributes for a class in the UML class diagram (see Fig. 13, bottom);
- A group of key-value pairs that can be used for the representation of properties that have one-to-one relationship with the element owning them. An example application of this can be seen in Fig. 13, at the top, where the "General" tab of the class properties contains various values describing the class—such as "abstract" flag, name of the class, etc.



Class: MetaModel::lv::lu::mii::metaclipse::model::properties::GridColumn

Name: GridColumn
Abstract: ☐
Package: ::MetaModel::lv::lu::mii::metaclipse::model::properties
Is temporary: ☐

Class: MetaModel::lv::lu::mii::metaclipse::model::properties::GridColumn

| Name: | Type: | Multiplicity: | Is temporary: |
|---|---|---|---|
| title | String | | ☐ |
| inplaceEditorType | InplaceEditorType | | ☐ |
| disabled | Boolean | | ☐ |

**Fig. 13.** KeyValueGroup properties section implementation (at the top) and DataGrid implementation (at the bottom) in action

These two kinds of section are implemented as part of the properties engine in METAclipse. DataGrid and KeyValueGroup classes in the metamodel (see Fig. 12) correspond to the data grid and key-value pair group section implementations, respectively. Both section implementations use the same metamodel structure for the description of their contents. This turned out to be particularly useful for the development of transformations, as it allowed a uniform design of the property-building transformations.

The structure used for the two section implementations consists of three main classes: GridColumn, GridRow, and RowElement. In case of the DataGrid section implementation, GridColumn corresponds to the table column. The title attribute will be shown as the header of the table. Attribute inplaceEditorType denotes the kind of editor that will be used for editing of the data found in the column. Possible values are defined by the InplaceEditorType enumeration and include such editors as text field, combo-box, checkbox, radio group etc. A special kind of editor is CUSTOM_EDITOR, which means that an external dialog has to be shown instead of in-place editor. This will be discussed in more detail below. For editing of the combo-box or radio group fields, additionally a set of possible values must be defined. This is done through the possibleValues association from the GridColumn class to the ValueVariant class.

The GridRow class corresponds to one row in the grid. The DataGrid class will hold an ordered reference to all row classes through "rows" association. Actual data of the table cells is represented by the RowElement class. The association "column" of this class will define what column the row element belongs to, while the association "row" will indicate in which row it should be displayed.

As stated before, the KeyValueGroup section implementation uses the same model. To understand how the structure is applied to the KeyValueGroup implementation, we can imagine that this implementation is nothing more than DataGrid with one row, which is displayed vertically instead of horizontally. So, there will be exactly one GridRow instance and each GridColumn instance will correspond to the label of one key-value pair in the KeyValueGroup section (for example, “name” or “abstract” at the property view shown at the top of Fig. 13). RowElement instances correspond to the value part of key-value pairs, i.e., the values of the properties that can be edited.

**Property Editors and Commands**

Not all properties can be edited directly in the properties view—some require more advanced editing capabilities. For example, editing of a property denoting a color or a font requires a proper color dialog to be shown. Also properties that must be chosen from a list with lots of entries are inconvenient to be edited with a simple combo-box. The metamodel of the properties engine contains an additional set of classes for the definition of external editors (see Fig. 14).

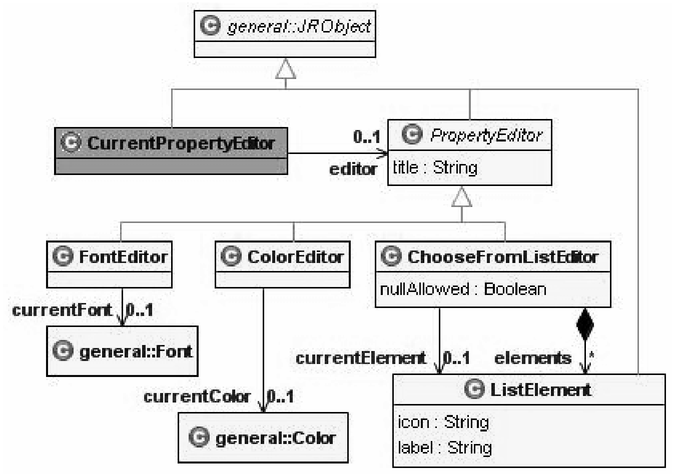


**Fig. 14.** Property part of the presentation metamodel: editor classes

Theoretically it would also be possible to create a universal dialog engine, so that any kind of dialogs could be constructed. However, it would require very large effort to build such engine. Therefore, it was decided to build concrete dialogs for different tasks. In the metamodel, a common superclass PropertyEditor is introduced for all dialogs. Three implementations are provided by the engine: the FontEditor class representing the font dialog, the ColorEditor class representing the color dialog and the ChooseFromListEditor representing the dialog for showing large lists.

If an external dialog is needed for a particular column, the inplaceEditorType attribute of the GridColumn instance must be set to CUSTOM_EDITOR. The engine will then display a button for invoking the external editor. If the button is pressed, the ShowEditorCommand (see Fig. 15) will be invoked and transformations will have to construct the dialog to be shown. The editor constructed then has to be linked to the CurrentPropertyEditor singleton, because the engine will consult this singleton to find which editor to show.

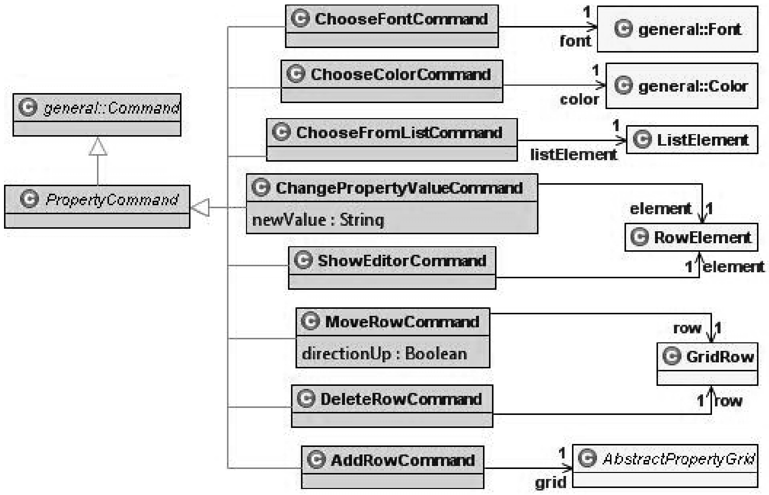


**Fig. 15.** Property part of the presentation metamodel: command classes

After showing the dialog and having the user choose something, the corresponding command is executed, containing the information about user actions in the dialog. Thus, for the font dialog, ChooseFontCommand is executed with the chosen font attached through the font association. Similarly, ChooseColorCommand is executed after choosing any color from the color dialog and ChooseFromListCommand, after choosing some list item from the list dialog.

The remaining commands not yet discussed are ChangePropertyValueCommand, which is invoked when any of in-place property editors is used to change the value of some property; MoveRowCommand, which is used to change the order of the DataGrid rows; DeleteRowComand, which deletes DataGrid rows; and AddRowCommand, which creates new DataGrid rows.

## 4.6 Graph Diagram Engine

The most important of all engines is the graph diagram engine. This engine is used for visual graph diagram editing (see Fig. 6, part 4). Eclipse technologies used for the graph diagram engine are the Graphical Editing Framework GEF [13] and the Graphical Modeling Framework GMF [14]. GMF is the most popular metamodel-based graphical tool building platform for Eclipse. GMF utilizes EMF (Eclipse Modeling Framework) and GEF (Graphical Editing Framework) technologies. EMF is used for model management and GEF, for graphical user interface.

GMF uses a static-mapping-driven approach. It defines a set of metamodels: graphical (presentation), tooling and mapping metamodels. In addition, it uses ECore as the domain metamodel. The graphical metamodel defines the graphical element types. The tooling metamodel defines the palette and menus. The mapping metamodel defines the mapping possibilities between the models. To build an editor in GMF, the domain, graphical, tooling and mapping models are defined, then generation is performed and manual code in Java added. An analysis of the GMF and a comparison of the static-mapping-driven approach as such to the transformation-driven approach described here are given in the paper "Building Tools by Model Transformations in Eclipse" [24].

The graphical (presentation) metamodel is well adapted to the generation step in GMF, but cannot be used directly by the transformation approach. The same situation is true for the tooling metamodel. Therefore, nothing of the GMF definition part can actually be reused in the proposed METAclipse approach. As a consequence, there are no explicit graphical element types to be used by transformations.

Fortunately, the GMF runtime [34] uses another metamodel—the notation metamodel. This metamodel describes graphical instances in the runtime—nodes, edges, compartments and labels (exactly, the layer required by transformations to build graphical objects dynamically). In fact, the GMF runtime is a graphical engine for Eclipse, significantly extending GEF in the direction required for diagram building. This allows at least partial reuse of the GMF runtime in METAclipse.

The created graph diagram engine does not fall back from professional Eclipse-based tools like RSA [35] in its diversity of features and graphical quality. The developed metamodel, presented further, allows relatively simple control of quite advanced graphical structures and behavior. Although the graph diagram engine was the most difficult to implement, the reuse of GMF runtime and GEF components allowed keeping the required effort for building it reasonably low.

## The General Part of the Graph Diagram Engines Metamodel

The main part of the graph diagram engines metamodel in METAclipse is quite similar to the GMF notation metamodel. However, it is not the same. It has been made more accessible for the transformations and more easily usable in various contexts of METAclipse (see Fig. 16).

The root element corresponding to the actual diagram is the Diagram class. It consists of DiagramElement class instances, which can be either Node or Edge. Node class instances correspond to the graph diagram nodes and Edge instances correspond to edges. Note that Diagram itself is also a kind of node. This allows the use of sub-diagrams. The Diagram element defines the general attributes of all elements, such as line style and width. Node defines the general attributes of all kinds of nodes. The Edge class defines the routing of the edges via the routing style attribute and association with Bendpoint instances. Routing style defines how the line should be laid out on the diagram and Bendpoint instances define the layout constraints.

Besides Diagram itself, the nodes are divided into two categories—SimpleNode and CompositeNode. SimpleNode denotes the nodes that may not contain any children. CompositeNode, on the other hand, may contain children. Theoretically, Diagram also is a composite node. However, because of its specific nature, it is not in the class hierarchy of the composite nodes.

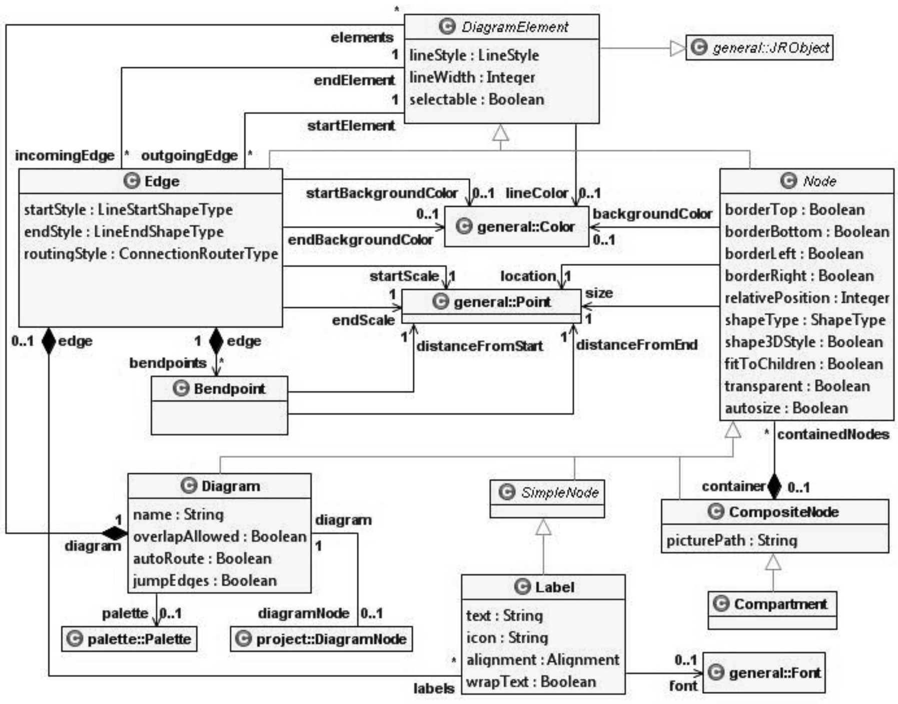


**Fig. 16.** Graph diagram part of the presentation metamodel without commands and palette

There is just one kind of SimpleNode type—the Label class. Labels are static text elements that may also display an icon. CompositeNode is not abstract, thus it may be instantiated itself, but there is also one special type of the composite node, i.e. Compartment. Compartment is a kind of grouping, used, for example for class diagrams in UML [10].

Just as an example, let us consider the UML class Diagram (like the one in Fig. 16). Diagram itself is represented with the Diagram class instance. It consists of CompositeNode-s, which in turn consist of one label for class icon and name, one compartment with labels for attributes, and one compartment with labels for operations (operations not shown in the figure). Associations are edges with different sets of attribute values for different kinds of associations. These are the bricks for building class diagrams in the METAclipse framework.

In Fig. 17 the command part of the graph diagram engines metamodel is shown. There are just four commands specific to the graph diagram engine:

- CreateEdgeCommand, used for creation of the edges;
- CreateNodeCommand, used for the creation of the nodes;
- MoveNodeCommand, used for the semantic moving of the nodes (in case the node is dropped in another node, for example);
- RedirectEdgeCommand, used for relocating the edge start or end to a different node.

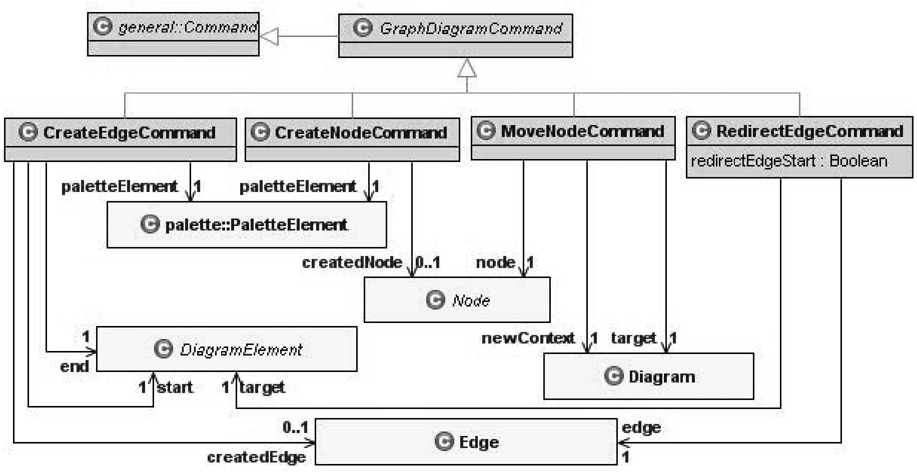


**Fig. 17.** Graph diagram command part of the presentation metamodel

Additionally, there are some already discussed common commands accessible in graph diagram engine, like NavigateCommand, SelectCommand, etc. These are used for the tasks that are common to more than just one METAclipse engine.

## Palette Part of the Graph Diagram Engines Metamodel

The metamodel for description of the palettes has been separated from the graph diagram metamodel as it could be reused also for other diagram kinds. Fig. 18 shows the palette part of the graph diagram engines metamodel.

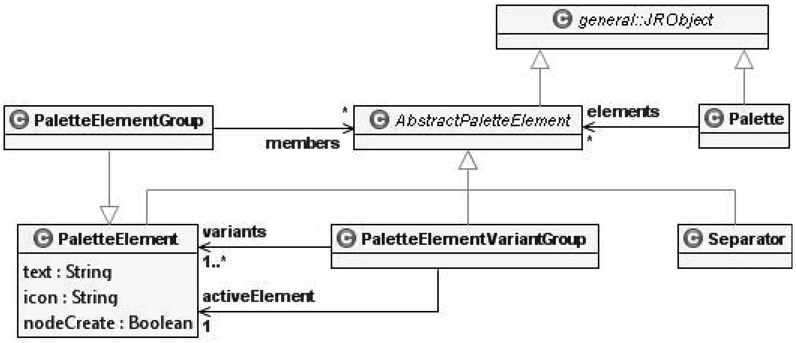


**Fig. 18.** Palette part of the presentation metamodel

The structure of the palette metamodel represents the possibilities to build palette in Eclipse. The Palette class represents the palette itself. It consists of AbstractPaletteElement instances. There are four kinds of palette elements that can be used:

- PaletteElement—a simple palette element with an icon and an label;
- Separator—a separating line;
- PaletteElementGroup—a container for similar palette elements grouped together. Groups cannot be nested and may be shown or hidden on user request;
- PaletteElementVariantGroup—a special kind of palette element group used for displaying the variants of the same palette element. Visually this group is shown as a normal palette element; however, it allows the switching to another palette element variant upon user request.

## 5 Transformation Structure

Describing the transformation part of the framework is not the objective of this paper. Therefore transformations will be discussed very briefly. As already stated, transformations in METAclipse are written in the MOLA model transformation language [28]. The MOLA compiler uses another model transformation language developed at UL IMCS, i.e. Lx language series [33]. Lx then is compiled to efficient C++ code, which is able to work with large models in fractions of a second. Only by accomplishing such performance is it possible to satisfy all needs of METAclipse, as every semantic user operation results in non-trivial transformations.

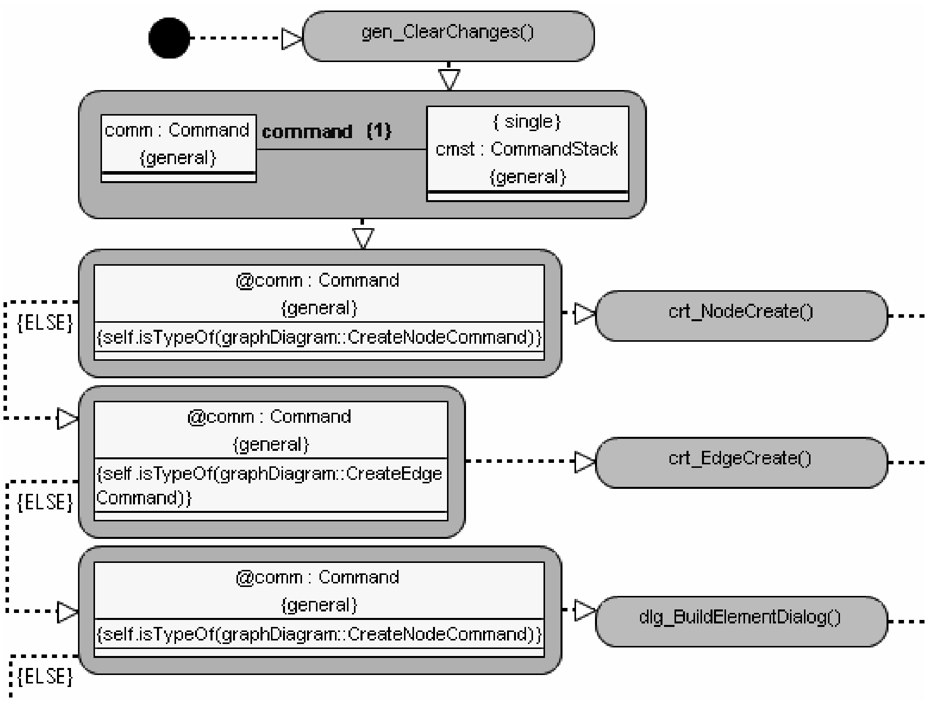


**Fig. 19.** Example of a MOLA transformation: a small excerpt of command handling procedure

In METAclipse there is only one entry point for the transformations, i.e., it is always the same transformation that gets called when executing a command. It is then the task of the transformation to call different procedures that implement model transformations that correspond to the particular command. In Fig. 19, one small part of the command parsing or main transformation is shown. It serves as an example of what MOLA transformations look like visually and at the same time displays how the single main transformation calls the sub-transformations in order to react to particular commands.

The transformation library is actually the key component that finally defines a concrete DSL tool created with METAclipse. Different tools built in METAclipse will have different transformation libraries. In order to build a tool, the toolsmith must first define the domain metamodel. Then he/she must link the domain metamodel to the presentation metamodel described in the previous section through model transformations. The presentation metamodel may be augmented for the transformation needs with new links or attributes. The only restriction is that existing classes, attributes and associations must remain intact. Finally, if necessary, the toolsmith must implement various functions through transformations that are needed for a particular tool.

## 6 Future Work

Currently METAclipse already has all the functionality needed for successful building of rich DSL editors. So, for example, the MOLA editor, built with METAclipse, has proved to be a powerful tool for editing MOLA transformations and is being successfully used. There is still a lot of work to be done in order to make the creation of transformations easier, so that tools could be built with much less effort. This would include generalization of common transformations, creation of reusable transformation frameworks (small frameworks for properties, styles, etc.), incorporation of the static mapping approach, definition of helper-functions, etc. Analysis of the transformation part, however, is beyond the scope of this paper.

Of course, there are also tasks to be done in order to make the METAclipse presentation framework (engines) more convenient and easier to use. Additional features could be implemented to enable more functionality for the tools. Some of these tasks are:

- Creating a more advanced property engine in order to allow building of more customized property pages. Currently the layout and contents of property sheets are very rigid and only a limited number of various controls can be used. There are cases when it is necessary to have richer property editors;
- Introducing the possibility for transformations to impact the engines, meaning that some special commands could be issued from transformations, which then would be interpreted by engines. This would be necessary, for example, to provide interactive debugging support for DSL editors.
- Adding possibilities to include animations. This would also be particularly useful for debugging.

- Implementing XMI import/export for domain part of the models. EMF already has the functionality needed for serialization and de-serialization of the models to XMI, however, currently only the presentation model is loaded via wise objects.
- Enhancement of the current graph diagram engine to allow more advanced constructs, such as swimlanes and pins used in UML activity diagrams.
- Creation of new engines for editing of other kinds of diagrams.

The named tasks represent just several areas in which it is already thought of to extend the METAclipse framework. The effort needed to implement the features listed above is relatively small compared to what has been already invested to provide the basic functionality of METAclipse, and all these tasks can be considered as "extras." Of course, the number of new features that could be added and that could be useful for the toolsmiths, as well as for tool users, is virtually unlimited.

## References

1. Smolander, K., Martiin, P., Lyytinen, K., Tahvanainen, V-P.: Metaedit—a flexible graphical environment for methodology modeling. Springer-Verlag, 1991.
2. Ebert, J., Suttenbach, R., Uhe, I.: Meta-CASE in Practice: a Case for KOGGE. Proceedings of the 9th International Conference, CAiSE'97, Barcelona, Spain, 1997, pp. 203–216.
3. DOME Users Guide, http://www.htc.honeywell.com/dome/support.htm
4. Karsai G.: A Configurable Visual Programming Environment: A Tool for Domain-Specific Programming, IEEE Computer Society Press, pp. 36–44, 1995.
5. MetaEdit+, http://www.metacase.com/
6. Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason IV, C., Nordstrom, G., Sprinkle, J., Volgyesi, P.: The Generic Modeling Environment. Workshop on Intelligent Signal Processing, Budapest, Hungary, May 17, 2001.
7. de Lara, J., Vangheluwe, H., Alfonseca, M.: Meta-Modeling and Graph Grammars for Multi-Paradigm Modeling in AToM3. Software and System Modeling, 3(3), 2004, pp. 194–209.
8. Steven Kelly, Kalle Lyytinen, Matti Rossi: MetaEdit+ A fully configurable multi-user and multi-tool CASE and CAME environment Lecture Notes in Computer Science, Volume 1080, Proceedings of the 8th International Conference on Advances Information System Engineering, pp. 1–21, Springer-Verlag, 1996.
9. Meta-Object Facility (MOF), http://www.omg.org/mof/
10. OMG, Unified Modeling Language: Superstructure, version 2.0, http://www.omg.org/docs/formal/05-07-04.pdf
11. Celms, E., Kalnins, A., Lace, L.: Diagram definition facilities based on metamodel mappings. Proceedings of the 18th International Conference, OOPSLA'2003, Workshop on Domain-Specific Modeling, Anaheim, California, USA, October 2003, pp. 23–32.
12. Eclipse Modeling Framework (EMF, Eclipse Modeling subproject), http://www.eclipse.org/emf/
13. Graphical Editor Framework (GEF, Eclipse Tools subproject), http://www.eclipse.org/gef/
14. Graphical Modeling Framework (GMF, Eclipse Modeling subproject), http://www.eclipse.org/gmf/

15. N. Zhu1, J. Grundy and J. Hosking. Pounamu: a meta-tool for multi-view visual language environment construction. Proc. IEEE Symposium on Visual Languages and Human Centric Computing (VLHCC'04), pp. 254–256, 2004.
16. The Generic Eclipse Modeling System (GEMS), http://www.eclipse.org/gmt/gems/
17. S. Cook, G. Jones, S. Kent and A. C. Wills. Domain-Specific Development with Visual Studio DSL Tools. Addison-Wesley, 2007.
18. Ermel, C., Ehrig, K., Taentzer, G., Weiss, E.: Object Oriented and Rule-based Design of Visual Languages using Tiger. Proceedings of GraBaTs'06, 2006, pp. 12.
19. I. Rath, D. Varro. Challenges for advanced domain-specific modeling frameworks. Proc. of Workshop on Domain-Specific Program Development (DSPD), ECOOP 2006, France.
20. Taentzer, G: AGG: A Graph Transformation Environment for Modeling and Validation of Software. Application of Graph Transformations with Industrial Relevance (AGTIVE'03), Vol. 3062, Springer LNCS, 2004.
21. Visual Automated Model Transformations (VIATRA2), GMT subproject, Budapest University of Technology and Economics, http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/VIATRA2/index.html
22. Fujaba. Universitat Paderborn, Institut fur Informatik. http://wwwcs.uni-paderborn.de/cs/fujaba/documents/user/manuals/FujabaDoc.pdf
23. C. Amelunxen, A. Königs, T. Rötschke, A. Schürr: MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. Model Driven Architecture—Foundations and Applications: Second European Conference, Lecture Notes in Computer Science, Vol. 4066, pp. 361–375, Springer 2006.
24. Kalnins, A., Vilitis, O., Celms, E., Kalnina, E., Sostaks, A., Barzdins, J.: Building Tools by Model Transformations in Eclipse. Proceedings of DSM'07 workshop of OOPSLA 2007, Montreal, Canada, Jyväskylä University Printing House, 2007, pp. 194–207.
25. Barzdins, J., Zarins, A., Cerans, K., Kalnins, A., Rencis, E., Lace, L., Liepins, R., Sprogis, A.: GrTP: Transformation Based Graphical Tool Building Platform. Proceedings of MODELS 2007, MDDAUI 2007 workshop, Nashville, Tennessee, USA, September 30–October 5, 2007, pp. 4.
26. Kalnins, A., Barzdins, J., Celms, E.: Model Transformation Language MOLA. Proceedings of MDAFA 2004, Vol. 3599, Springer LNCS, 2005, pp. 62–76.
27. ReDSeeDS. Requirements Driven Software Development System. European FP6 IST project. http://www.redseeds.eu/, 2007.
28. UL IMCS, MOLA pages, http://mola.mii.lu.lv/
29. Barzdins, J., Barzdins, G., Balodis, R., Cerans, K., Kalnins, A., Opmanis, M., Podnieks, K.: Towards Semantic Latvia. Proceedings of Seventh International Baltic Conference on Databases and Information Systems, Communications, Vilnius, Lithuania, O. Vasileckas, J. Eder, A. Caplinskas (Eds.), Vilnius, Technika, 2006, pp. 203–218.
30. Java Native Interface Specification, http://java.sun.com/j2se/1.5.0/docs/guide/jni/spec/jniTOC.html
31. Eclipse Model To Text project, http://www.eclipse.org/modeling/m2t/
32. The Eclipse Tabbed Properties View, http://www.eclipse.org/articles/Article-Tabbed-Properties/tabbed_properties_view.html
33. Lx Transformation Language Set, http://Lx.mii.lu.lv/, 2007.
34. R. Gronback, Build Better Graphical Editors with the Graphical Modeling Framework, Slides, Eclipseworld 2006, http://wiki.eclipse.org/images/0/08/Gronback_EclipseWorld2006_GMF.ppt.zip
35. Rational Software Architect (RSA), http://www-306.ibm.com/software/awdtools/architect/swarchitect/

# MATHEMATICAL FOUNDATIONS

LATVIJAS UNIVERSITĀTES RAKSTI. 2008, 733. sēj.:
DATORZINĀTNE UN INFORMĀCIJAS TEHNOLOĢIJAS 215.–238. lpp.

# Quantum Query Algorithm Constructions for Computing *AND*, *OR* and *MAJORITY* Boolean Functions

Alina Vasiljeva [*]

Institute of Mathematics and Computer Science University of Latvia
Raiņa bulvāris 29, Rīga, LV-1459, Latvia
Alina.Vasiljeva@gmail.com

**Abstract.** Quantum algorithms can be analyzed in a query model to compute Boolean functions where input is given in a black box and the aim is to compute function value for arbitrary input using as few queries as possible. We concentrate on quantum query algorithm designing tasks in this paper. The main aim of the research was to find new efficient algorithms and develop general algorithm designing techniques. First, we present several exact quantum query algorithms for certain problems that are better than classical counterparts. Next, we introduce algorithm transformation methods that allow significant enlarging of exactly computable functions sets. Finally, we propose quantum algorithm designing methods. Given algorithms for the set of sub-functions, our methods use them to design a more complex one, based on algorithms described before. Methods are applicable for input algorithms with specific properties and preserve acceptable error probability and number of queries. Methods offer constructions for computing *AND*, *OR* and *MAJORITY* kinds of Boolean functions.

**Keywords.** Quantum computing, quantum query algorithms, complexity theory, Boolean functions, algorithm design.

## 1 Introduction

Let $f(x_1, x_2, ..., x_n) : \{0,1\}^n \rightarrow \{0,1\}$ be a Boolean function. We have studied the query model, where a black box contains the input $(x_1, x_2, ..., x_n)$ and can be accessed by questioning $x_i$ values. The goal here is to compute the value of the function. The complexity of a query algorithm is measured by the number of questions it asks. The classical version of this model is known as *decision trees* [1]. Quantum query algorithms can solve certain problems faster than classical algorithms. The best-known exact quantum algorithm was designed for *PARITY* function with $n/2$ questions vs. $n$ questions required by classical algorithm [2,3].

The problem of quantum algorithm construction is not that easy. Although there is a large amount of lower and upper bound estimations of quantum algorithm complexity [2, 6, 7], examples of non-trivial and original quantum query algorithms are very few. Moreover, there is no special technique described to build a quantum algorithm for a certain function with complexity defined in advance.

[*] Research supported by the European Social Fund

Most probably it would take a lot of time even for experienced quantum computation specialist to construct an efficient query algorithm, for example, for such functions:

$$F_4(x_1,x_2,x_3,x_4) = \neg(x_1 \oplus x_2) \wedge \neg(x_3 \oplus x_4)$$

$$F_6(X) = (\neg(x_1 \oplus x_2) \wedge \neg(x_2 \oplus x_3)) \wedge (\neg(x_4 \oplus x_5) \wedge \neg(x_5 \oplus x_6))$$

or

$$F_{10}(X) = (f_1 \wedge f_2 \wedge f_3) \vee (f_1 \wedge f_2 \wedge f_4) \vee (f_1 \wedge f_3 \wedge f_4) \vee (f_2 \wedge f_3 \wedge f_4)\text{ ,where}$$
$$f_1 = (x_1 \oplus x_2) \vee (x_3 \oplus x_4);\ f_2 = x_5 \oplus x_6;\ f_3 = \neg(x_7 \oplus x_8) \wedge \neg(x_8 \oplus x_9);\ f_4 = \neg x_{10}$$

In our work we have tried to develop general constructions and approaches for computing Boolean functions in quantum query settings.

Boolean functions are widely adopted in real life processes, that is the reason why our capacity to build a quantum algorithm for an arbitrary function appears to be extremely important. While working on common techniques, we are trying to collect examples of efficient quantum algorithms to build up a base for powerful computation using the advantages of the quantum computer.

The paper is organized as follows. Section 2 consists of theoretical background and definitions. In section 3 two exact quantum query algorithm are presented, which will be used as a base in further sections. In section 4 we present three algorithm transformation methods. Section 5 contains the major part of results - algorithm constructions for computing *AND*, *OR* and *MAJORITY* kinds of Boolean functions. Finally, the summary of results is given in section 6.

## 2 Notation and Definitions

Let $f(x_1,x_2,...,x_n):\{0,1\}^n \to \{0,1\}$ be a Boolean function. We use $\oplus$ to denote XOR operation (exclusive OR). We use $\overline{f}$ for the function 1 - *f*. We also use abbreviation QQA for "quantum query algorithm".

### 2.1 Quantum Computing

We apply the basic model of quantum computing. For more details see textbooks by Gruska [4] and Nielsen and Chuang [5].

An *n*-dimensional quantum pure state is a vector $|\psi\rangle \in C^n$ of norm 1. Let $|0\rangle,|1\rangle,..,$ $|n\text{-}1\rangle$ be an orthonormal basis for $C^n$. Then, any state can be expressed as $|\psi\rangle = \sum_{i=0}^{n-1} a_i |i\rangle$ for some $a_i \in C$. Since the norm of $|\psi\rangle$ is 1, we have $\sum_{i=0}^{n-1} |a_i|^2 = 1$. States $|0\rangle,|1\rangle,\ldots,|n\text{-}1\rangle$ are called *basic states.* Any state of the form $\sum_{i=0}^{n-1} a_i |i\rangle$ is called a *superposition* of $|0\rangle,\ldots,|n\text{-}1\rangle$. The coefficient $a_i$ is called an *amplitude* of $|i\rangle$.

The state of a system can be changed using *unitary transformations*. Unitary transformation $U$ is a linear transformation on $C^n$ that maps vector of unit norm to vectors of unit norm.

The simplest case of quantum measurement is used in our model. It is the full measurement in the computation basis. Performing this measurement on a state $|\psi\rangle = a_0|0\rangle + \ldots a_k|k\rangle$ gives the outcome $i$ with probability $|a_i|^2$. The measurement changes the state of the system to $|i\rangle$ and destroys the original state $|\psi\rangle$.

## 2.2 Query Model

Query algorithm is a model for computing Boolean functions. In this model, a black box contains the input $(x_1, x_2, ..., x_n)$ and can be accessed by questioning $x_i$ values. Query algorithm must be able to determine the value of a function correctly for arbitrary input contained in a black box. The complexity of the algorithm is measured by the number of queries to the black box which it uses. The classical version of this model is known as *decision trees*. For details, see the survey by Buhrman and de Wolf [1].

We consider computing Boolean functions in the quantum query model. For more details, see the survey by Ambainis [6] and textbooks by Gruska [4] and de Wolf [2]. A quantum computation with $T$ queries is a sequence of unitary transformations:

$$U_0 \to Q_0 \to U_1 \to Q_1 \to \ldots \to U_T \to Q_{T-1} \to U_T$$

$U_i$'s can be arbitrary unitary transformations that do not depend on the input bits $x_1, x_2, ..., x_n$. $Q_i$'s are query transformations. Computation starts in the state $|\vec{0}\rangle$. Then we apply $U_0$, $Q_0$, ..., $Q_{T-1}$, $U_T$ and measure the final state.

There are several different, but equally acceptable ways to define quantum query algorithms [2]. The most important consideration is to choose an appropriate definition for the query black box, defining a way of asking questions and receiving answers from the oracle.

Next we will precisely describe the full process of quantum query algorithm definition and notation used in this paper.

Each quantum query algorithm is characterized by the following parameters:

1) *Unitary transformations*

All unitary transformations and the sequence of their application (including the query transformation parts) should be specified. Each unitary transformation is a unitary matrix.

Here is an example of an algorithm sequence specification with $T$ queries:

$$|\vec{0}\rangle \to U_0 \to Q_1 \to \ldots \to Q_{N-1} \to U_N \to [QM],$$

where $|\vec{0}\rangle$ is initial state, [QM] – quantum measurement.

For convenience we will use *bra* notation to describe state vectors and algorithm flows. Quantum mechanics employs the following notation for state vectors [5]:

$$\textit{Ket}\text{ notation: } |\psi\rangle = \begin{pmatrix} \alpha_1 \\ \dots \\ \alpha_n \end{pmatrix} \quad \textit{Bra}\text{ notation: } \langle\psi| = |\psi\rangle^+ = (\alpha_1^*, \quad \dots, \quad \alpha_n^*)$$

Algorithm designed in *bra* notation can be converted to *ket* notation by replacing each unitary transformation matrix with its adjoint matrix (conjugate transpose):

Quantum query algorithm flow in *bra* notation: $\langle\psi| = \langle\vec{0}|U_0 Q_0 \dots Q_{N-1} U_N$

Quantum query algorithm flow in *ket* notation: $|\psi\rangle = U_N^+ Q_{N-1}^+ \dots Q_0^+ U_0^+ |\vec{0}\rangle$

2) *Queries*

We use the following definition of query transformation: if input is a state $|\psi\rangle = \sum_i a_i |i\rangle$ , then the output is $|\phi\rangle = \sum_i (-1)^{xk} a_i |i\rangle$, where we can arbitrary choose variable assignment $x_k$ for each amplitude $\alpha_i$. Assume we have a quantum state with *m* amplitudes $\langle\psi| = (\alpha_1, \alpha_2, \dots, \alpha_m)$. For the *n* argument function, we define a query as $QQ_i = (\alpha_1 \equiv k_1, \dots, \alpha_m \equiv k_m)$, where *i* is the number of question and $k_j \in \{1..n\}$ is the number of queried variable for *j*-th amplitude (*QQ* abbreviates "quantum query"). If $x_{k_j} = 1$, a query will change the sign of the *j*-th amplitude to the opposite sign; in other case, the sign will remain as-is. Unitary matrix that corresponds to query transformation $QQ_i = (\alpha_1 \equiv k_1, \dots, \alpha_m \equiv k_m)$ is:

$$QQ_i = \begin{pmatrix} (-1)^{X_{k1}} & 0 & \dots & 0 \\ 0 & (-1)^{X_{k2}} & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & (-1)^{X_{km}} \end{pmatrix}$$

3) *Measurement*

Each basic state of a quantum system corresponds to the algorithm output. We assign a value of a function to each output. We denote it as $QM = (\alpha_1 \equiv k_1, \dots, \alpha_m \equiv k_m)$, where $k_i \in \{0,1\}$ (*QM* abbreviates "quantum measurement"). The result of running algorithm on input *X* is *j* with a probability that equals the sum of squares of all amplitudes, which corresponds to outputs with value *j*.

Very convenient way of quantum query algorithm representation is a graphical picture and we will use this style when describing designed quantum query algorithms.

### 2.3 Query Algorithm Complexity

The complexity of a query algorithm is based on the number of questions it uses to determine the value of a function on worst-case input.

The *deterministic complexity* of a function *f*, denoted by *D*(*f*), is the maximum number of questions that must be asked on any input by a deterministic algorithm for *f* [1].
The sensitivity of $f$ on input $(x_1,x_2,\dots,x_n)$ is the number of variables $x_i$ with the following property: $f(x_1,\dots,x_i,\dots,x_n)\neq f(x_1,\dots,1\text{-}x_i,\dots,x_n)$. The sensitivity of $f$ is the maximum sensitivity of all possible inputs. It has been proved that $D(f)\geq s(f)$ [1].

A quantum query algorithm *computes f exactly* if the output equals *f*(*x*) with a probability 1, for all $x\in\{0,1\}^n$. Complexity is denoted by $Q_E(f)$ [1].

A quantum query algorithm *computes f with bounded-error* if the output equals *f*(*x*) with probability $p>1/2$, for all $x\in\{0,1\}^n$. Complexity is denoted by $Q_P(f)$ [1].

## 3 Basic Exact Quantum Query Algorithms

In this section we present two basic exact quantum query algorithms, which will be used as a base for construction methods in further sections.
First algorithm computes 3-argument Boolean function, but second one computes 4-argument Boolean function. Both algorithms are interesting; first of all, because they are better than the best possible classical algorithms. Secondly, algorithms satisfy specific properties, which make them useful for computing more complex Boolean functions.

### 3.1 3-Variable Function with 2 Queries

In this section we present quantum query algorithm for 3-variable Boolean function that saves one query comparing to the best possible classical deterministic algorithm.

**Problem:** *Check if all input variable values are equal.*

Possible real life application is, for example, automated voting system, where statement is automatically approved only if all participants voted for acceptance/rejection equally. We provide solution for 3-party voting routine. We reduce a problem to computing the following Boolean function defined by the logical formula: $EQUALITY_3(X)=\neg(x_1\oplus x_2)\wedge\neg(x_2\oplus x_3)$.

**Deterministic complexity:** $D(EQUALITY_3)=3$, by sensitivity on any accepting input.

**Algorithm 1.** Exact quantum query algorithm for $EQUALITY_3$ is presented in Figure 1. Each horizontal line corresponds to the amplitude of the basic state. Computation starts with amplitude distribution $\left\langle\vec{0}\right|=(1,0,0,0)$. Three large rectangles correspond to the 4x4 unitary matrices ($U_0$, $U_1$, $U_2$). Two vertical layers of circles specify the queried variable order for each query ($Q_0$, $Q_1$). Finally, four small squares at the end of each horizontal line define the assigned function value for each output.

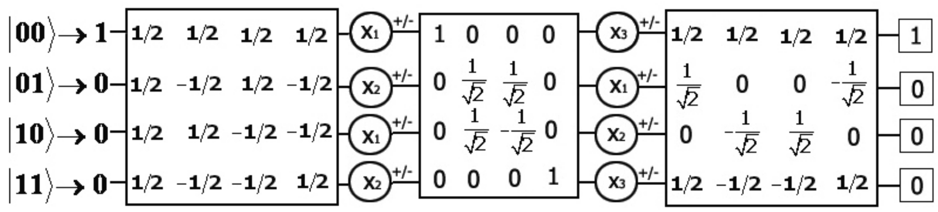


**Fig. 1.** Exact Quantum Query Algorithm for *EQUALITY*$_3$

We show the computation process for accepting input X=111:

$$\langle\psi| = (1/2,\ 1/2,\ 1/2,\ 1/2)Q_0U_1Q_1U_2 = (-1/2,-1/2,-1/2,-1/2)U_1Q_1U_2 =$$
$$= \left(-1/2,\ -1/\sqrt{2},\ 0,\ -1/2\right)Q_1U_2 = \left(1/2,\ 1/\sqrt{2},\ 0,\ 1/2\right)U_2 = \mathbf{(1,0,0,0)}$$
$$\Rightarrow \text{[ACCEPT]}$$

Table 1 shows computation process for each possible input. Processing result always equals *EQUALITY*$_3$ value with probability $p$=1.

**Table 1.** Quantum Query Algorithm Computation Process for *EQUALITY*$_3$

| $X$ | after $\langle\vec{0}\vert U_0Q_0$ | after $\langle\vec{0}\vert U_0Q_0U_1Q_1$ | final state | result |
|---|---|---|---|---|
| 000 | $\left(\frac{1}{2},\frac{1}{2},\frac{1}{2},\frac{1}{2}\right)$ | $\left(\frac{1}{2},\frac{1}{\sqrt{2}},0,\frac{1}{2}\right)$ | (1,0,0,0) | **1** |
| 001 | $\left(\frac{1}{2},\frac{1}{2},\frac{1}{2},\frac{1}{2}\right)$ | $\left(-\frac{1}{2},\frac{1}{\sqrt{2}},0,-\frac{1}{2}\right)$ | (0,0,0,-1) | **0** |
| 010 | $\left(\frac{1}{2},-\frac{1}{2},\frac{1}{2},-\frac{1}{2}\right)$ | $\left(\frac{1}{2},0,\frac{1}{\sqrt{2}},-\frac{1}{2}\right)$ | (0,0,1,0) | **0** |
| 011 | $\left(\frac{1}{2},-\frac{1}{2},\frac{1}{2},-\frac{1}{2}\right)$ | $\left(-\frac{1}{2},0,\frac{1}{\sqrt{2}},\frac{1}{2}\right)$ | (0,-1,0,0) | **0** |
| 100 | $\left(-\frac{1}{2},\frac{1}{2},-\frac{1}{2},\frac{1}{2}\right)$ | $\left(-\frac{1}{2},0,\frac{1}{\sqrt{2}},\frac{1}{2}\right)$ | (0,-1,0,0) | **0** |
| 101 | $\left(-\frac{1}{2},\frac{1}{2},-\frac{1}{2},\frac{1}{2}\right)$ | $\left(\frac{1}{2},0,\frac{1}{\sqrt{2}},-\frac{1}{2}\right)$ | (0,0,1,0) | **0** |
| 110 | $\left(-\frac{1}{2},-\frac{1}{2},-\frac{1}{2},-\frac{1}{2}\right)$ | $\left(-\frac{1}{2},\frac{1}{\sqrt{2}},0,-\frac{1}{2}\right)$ | (0,0,0,-1) | **0** |
| 111 | $\left(-\frac{1}{2},-\frac{1}{2},-\frac{1}{2},-\frac{1}{2}\right)$ | $\left(\frac{1}{2},\frac{1}{\sqrt{2}},0,\frac{1}{2}\right)$ | (1,0,0,0) | **1** |

### 3.2 4-Variable Function with 2 Queries

In this section we present our solution for the computational problem of comparing elements of a binary string.

**Problem:** *For a binary string of length 2k check if elements are equal by pairs:*

$$x_1=x_2,\ x_3=x_4,\ x_5=x_{6,\dots},\ x_{2k-1}=x_{2k}$$

We present an algorithm for string of length 4. We reduce the problem to computing the Boolean function of 4 variables. Boolean function can be represented by formula:

$$PAIR\_EQUALITY_4(x_1,x_2,x_3,x_4)=\neg(x_1\oplus x_2)\wedge\neg(x_3\oplus x_4).$$

**Deterministic complexity**: $D(PAIR\_EQUALITY_4)=4$, by sensitivity on accepting input.

**Algorithm 2**. Exact quantum query algorithm for $PAIR\_EQUALITY_4$ is presented in Figure 2.

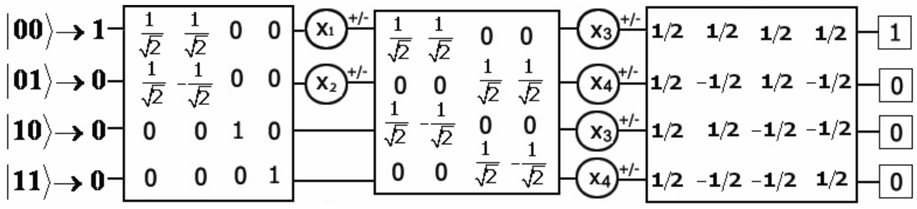


**Fig. 2.** Exact Quantum Query Algorithm for $PAIR\_EQUALITY_4$

Computational flow for each function input is presented in Table 2.

**Table 2.** Quantum Query Algorithm Computation Process for $PAIR\_EQUALITY_4$

| $X$ | after $\langle\vec{0}\|U_0Q_0$ | after $\langle\vec{0}\|U_0Q_0U_1Q_1$ | final state | result |
|---|---|---|---|---|
| 0000 | $\left(\frac{1}{\sqrt{2}},\frac{1}{\sqrt{2}},0,0\right)$ | $\left(\frac{1}{2},\frac{1}{2},\frac{1}{2},\frac{1}{2}\right)$ | (1,0,0,0) | **1** |
| 0001 | $\left(\frac{1}{\sqrt{2}},\frac{1}{\sqrt{2}},0,0\right)$ | $\left(\frac{1}{2},-\frac{1}{2},\frac{1}{2},-\frac{1}{2}\right)$ | (0,1,0,0) | **0** |
| 0010 | $\left(\frac{1}{\sqrt{2}},\frac{1}{\sqrt{2}},0,0\right)$ | $\left(-\frac{1}{2},\frac{1}{2},-\frac{1}{2},\frac{1}{2}\right)$ | (0,-1,0,0) | **0** |
| 0011 | $\left(\frac{1}{\sqrt{2}},\frac{1}{\sqrt{2}},0,0\right)$ | $\left(-\frac{1}{2},-\frac{1}{2},-\frac{1}{2},-\frac{1}{2}\right)$ | (-1,0,0,0) | **1** |

| | | | | |
|---|---|---|---|---|
| 0100 | $\left(\frac{1}{\sqrt{2}},-\frac{1}{\sqrt{2}},0,0\right)$ | $\left(\frac{1}{2},\frac{1}{2},-\frac{1}{2},-\frac{1}{2}\right)$ | (0,0,1,0) | **0** |
| 0101 | $\left(\frac{1}{\sqrt{2}},-\frac{1}{\sqrt{2}},0,0\right)$ | $\left(\frac{1}{2},-\frac{1}{2},-\frac{1}{2},\frac{1}{2}\right)$ | (0,0,0,1) | **0** |
| 0110 | $\left(\frac{1}{\sqrt{2}},-\frac{1}{\sqrt{2}},0,0\right)$ | $\left(-\frac{1}{2},\frac{1}{2},\frac{1}{2},-\frac{1}{2}\right)$ | (0,0,0,-1) | **0** |
| 0111 | $\left(\frac{1}{\sqrt{2}},-\frac{1}{\sqrt{2}},0,0\right)$ | $\left(-\frac{1}{2},-\frac{1}{2},\frac{1}{2},\frac{1}{2}\right)$ | (0,0,-1,0) | **0** |
| 1000 | $\left(-\frac{1}{\sqrt{2}},\frac{1}{\sqrt{2}},0,0\right)$ | $\left(-\frac{1}{2},-\frac{1}{2},\frac{1}{2},\frac{1}{2}\right)$ | (0,0,-1,0) | **0** |
| 1001 | $\left(-\frac{1}{\sqrt{2}},\frac{1}{\sqrt{2}},0,0\right)$ | $\left(-\frac{1}{2},\frac{1}{2},\frac{1}{2},-\frac{1}{2}\right)$ | (0,0,0,-1) | **0** |
| 1010 | $\left(-\frac{1}{\sqrt{2}},\frac{1}{\sqrt{2}},0,0\right)$ | $\left(\frac{1}{2},-\frac{1}{2},-\frac{1}{2},\frac{1}{2}\right)$ | (0,0,0,1) | **0** |
| 1011 | $\left(-\frac{1}{\sqrt{2}},\frac{1}{\sqrt{2}},0,0\right)$ | $\left(\frac{1}{2},\frac{1}{2},-\frac{1}{2},-\frac{1}{2}\right)$ | (0,0,1,0) | **0** |
| 1100 | $\left(-\frac{1}{\sqrt{2}},-\frac{1}{\sqrt{2}},0,0\right)$ | $\left(-\frac{1}{2},-\frac{1}{2},-\frac{1}{2},-\frac{1}{2}\right)$ | (-1,0,0,0) | **1** |
| 1101 | $\left(-\frac{1}{\sqrt{2}},-\frac{1}{\sqrt{2}},0,0\right)$ | $\left(-\frac{1}{2},\frac{1}{2},-\frac{1}{2},\frac{1}{2}\right)$ | (0,-1,0,0) | **0** |
| 1110 | $\left(-\frac{1}{\sqrt{2}},-\frac{1}{\sqrt{2}},0,0\right)$ | $\left(\frac{1}{2},-\frac{1}{2},\frac{1}{2},-\frac{1}{2}\right)$ | (0,1,0,0) | **0** |
| 1111 | $\left(-\frac{1}{\sqrt{2}},-\frac{1}{\sqrt{2}},0,0\right)$ | $\left(\frac{1}{2},\frac{1}{2},\frac{1}{2},\frac{1}{2}\right)$ | (1,0,0,0) | **1** |

## 4 Algorithm Transformation Methods

In this section we introduce quantum query algorithm transformation methods that can be useful for enlarging a set of exactly computable Boolean functions. Each method receives exact QQA on input, processes it as defined, and as a result slightly different exact algorithm is obtained that computes another function.

### 4.1 Output Value Assignment Inversion

The first method is the simplest one. All we need to do with original algorithm is to change assigned function value for each output to the opposite.

| ***First transformation method - Output value assignment inversion*** |
|---|
| **Input.** An arbitrary exact QQA that computes *f*(*X*). |
| **Transformation actions.** |
| • For each algorithm output change assigned value of function to opposite. If original assignment was $QM = (\alpha_1 \equiv k_1, ..., \alpha_m \equiv k_m)$, where $k_i \in \{0,1\}$, Then it is transformed to $QM' = (\alpha_1 \equiv \overline{k}_1, ..., \alpha_m \equiv \overline{k}_m)$, where $\overline{k}_i = 1 - k_i$. |
| **Output.** An exact QQA that computes $\overline{f}(X)$. |

**Box 1.** Description of the First Transformation Method

## 4.2 Output Value Assignment Permutation

Describing next method we will limit ourselves to using only exact QQA with specific properties as an input for transformation method.

**Property 1.** *We say that exact QQA satisfies Property 1 IFF on any input system state before a measurement is such that for exactly one amplitude* $\alpha_i$ *holds true that* $|\alpha_i|^2 = 1$. *For other amplitudes holds true that* $|\alpha_j|^2 = 0$, *for* $\forall j \neq i$.

Algorithm 1 and Algorithm 2 from section 3 satisfy Property 1.

| ***Second transformation method - Output value assignment permutation*** |
|---|
| **Input.** |
| • An exact QQA satisfying *Property 1* that computes *f*(*X*). |
| • Permutation $\sigma$ of the set $OutputValues = \{k_1, k_2, ..., k_m\}$. |
| **Transformation actions.** |
| • Permute function values assigned to outputs in order specified by $\sigma$. If original assignment was $QM = (\alpha_1 \equiv k_1, ..., \alpha_m \equiv k_m)$, where $k_i \in \{0,1\}$, Then it is transformed to $QM' = (\alpha_1 \equiv \sigma(k_1), ..., \alpha_m \equiv \sigma(k_m))$. |
| **Output.** An exact QQA for some function *g*(*X*). |

**Box 2.** Description of the Second Transformation Method

**Proof of correctness.** Application of the method does not break the exactness of QQA, because the essence of *Property 1* is that before the measurement we always obtain non-zero amplitude in exactly one output. Since function value is clearly specified for each output we would always observe specific value with probability 1 for any input.

□

The structure of new function *g*(*X*) strictly depends on internal properties of original algorithm. To explicitly define new function one needs to inspect original algorithm behavior on each input and construct a truth table for new output value assignment.

## 4.3 Query Variable Permutation

Let $\sigma$ be a permutation of the set $\{1,2,...,n\}$, where elements correspond to variable numbers. By saying that function *g*(*X*) is obtained by permutation of *f*(*X*) variables we mean the following: $g(X) = f\left(x_{\sigma(1)}, x_{\sigma(2)}, ..., x_{\sigma(n)}\right)$. In our third transformation method we expand the idea of variable permutation to QQA algorithm definition.

***Third transformation method – Query variable permutation***

**Input.**

- An arbitrary exact QQA that computes $f_n(X)$.
- Permutation $\sigma$ of variable numbers $VarNum = \{0,1,...,n\}$.

**Transformation actions.**

- Apply permutation of variable numbers $\sigma$ to all query transformations. If original *i*-th query was defined as $QQ_i = (\alpha_1 \equiv k_1, ..., \alpha_m \equiv k_m)$, Then it is transformed to $QQ_i' = (\alpha_1 \equiv \sigma(k_1), ..., \alpha_m \equiv \sigma(k_m))$, $k_i \in \{1,..,n\}$.

**Output.** An exact QQA computing a function $g(X) = f\left(x_{\sigma(1)}, x_{\sigma(2)}, ..., x_{\sigma(n)}\right)$.

**Box 3.** Description of the Third Transformation Method

**Proof of correctness.** If we apply transformation method described in Box 3, variable values will influence new algorithm flow according to the order specified by permutation $\sigma$, thus an algorithm computes *g*(X) instead of *f*(X).

□

## 4.4 Results of Applying Transformation Methods

Now we will demonstrate transformation methods application results for basic exact algorithms from section 3.

By using *EQUALITY*$_3$ function we obtained a set of 3-argument Boolean functions, we denote it with *QFunc3*, where for each function there is an exact QQA which computes it with 2 queries. In total 8 different functions were obtained $|QFunc3| = 8$. Functions are presented in Table 3.

**Table 3.** Results of Applying Transformation Methods for *EQUALITY*$_3$ Algorithm (set *QFunc3*)

| *X* | *EQUALITY* | Output value assignment pernutation | | | Output value assignment inversion | | | |
|---|---|---|---|---|---|---|---|---|
| | (1,0,0,0) | (0,1,0,0) | (0,0,1,0) | (0,0,0,1) | (0,1,1,1) | (1,0,1,1) | (1,1,0,1) | (1,1,1,0) |
| 000 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| 001 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| 010 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| 011 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| 100 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| 101 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| 110 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| 111 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| *D(f)* | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 2 |
| $Q_E(f)$ | **2** | **2** | **2** | **2** | **2** | **2** | **2** | **2** |

By using *PAIR_EQUALITY*$_4$ function we obtained a set of 4-argument Boolean functions, we denote it with *QFunc4*, where for each function there is an exact QQA which computes it with 2 queries. In total 24 different functions were obtained $|QFunc4| = 24$ and half of it is presented in table 4.

**Table 4.** Results of Applying Transformation Methods for *PAIR_EQUALITY*$_4$ Algorithm

| *X* | PAIR EQUALITY | 2nd method | | | 3rd method + 2nd method $\sigma_{VarNum} = \begin{pmatrix}1234\\1324\end{pmatrix}$ | | | | 3rd method + 2nd method $\sigma_{VarNum} = \begin{pmatrix}1234\\3124\end{pmatrix}$ | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $\begin{pmatrix}1\\0\\0\\0\end{pmatrix}$ | $\begin{pmatrix}0\\1\\0\\0\end{pmatrix}$ | $\begin{pmatrix}0\\0\\1\\0\end{pmatrix}$ | $\begin{pmatrix}0\\0\\0\\1\end{pmatrix}$ | $\begin{pmatrix}1\\0\\0\\0\end{pmatrix}$ | $\begin{pmatrix}0\\1\\0\\0\end{pmatrix}$ | $\begin{pmatrix}0\\0\\1\\0\end{pmatrix}$ | $\begin{pmatrix}0\\0\\0\\1\end{pmatrix}$ | $\begin{pmatrix}1\\0\\0\\0\end{pmatrix}$ | $\begin{pmatrix}0\\1\\0\\0\end{pmatrix}$ | $\begin{pmatrix}0\\0\\1\\0\end{pmatrix}$ | $\begin{pmatrix}0\\0\\0\\1\end{pmatrix}$ |
| 0000 | **1** | 0 | 0 | 0 | 0 | 0 | 0 | **1** | **1** | 0 | 0 | 0 |
| 0001 | 0 | **1** | 0 | 0 | 0 | **1** | 0 | 0 | 0 | **1** | 0 | 0 |
| 0010 | 0 | 0 | **1** | 0 | 0 | **1** | 0 | 0 | 0 | 0 | **1** | 0 |
| 0011 | 0 | 0 | 0 | **1** | 0 | 0 | 0 | **1** | 0 | 0 | 0 | **1** |
| 0100 | 0 | **1** | 0 | 0 | 0 | 0 | **1** | 0 | 0 | 0 | **1** | 0 |
| 0101 | **1** | 0 | 0 | 0 | **1** | 0 | 0 | 0 | 0 | 0 | 0 | **1** |
| 0110 | 0 | 0 | 0 | **1** | **1** | 0 | 0 | 0 | **1** | 0 | 0 | 0 |
| 0111 | 0 | 0 | **1** | 0 | 0 | 0 | **1** | 0 | 0 | **1** | 0 | 0 |
| 1000 | 0 | 0 | **1** | 0 | 0 | 0 | **1** | 0 | 0 | **1** | 0 | 0 |
| 1001 | 0 | 0 | 0 | **1** | **1** | 0 | 0 | 0 | **1** | 0 | 0 | 0 |
| 1010 | **1** | 0 | 0 | 0 | **1** | 0 | 0 | 0 | 0 | 0 | 0 | **1** |
| 1011 | 0 | **1** | 0 | 0 | 0 | 0 | **1** | 0 | 0 | 0 | **1** | 0 |
| 1100 | 0 | 0 | 0 | **1** | 0 | 0 | 0 | **1** | 0 | 0 | 0 | **1** |
| 1101 | 0 | 0 | **1** | 0 | 0 | **1** | 0 | 0 | 0 | 0 | **1** | 0 |
| 1110 | 0 | **1** | 0 | 0 | 0 | **1** | 0 | 0 | 0 | **1** | 0 | 0 |
| 1111 | **1** | 0 | 0 | 0 | 0 | 0 | 0 | **1** | **1** | 0 | 0 | 0 |
| ***D(f)*** | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| $Q_E(f)$ | **2** | **2** | **2** | **2** | **2** | **2** | **2** | **2** | **2** | **2** | **2** | **2** |

## 5 Algorithm Constructing Methods

In this section we will present several quantum query algorithm constructing methods. Each method requires explicitly specified exact QQAs on input, but as a result a bounded-error QQA for more complex function is constructed. Our methods maintain quantum query complexity for complex function in comparison to increased deterministic complexity, thus enlarging the gap between classical and quantum complexities of an algorithm. We offer a general constructions for computing *AND*, *OR* and *MAJORITY* kinds of Boolean functions.

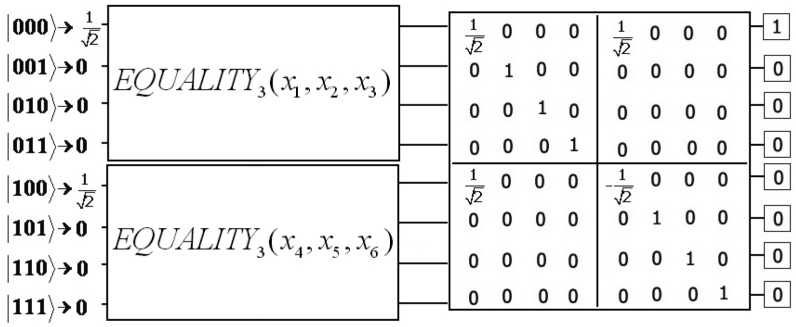
### 5.1 Bounded-error QQA for 6-Variable Function

We consider composite Boolean function, where two instances of $EQUALITY_3$ (section 3.1) are joined with logical AND operation:

$$EQUALITY_3^{\wedge 2}(x_1,...,x_6) = (\neg(x_1 \oplus x_2) \wedge \neg(x_2 \oplus x_3)) \wedge (\neg(x_4 \oplus x_5) \wedge \neg(x_5 \oplus x_6))$$

**Deterministic complexity.** $D(EQUALITY_3^{\wedge 2}) = 6$, by sensitivity on X=111111.

**Algorithm 3.** Our approach in designing an algorithm for $EQUALITY_3^{\wedge 2}$ is to employ quantum parallelism and superposition principle. We execute algorithm pattern defined by original algorithm for $EQUALITY_3$ in parallel for both blocks of $EQUALITY_3^{\wedge 2}$ variables. Finally, we apply additional quantum gate to correlate amplitude distribution. Algorithm flow is depicted explicitly in figure 3.



$|000\rangle \to \frac{1}{\sqrt{2}}$
$|001\rangle \to 0$
$|010\rangle \to 0$
$|011\rangle \to 0$
$|100\rangle \to \frac{1}{\sqrt{2}}$
$|101\rangle \to 0$
$|110\rangle \to 0$
$|111\rangle \to 0$
$EQUALITY_3(x_1,x_2,x_3)$
$EQUALITY_3(x_4,x_5,x_6)$
1
0
0
0
0
0
0
0

$$\begin{pmatrix} \frac{1}{\sqrt{2}} & 0 & 0 & 0 & \frac{1}{\sqrt{2}} & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ \frac{1}{\sqrt{2}} & 0 & 0 & 0 & -\frac{1}{\sqrt{2}} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

**Fig. 3.** Bounded-error QQA for $EQUALITY_3^{\wedge 2}$

**Quantum complexity.** Algorithm 3 computes $EQUALITY_3^{\wedge 2}$ using 2 queries with correct answer probability $p = 3/4$: $Q_{3/4}(EQUALITY_3^{\wedge 2}) = 2$.

**Proof.**
To calculate probabilities of obtaining correct function value it is enough to examine 4 cases depending on the value of each term of $EQUALITY_3^{\wedge 2}$. Results are presented

in a table below. We use wildcards "?" and "*" to denote that exactly one value under the same wildcard is $\pm\frac{1}{\sqrt{2}}$ (we don't care which one), but all others are zeroes.

**Table 5.** Calculation of Probabilities Depending on Algorithm Flow for $EQUALITY_3^{\wedge 2}$ .

| $EQUALITY_3$ $(x_1,x_2,x_3)$ | $EQUALITY_3$ $(x_4,x_5,x_6)$ | Amplitude distribution before last gate | Amplitude distribution after last gate | $p("1")$ |
|---|---|---|---|---|
| 0 | 0 | (0,?,?,?,0,*,*,*) | (0,?,?,?,0,*,*,*) | **0** |
| 0 | 1 | $\left(0,?,?,?,\frac{1}{\sqrt{2}},0,0,0\right)$ | $\left(\frac{1}{2},?,?,?,-\frac{1}{2},0,0,0\right)$ | **1/4** |
| 1 | 0 | $\left(\frac{1}{\sqrt{2}},0,0,0,0,?,?,?\right)$ | $\left(\frac{1}{2},0,0,0,\frac{1}{2},?,?,?\right)$ | **1/4** |
| 1 | 1 | $\left(\frac{1}{\sqrt{2}},0,0,0,\frac{1}{\sqrt{2}},0,0,0\right)$ | (1,0,0,0,0,0,0,0) | **1** |

So, we have $p("1")=1$ and $p("0")=3/4$, we did not use additional queries, thus estimation $Q_{3/4}(EQUALITY_3^{\wedge 2})=2$ is proved. □

## 5.2 First Constructing Method – *AND(f₁,f₂)*

In this section we will generalize approach used in previous section. To be able to use generalized version of the method we will limit ourselves to examining only exact QQA with specific properties.

**Property 2+** *We say that exact QQA satisfies Property2+ IFF there is exactly one accepting basic state and on any input for its amplitude $\alpha \in C$ only two values are possible before the final measurement: either $\alpha=0$ or $\alpha=1$.*

Algorithm 1 presented in section 3.1 satisfies Property 2+.

**Property 2-** *We say that exact QQA satisfies Property2- IFF there is exactly one accepting basic state and on any input for its amplitude $\alpha \in C$ only two values are possible before the final measurement: either $\alpha=0$ or $\alpha=-1$.*

**Lemma 1.** *It is possible to transform an algorithm that satisfies Property2- to an algorithm that satisfies Property2+ by applying additional unitary transformation.*

**Proof.** Let's assume that we have QQA satisfying *Property2-* and *k* is the number of accepting output. To transform algorithm to satisfy *Property2+* apply the following quantum gate: $U=(u_{ij})=\begin{cases} 0, & \text{if } i\neq j \\ 1, & \text{if } i=j\neq k \\ -1, & \text{if } i=j=k \end{cases}$ □

***First constructing method - AND($f_1$,$f_2$)***

**Input.**

- Two exact QQAs A1 and A2 satisfying *Property2+* that compute correspondingly Boolean functions $f_1(X_1)$ and $f_2(X_2)$.

**Transformation actions.**

1) If A1 and A2 utilize quantum systems of different size, extend the smallest one with auxiliary space to obtain an equal number of amplitudes. We denote the dimension of obtained Hilbert spaces with *m*.
2) For new algorithm utilize a quantum system with $2m$ amplitudes.
3) Combine unitary transformations and queries of A1 and A2 in the following way: $U_i = \begin{pmatrix} U_i^1 & O \\ O & U_i^2 \end{pmatrix}$, here $O$'s are $m \times m$ zero-matrices, $U_i^1$ and $U_i^2$ are either unitary transformations or query transformations of $A1$ and $A2$.
4) Start computation from the state $\langle\psi| = \left(1/\sqrt{2},\ 0,...,0,\ 1/\sqrt{2},\ 0,..,0\right)$.
5) Before the final measurement apply additional unitary gate. Let's denote the positions of accepting outputs of A1 and A2 by $acc_1$ and $acc_2$. Then the final gate is defined as follows:

$$U = \left(u_{ij}\right) = \begin{cases} 1, & \text{if } (i=j) \,\&\, (i \neq acc_1) \,\&\, (i \neq (m+acc_2)) \\ 1/\sqrt{2}, & \text{if } (i=j=acc_1) \\ 1/\sqrt{2}, & \text{if } (i=acc_1) \,\&\, (j=(m+acc_2)) \text{ OR } (i=(m+acc_2)) \,\&\, (j=acc_1) \\ -1/\sqrt{2}, & \text{if } (i=j=(m+acc_2)) \\ 0, & \text{otherwise} \end{cases}$$

6) Define as accepting output exactly one basic state $|acc_1\rangle$.

**Output.** A bounded-error QQA $A$ computing a function $F(X) = f_1(X_1) \wedge f_2(X_2)$ with probability $p = 3/4$ and complexity is $Q_{3/4}(A) = \max(Q_E(A_1), Q_E(A_2))$.

**Box 4.** Description of the First Constructing Method for *AND*($f_1$,$f_2$)

## 5.3 Bounded-error Quantum Algorithm for 8-Variable Function

Next step is to realize similar approach for *OR* operation. This time we take the second exact algorithm for *PAIR_EQUALITY*$_4$ as a base.

We consider composite Boolean function, where two instances of *PAIR_EQUALITY*$_4$ are joined with OR operation:

$$PAIR\_EQUALITY_4^{\vee 2}(x_1,...,x_8) = PAIR\_EQUALITY_4(x_1,x_2,x_3,x_4) \vee PAIR\_EQUALITY_4(x_5,x_6,x_7,x_8)$$

$$PAIR\_EQUALITY_4^{\vee 2}(x_1,...,x_8) = \left(\neg(x_1 \oplus x_2) \wedge \neg(x_3 \oplus x_4)\right) \vee \left(\neg(x_5 \oplus x_6) \wedge \neg(x_7 \oplus x_8)\right)$$

We succeeded in constructing quantum algorithm for $PAIR\_EQUALITY_4^{\vee 2}$, however algorithm structure is more complex than in *AND* operation case.

**Algorithm 4.** This time we use 4 qubit quantum system, so totally there are 16 amplitudes. First, we execute *PAIR_EQUALITY₄* algorithm pattern in parallel on first 8 amplitudes, and then apply two additional quantum gates $U_{SWAP}$ and $U_{OR}$:

$$U_{SWAP} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .. & 0 \\ 0 & 0 & 0 & 0 & [1] & 0 & 0 & 0 & 0 & 0 & .. & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .. & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & .. & 0 \\ 0 & [1] & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .. & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & [1] & 0 & .. & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & .. & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & .. & 0 \\ 0 & 0 & 0 & 0 & 0 & [1] & 0 & 0 & 0 & 0 & .. & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & .. & 0 \\ .. & .. & .. & .. & .. & .. & .. & .. & .. & .. & .. & .. \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .. & 1 \end{pmatrix}$$

$$U_{OR} = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 1/2 & 1/2 & 1/2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & -1/2 & 1/2 & -1/2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 1/2 & -1/2 & -1/2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & -1/2 & -1/2 & 1/2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1/2 & 1/2 & 1/2 & 1/2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1/2 & -1/2 & 1/2 & -1/2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1/2 & 1/2 & -1/2 & -1/2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1/2 & -1/2 & -1/2 & 1/2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Quantum measurement:

$$QM = \left([1,1],[1,0,0,0],[1,0,0,0],0,0,0,0,0,0\right)$$

Complete algorithm structure is presented in Figure 4.

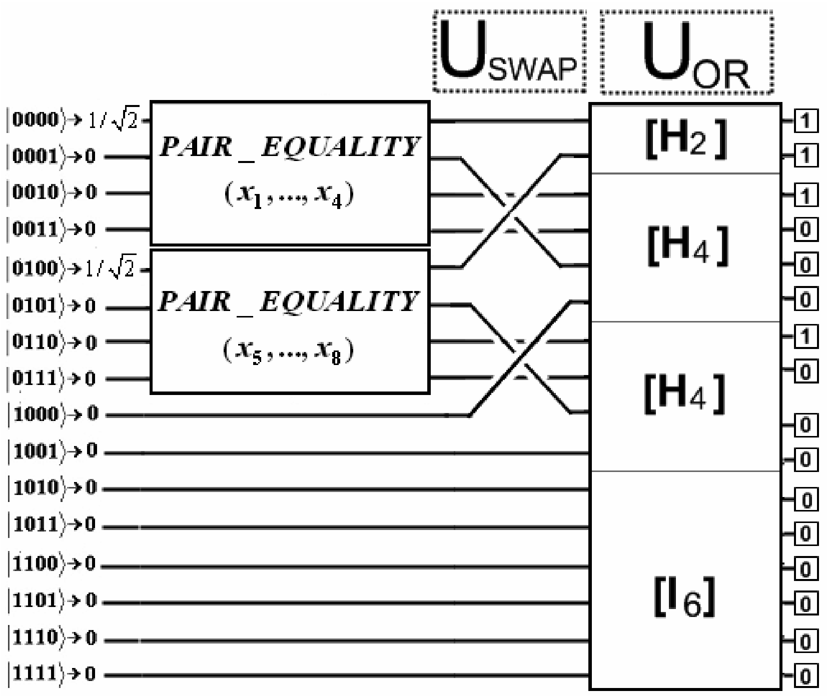


**Fig. 4.** Bounded-error QQA for $PAIR\_EQUALITY_4^{\vee 2}$

**Quantum complexity.** Algorithm 4 computes $PAIR\_EQUALITY_4^{\vee 2}$ using 2 queries with correct answer probability $p = 5/8$ : $Q_{5/8}(PAIR\_EQUALITY_4^{\vee 2}) = 2$ .

**Proof.** We demonstrate computation process results, what cover all possible inputs.

**I** $PAIR\_EQUALITY_4(x_1, x_2, x_3, x_4) = 1$ and $PAIR\_EQUALITY_4(x_5, x_6, x_7, x_8) = 1$

| Amplitude distribution before $U_{OR}$ | Amplitude distribution before the measurement | $p("1")$ |
|---|---|---|
| $\left(\left[\pm\frac{1}{\sqrt{2}}, \pm\frac{1}{\sqrt{2}}\right], [0,0,0,0], [0,0,0,0], 0,0,0,0,0,0\right)$ | $\pm(1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)$ or $\pm(0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0)$ | 1 |

**II** $PAIR\_EQUALITY_4(x_1,x_2,x_3,x_4)=1$ and $PAIR\_EQUALITY_4(x_5,x_6,x_7,x_8)=0$

| Amplitude distribution before $U_{OR}$ | Amplitude distribution before the measurement | $p("1")$ |
|---|---|---|
| $\left(\left[\pm\frac{1}{\sqrt{2}},0\right],[0,0,0,0],[?,?,?,0], 0,0,0,0,0,0\right)$ | $\left(\left[\pm\frac{1}{2},\pm\frac{1}{2}\right],[0,0,0,0], \left[\pm\frac{1}{2\sqrt{2}},\pm\frac{1}{2\sqrt{2}},\pm\frac{1}{2\sqrt{2}},\pm\frac{1}{2\sqrt{2}}\right],0,0,0,0,0,0\right)$ | $\frac{1}{4}+\frac{1}{4}+\frac{1}{8}= =\frac{5}{8}$ |

**III** $PAIR\_EQUALITY_4(x_1,x_2,x_3,x_4)=0$ and $PAIR\_EQUALITY_4(x_5,x_6,x_7,x_8)=1$

| Amplitude distribution before $U_{OR}$ | Amplitude distribution before the measurement | $p("1")$ |
|---|---|---|
| $\left(\left[0,\pm\frac{1}{\sqrt{2}}\right],[?,?,?,0],[0,0,0,0], 0,0,0,0,0,0\right)$ | $\left(\left[\pm\frac{1}{2},\pm\frac{1}{2}\right],\left[\pm\frac{1}{2\sqrt{2}},\pm\frac{1}{2\sqrt{2}},\pm\frac{1}{2\sqrt{2}},\pm\frac{1}{2\sqrt{2}}\right], [0,0,0,0],0,0,0,0,0,0\right)$ | $\frac{1}{4}+\frac{1}{4}+\frac{1}{8}= =\frac{5}{8}$ |

**IV** $PAIR\_EQUALITY_4(x_1,x_2,x_3,x_4)=0$ and $PAIR\_EQUALITY_4(x_5,x_6,x_7,x_8)=0$

| Amplitude distribution before $U_{OR}$ | Amplitude distribution before the measurement | $p("1")$ |
|---|---|---|
| $\left([0,0],[?,?,?,0],[*,*,*,0], 0,0,0,0,0,0\right)$ | $\left([0,0],\left[\pm\frac{1}{2\sqrt{2}},\pm\frac{1}{2\sqrt{2}},\pm\frac{1}{2\sqrt{2}},\pm\frac{1}{2\sqrt{2}}\right], \left[\pm\frac{1}{2\sqrt{2}},\pm\frac{1}{2\sqrt{2}},\pm\frac{1}{2\sqrt{2}},\pm\frac{1}{2\sqrt{2}}\right],0,0,0,0,0,0\right)$ | $\frac{1}{8}+\frac{1}{8}=\frac{1}{4}$ |

Correct function result is always obtained with probability not less than 5/8, thus complexity estimation is proved.

□

## 5.4 Second Constructing Method – *OR*($f_1$,$f_2$)

In this section we generalize approach for computing composite Boolean functions matching *OR*($f_1$,$f_2$) pattern.

First, we define next QQA property.

**Property 3** *We say that exact QQA satisfies Property3 IFF*

- *it satisfies Property1;*
- *there is exactly one accepting basic state;*
- *on any input accepting state amplitude value before measurement is* $\alpha\in\{-1,0,1\}$

Algorithm 1 and Algorithm 2 from section 3 both satisfy Property3.

The following lemma will be useful during method application.

**Lemma 2.** *For any QQA on any computation step it is possible to swap amplitude values in arbitrary order by applying specific quantum gate.*

**Proof.** Assume we need to swap amplitude values according to permutation $\sigma = \begin{pmatrix} \alpha_1 & \alpha_2 & \dots & \alpha_n \\ \beta_1 & \beta_2 & \dots & \beta_n \end{pmatrix}$. Then we can define quantum gate $U_{SWAP} = \{u_{ij}\}$ elements as:

- $\forall k \in \{1 \dots n\}: \ u_{\alpha_k \beta_k} = 1$;
- $u_{ij} = 0$, in all other cases. □

Now we are ready to formulate a method for computing *OR(f₁,f₂)* kind of functions. For simplicity we consider only such input algorithms, which employ 2 qubit system. However, approach can be generalized for quantum systems of arbitrary size.

***Second constructing method – OR($f_1$,$f_2$)***

**Input.**

- Two exact QQAs A1 and A2 satisfying *Property3*, which use quantum systems with 2 qubits and compute correspondingly Boolean functions $f_1(X_1)$ and $f_2(X_2)$.

**Transformation actions.**

1) Use 4 qubit quantum system for new algorithm, totally $2^4 = 16$ basic states.
2) Convert initial state $\langle \vec{0} | = (1,0,0,0,...,0)$ into state:

$$\langle \psi | = \left( \left[ \frac{1}{\sqrt{2}}, 0,0,0 \right], \left[ \frac{1}{\sqrt{2}}, 0,0,0 \right], 0,0,0,0,0,0,0,0 \right)$$

3) Combine A1 and A2 unitary and query transformations in the following way:

$$U_i = \begin{pmatrix} \left[U_i^1\right] & O_{4x4} & O_{4x8} \\ O_{4x4} & \left[U_i^2\right] & O_{4x8} \\ O_{8x4} & O_{8x4} & \left[I_8\right] \end{pmatrix}$$, where [I₈] is 8x8 identity matrix.

4) Apply amplitude swapping gate $U_{\text{SWAP}}$, which was defined in the proof of lemma 2, to arrange amplitudes in the following order:

- 1st amplitude ↔ first sub-algorithm accepting amplitude;
- 2nd amplitude ↔ second sub-algorithm accepting amplitude;
- 3rd, 4th, 5th amplitudes ↔ first sub-algorithm rejecting amplitudes;
- 7th, 8th, 9th amplitudes ↔ second sub-algorithm rejecting amplitudes.

5) Apply the last quantum gate, which was precisely defined in previous section:

$$U_{OR} = \begin{pmatrix} [H_2] & O_{2x4} & O_{2x4} & O_{2x6} \\ O_{4x2} & [H_4] & O_{4x4} & O_{4x6} \\ O_{4x2} & O_{4x4} & [H_4] & O_{4x6} \\ O_{6x2} & O_{6x4} & O_{6x4} & [I_6] \end{pmatrix}$$

6) Assign function values to algorithm outputs s follows:

$$QM = ([1,1],[1,0,0,0],[1,0,0,0],0,0,0,0,0,0)$$

**Output.** A bounded-error QQA $A$ computing a function $F(X) = f_1(X_1) \vee f_2(X_2)$ with probability $p = 5/8$ and complexity is $Q_{5/8}(A) = \max(Q_E(A_1), Q_E(A_2))$.

**Box 5.** Description of the Second Constructing Method for $OR(f_1,f_2)$

## 5.5 Bounded-error Quantum Algorithm for 12-Variable Function

Let us try to increase the effect gained by employing quantum parallelism. Next idea is to execute 4 instances of algorithm in parallel, adjusting algorithm parameters in appropriate way. We will take as a pattern function *EQUALITY₃* from section 3.1.

Designed algorithm and additional gates are presented in Figure 5 and below. Algorithm computes some 12-variable Boolean function with bounded-error.

**Algorithm 5**

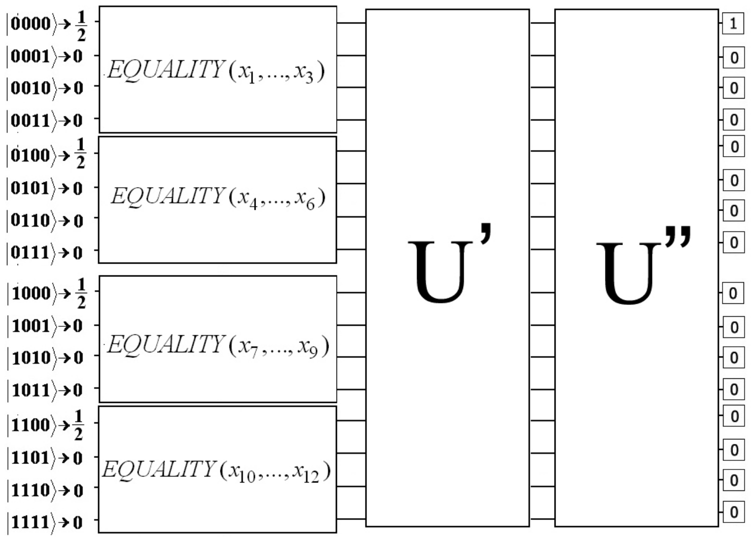


**Fig. 5.** Bounded-error Quantum Query Algorithm for 12-Variable Function

Additional quantum gates (empty matrix cells correspond to “0”):

$$U'=\left(\begin{array}{cccccccccccccccc}
\frac{1}{\sqrt{2}}&0&0&0&\frac{1}{\sqrt{2}}&0&0&0&0&0&0&0&0&0&0&0\\
0&1&&&0&&&&0&&&&&&&0\\
0&&1&&0&&&&0&&&&&&&0\\
0&&&1&0&&&&0&&&&&&&0\\
\frac{1}{\sqrt{2}}&0&0&0&\frac{-1}{\sqrt{2}}&&&&0&&&&&&&0\\
0&&&&&1&&&0&&&&&&&0\\
0&&&&&&1&&0&&&&&&&0\\
0&&&&&&&1&0&&&&&&&0\\
0&0&0&0&0&0&0&0&\frac{1}{\sqrt{2}}&0&0&0&\frac{1}{\sqrt{2}}&0&0&0\\
0&&&&&&&&0&1&&&0&&&0\\
0&&&&&&&&0&&1&&0&&&0\\
0&&&&&&&&0&&&1&0&&&0\\
0&&&&&&&&\frac{1}{\sqrt{2}}&0&0&0&\frac{-1}{\sqrt{2}}&&&0\\
0&&&&&&&&0&&&&&1&&0\\
0&&&&&&&&0&&&&&&1&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1
\end{array}\right)
\quad
U''=\left(\begin{array}{cccccccccccccccc}
\frac{1}{\sqrt{2}}&0&0&0&0&0&0&0&\frac{1}{\sqrt{2}}&0&0&0&0&0&0&0\\
0&1&&&&&&&0&&&&&&&0\\
0&&1&&&&&&0&&&&&&&0\\
0&&&1&&&&&0&&&&&&&0\\
0&&&&1&&&&0&&&&&&&0\\
0&&&&&1&&&0&&&&&&&0\\
0&&&&&&1&&0&&&&&&&0\\
0&&&&&&&1&0&&&&&&&0\\
\frac{1}{\sqrt{2}}&0&0&0&0&0&0&0&\frac{-1}{\sqrt{2}}&0&0&0&0&0&0&0\\
0&&&&&&&&0&1&&&&&&0\\
0&&&&&&&&0&&1&&&&&0\\
0&&&&&&&&0&&&1&&&&0\\
0&&&&&&&&0&&&&1&&&0\\
0&&&&&&&&0&&&&&1&&0\\
0&&&&&&&&0&&&&&&1&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1
\end{array}\right)$$

After examination of algorithm computational flow and calculation of probabilities we obtained the result that is formulated in the next statement.

**Quantum complexity.** Algorithm 5 computes function defined as:

$$F(x_1,...,x_{12})=1 \quad \Leftrightarrow \quad \left(\begin{array}{l}\text{Not less than 3 functions from: } EQUALITY(x_1,..,x_3),\\ EQUALITY(x_4,...,x_6), EQUALITY(x_7,...,x_9),\\ EQUALITY(x_{10},...,x_{12}) \text{ give value "1".}\end{array}\right)$$

and complexity is $Q_{9/16}(\text{Algorithm5})=2$.

**Deterministic complexity.** This time we did not achieve maximal possible gap. From the definition of function $F$ we find that sensitivity is $s(F)=9$, thus in this case we can only register a gap $D(f)\geq 9$ vs. $Q_{9/16}(f)=2$.

## 5.6 Third Constructing Method - *MAJORITY*

We examined the structure of algorithm in the previous section 5.5 and concluded that such approach would be useful for computing Boolean functions that belong to *MAJORITY* class.

**Definition 1.** *Boolean function MAJORITY$_n$(X), with $n=2k+1,\ k\in N$ arguments is defined as:*

$$MAJORITY_{2k+1}(X)=1 \quad \Leftrightarrow \quad \sum_{i=1}^{2k+1} x_i > k$$

When number of arguments is odd, then there always is a clear majority of “0” or “1” in input vector. When number of function arguments is even, the case when number of "0" and "1" is equal is not defined. We define another one class of Boolean functions for the case when number of function arguments is even.

**Definition 2.** *Boolean function MAJORITY_EVEN$_n$(X), with* $n=2k,\ k\in N,\ k>0$ *arguments is defined as:*

$$MAJORITY\_EVEN_{2k}(X)=1 \quad\Leftrightarrow\quad \sum_{i=1}^{2k} x_i > k$$

So, when number of "0" and "1" in input vector is equal, then function value is "0".

In addition to *MAJORITY* function we define also *MAJORITY* composite construction. The difference is that in *MAJORITY* construction we use other Boolean functions as *MAJORITY* arguments.

**Definition 3.** *We define MAJORITY$_n$ construction (*$n=2k+1,\ k\in N$*) as a Boolean function where arguments are arbitrary Boolean functions $f_i$ and which is defined as:*

$$\left(MAJORITY_{2k+1}[f_1,f_2,...,f_{2k+1}](X)=1 \quad\Leftrightarrow\quad \sum_{i=1}^{2k+1} f_i(x_i)>k\right),$$
$$\text{where } X=x_1x_2...x_{2k+1}$$

Construction *MAJORITY_EVEN$_n$* is defined in a similar way.

Let's again consider quantum algorithm 5 from the section 5.5. Definition of Boolean function was:

$$F(x_1,...,x_{12})=1 \quad\Leftrightarrow\quad \left(\begin{array}{l}\text{Not less than 3 functions from: } EQUALITY(x_1,..,x_3),\\ EQUALITY(x_4,...,x_6), EQUALITY(x_7,...,x_9),\\ EQUALITY(x_{10},...,x_{12}) \text{ give value "1".}\end{array}\right)$$

Now we can rewrite it as:

$$F(x_1,...,x_{12})=MAJORITY\_EVEN_4[EQUALITY_3](x_1,...,x_{12})=MAJORITY\_EVEN_4($$
$$EQUALITY_3(x_1,...,x_3),\ EQUALITY_3(x_4,...,x_6), EQUALITY_3(x_7,...,x_9),\ EQUALITY_3(x_{10},...,x_{12}))$$

Next, we formulate a general algorithm constructing method for computing *MAJORITY_EVEN$_4$* construction.

| ***Third constructing method - MAJORITY*** |
|---|
| **Input.**<br>• Four exact QQAs A1, A2, A3, A4 satisfying *Property2+* that compute correspondingly Boolean functions $f_1(X_1), f_2(X_2), f_3(X_3), f_4(X_4)$.<br>**Transformation actions.**<br>1) If any of input algorithms satisfy Property2-, then transform it to algorithm which satisfies Property2+ by applying lemma 1.<br>2) Combine unitary and query transformations of input algorithms in the |

following way: $U_i = \begin{pmatrix} U_i^1 & O & O & O \\ O & U_i^2 & O & O \\ O & O & U_i^3 & O \\ O & O & O & U_i^4 \end{pmatrix}$, where $U_i^k$ is *k*-th algorithm transformation. *O*'s are zero sub-matrices, size depends on number of input algorithm amplitudes.

3) Start computation in a state:

$$\langle\psi| = \left(\frac{1}{2},\ 0,...,0,\ \frac{1}{2},\ 0,...,0,\ \frac{1}{2},\ 0,...,0,\ \frac{1}{2},\ 0,...,0\right)$$

where positions of 1/2 correspond to positions of the first amplitude of input algorithms.

4) Before the measurement apply two additional quantum transformations. We denote input algorithm accepting amplitude numbers as $\alpha_1,\ \alpha_2,\ \alpha_3$ and $\alpha_4$.

$$U' = \{u_{ij}\} = \begin{cases} 1, & \text{if } (i = j \neq \alpha_1) \vee (i = j \neq \alpha_2) \vee (i = j \neq \alpha_3) \vee (i = j \neq \alpha_4) \\ 1/\sqrt{2}, & \text{if } (i = j = \alpha_1) \vee (i = j = \alpha_3) \\ -1/\sqrt{2}, & \text{if } (i = j = \alpha_2) \vee (i = j = \alpha_4) \\ 1/\sqrt{2}, & \text{if } (i = \alpha_1\ \&\ j = \alpha_2) \vee (i = \alpha_2\ \&\ j = \alpha_1) \\ 1/\sqrt{2}, & \text{if } (i = \alpha_3\ \&\ j = \alpha_4) \vee (i = \alpha_4\ \&\ j = \alpha_3) \\ 0, & \text{otherwise} \end{cases}$$

$$U'' = \{u_{ij}\} = \begin{cases} 1, & \text{if } (i = j \neq \alpha_1) \vee (i = j \neq \alpha_2) \vee (i = j \neq \alpha_3) \vee (i = j \neq \alpha_4) \\ 1/\sqrt{2}, & \text{if } (i = j = \alpha_1) \\ -1/\sqrt{2}, & \text{if } (i = j = \alpha_3) \\ 1/\sqrt{2}, & \text{if } (i = \alpha_1\ \&\ j = \alpha_3) \vee (i = \alpha_3\ \&\ j = \alpha_1) \\ 0, & \text{otherwise} \end{cases}$$

5) Define as accepting state exactly one basic state $|\alpha_1\rangle$, that correspond to algorithm A1 accepting state.

**Output.** A bounded-error QQA *A* computing a function $MAJORITY\_EVEN_4[f_1, f_2, f_3, f_4](X)$, where $X = X_1X_2X_3X_4$ with probability $p = 9/16$ and complexity is $Q_{9/16}(A) = \max(Q_E(A_1), Q_E(A_2), Q_E(A_3), Q_E(A_4))$.

**Box 6.** Description of the Third Constructing Method for *MAJORITY_EVEN*$_4$

By using a constant function $f(x)=1$ as one of constructing method input algorithms it is possible to achieve that resulting algorithm computes:

$$MAJORITY\_EVEN_4(f_1,f_2,f_3,1)=MAJORITY_3(f_1,f_2,f_3)$$

## 6 Results of Applying Methods

We applied transformation and designing methods to two basic exact QQAs described in section 3. In total we obtained 32 exact QQAs and 512 QQAs with bounded error. Each algorithm computes different Boolean function and uses only 2 queries. Results are summarized in table 6. Here *n* is number of variables of computable function.

**Table 6.** Results of Transformation and Constructing Methods Application

| **Basic exact quantum algorithms** | | | | |
|---|---|---|---|---|
| **Set** | **Size** | **Number of arguments** | **Number of questions** | **Probability** |
| *QFunc3* | **8** | 3 | 2 | 1 |
| *QFunc4* | **24** | 4 | 2 | 1 |
| **Constructed algorithms sets** | | | | |
| **Set** | **Size** | **Number of arguments** | **Number of questions** | **Probability** |
| *QFunc_AND* | **16** | 6 | 2 | 3/4 |
| *QFunc_OR* | **256** | 6,7,8 | 2 | 5/8 |
| *QFunc_MAJ_EVEN*$_4$ | **256** | 12 | 2 | 9/16 |
| *QFunc_MAJORITY*$_3$ | **64** | 9 | 2 | 9/16 |
| **Total** | **832** | | | |

The important point is that invention of each brand-new exact QQA with required properties will at once significantly increase a set of efficiently computable functions.

## 7 Conclusion

In this work we consider quantum query algorithm constructing problems. We have tried to develop some general approaches for designing algorithms for computing Boolean functions defined by logical formula. The main goal of research is to develop a framework for building ad-hoc quantum algorithms for arbitrary Boolean functions. In this paper we describe general constructions for designing quantum algorithms for *AND*, *OR* and *MAJORITY* kinds of Boolean functions.

First, we presented two exact quantum query algorithms for 3 and 4 argument functions. Both algorithms save questions comparing to the best possible classical algorithm. Algorithms are used in further sections as a base for algorithm transformation and constructing methods.

Next, we proposed techniques that allow transformation of an existing quantum query algorithm for a certain Boolean function so that the resulting algorithm computes a function with other logical structure. We illustrated methods by applying them to two basic exact algorithms.

Finally, we suggested approaches that allow building bounded-error quantum query algorithms for complex functions based on already known exact algorithms. Constructing methods include efficient solutions for *AND*, *OR* and *MAJORITY* constructions.

Combination of these three aspects allowed us to construct large sets of efficient quantum algorithms for various Boolean functions.

Further work in this direction could be to invent new efficient quantum algorithms that exceed already known separation from classical algorithms. Another important direction is improvement of general algorithm designing techniques.

## 8 Acknowledgments

I would like to thank my supervisor Rusins Freivalds for introducing me with quantum computation and for permanent support.

This research is supported by the European Social Fund.

## References

[1] H. Buhrman and R. de Wolf: “Complexity Measures and Decision Tree Complexity: A Survey”. Theoretical Computer Science, v. 288(1): 21–43 (2002).

[2] R. de Wolf: “Quantum Computing and Communication Complexity”. University of Amsterdam (2001).

[3] R. Cleve, A. Ekert, C. Macchiavello, et al. “Quantum Algorithms Revisited”. Proceedings of the Royal Society, London, A454 (1998).

[4] J. Gruska: “Quantum Computing”. McGraw-Hill (1999).

[5] M. Nielsen, I. Chuang: “Quantum Computation and Quantum Information”. Cambridge University Press (2000).

[6] Ambainis: “Quantum query algorithms and lower bounds (survey article)”. Proceedings of FOTFS III, to appear.

[7] Ambainis and R. de Wolf: “Average-case quantum query complexity”. Journal of Physics A 34, pp 6741–6754 (2001).

[8] Ambainis. “Polynomial degree vs. quantum query complexity”. Journal of Computer and System Sciences 72, pp. 220–238 (2006).

# QUALITY MODELS

# Conceptualising Informatization with the Onto6 Methodology

Uldis Straujums

University of Latvia, 19 Raina Blvd., Riga, LV-1586, Latvia
uldis.straujums@lu.lv

**Abstract** This paper presents a way of conceptualising informatization through a new methodology that is called Onto6. Informatization is defined as the maintained process of creating the technical, economic, and social conditions which are necessary for the fulfilment of information needs. The Onto6 methodology identifies objects and determines their interaction and functionality. The methodology is based on meta-ontology, and it involves the creation of an instance in accordance with the domain. The initial ontology is created from the ontology instance, and it is extended during the informatization. In the initial ontology, root metaphors and the relationship among same must be defined by the planner of informatization. More work is being done by the implementers of informatization to refine the initial ontology. This creates an ontology cluster which consists of meta-ontology, a meta-ontology instance, the initial ontology, and the refinements of that ontology. This reflects a conceptualisation of informatization in a particular domain. The author has taken part in the conceptualisation and maintenance of informatization in several domains. The Onto6 methodology that is proposed in this paper has been applied to several domains at the national level.

## 1 Defining Informatization

Use of the term "informatization" on the Internet has increased continuously over the course of time. The Google search engine found 40,400 links to the word in November 2004, 258,000 in January 2006, and 1.24 million in September 2007. The Alta Vista engine found only 1,868 references in April 1998. Despite this fact, however, it is difficult to come up with a precise definition of the term. Different sources provide slightly different definitions. The South Korean National Computing Agency [Lim01] defines "informatization" as "converting the main goods and energy of a social economy to information through the revolution of high data communication technology and utilising information produced by gathering, processing and distributing data within the vast fields of the society." A report prepared for the United Nations Economic Commission for Europe [Haf03], the definition says that "informatization" is the process whereby information technologies transforme economies and societies.
In this paper, we shall use the definition produced by Bicevskis, Andzans, Ikaunieks, Medvedis and Straujums in *Education Media International* [BAIMS04]:

"Informatization is the maintained process of creating the technical, economic and social conditions for the fulfilment of information needs."
All of these definitions make it clear that while informatization covers the area of computerisation, it is, in fact, a broader term than just that. This author has participated in the conceptualisation of informatization and the maintenance of informatization for several domains, ranging from domains at the national level to the domain of an educational institution. These are the case studies which are examined in this paper:

- The Latvian Education Informatization System (LIIS) project, which deals with the whole range of information issues – educational content, management, information services, infrastructure, as well as user training at several levels – schools, school boards, and the Ministry of Education and Science. The LIIS project is an essential component of the Latvian National Informatics Programme;
- The Unified State Library Information System project (VVBIS), which was developed on the basis of the requirement of the National Informatics Programme that Universal Information Services be created;
- The informatics curriculum standard for Latvia's general education schools. The analysis of the needs of the people of Latvia in terms of an ICT-educated workforce has been conducted, and solutions have been proposed as to the content of informatics courses. Detailed plans on how to provide the necessary training for existing and future informatics teachers have been drawn up.

All of the projects which are examined in this paper have actually been implemented over the course of many years and with considerable success.

## 2 An Ontological Approach to the Conceptualization of Informatization

Conceptualisation of informatization is a process with several phases – analysis of the existing situation, setting of goals, planning of activities, and evaluation of costs. This tends to be an iterative process, and several alternatives are compared. The consequence is that a framework must be established so as to correlate the knowledge that has been acquired during the developmental process. The requirements for such a framework include:

- The ability to organise the essential quality in a hierarchical form;
- The mapping of the qualities in the context of recognised standards;
- The interaction of organisational, domain-specific and technological aspects of the process.

All of these requirements led to the introduction of several different approaches in accumulating and then reusing knowledge – controlled vocabularies, thesauri, classification schemes, taxonomies, topic maps, frame languages, logical theories, and meta-models. There are many other approaches called ontologies. In formal terms, an ontology is the explicit specification of a conceptualisation for a domain [Gru95]. This specification can take the form of a logical theory, accounting for the intended meaning [Gua98], or it can strive to make use of the notion of linguistic relativism [Wys04]. According to Crubézy and Musen [CM04, p. 321], “ontologies support the creation of repositories of domain-specific reference knowledge – domain knowledge bases – for communication and sharing of this knowledge among people and computer applications.” The differences lie in the ability to describe terms and to define relations among them. The differences at the level of formality create an ontology spectrum [Wil04].

The controlled vocabulary, i.e., the list of enumerated terms, is at one end of the spectrum. Ideally, each term should have just one meaning. In practice, however, terms are qualified in accordance with different meanings in different domains. If several terms have the same meaning, one is preferred, while the others are classified as synonyms or aliases. The controlled vocabulary is used to built up more advanced ontologies. For example, a thesaurus is built up by adding associative relationships to vocabulary. Frame languages have the ability to express the properties, logical constraints, and detailed relationships of terms. A meta-model is an explicit model within a domain of interest, containing terms and rules that are needed to build specific models. A meta-model is an ontology, but it is a richer notion – it can be used as a set of building blocks and rules which apply to the construction of models, as a model of a domain of interest, or as an instance of another model.

## 3 Onto6 – a Methodology for the Conceptualization of Informatization

This author has developed a methodology that can be used in the early conceptualisation of informatization in different domains. The target audience for this methodology is made up of the users of the domain which is undergoing informatization. These users are usually unaware of formal means for describing systems – UML, OWL, GRAPES, OMT, etc.

If we take into account the skills and the knowledge of these clients, we understand that the methodology which is proposed to them must be simple, understandable, and extendable. The Onto6 methodology was inspired by several sources, particularly the GRAPES-86 modelling language, the Object Modelling Technique (OMT), and the 6W approach.

GRAPES (Graphical Engineering System) [GRA02] is a method for system development which supports the entire software development process, from problem analysis to implementation. The GRAPES-86 modelling language is the central element, making it possible to specify the structure, behaviour, and data of information processing systems, particularly distributed systems such as company organisations and network architectures. GRAPES-86 has a formal, defined text and graphic syntax, which means that it can be statically assessed. Furthermore, GRAPES-86 includes a dynamic processing model. The dynamic behaviour of GRAPES models can, therefore, be simulated and analysed. A modelling and development environment GRADE implementing GRAPES is developed [PKB93].

The level of formalism is too high, however, and the learning curve is too steep for typical informatization clients to make effective use of GRAPES-86. More concise means for the conceptualisation of informatization had to be defined.
The attractive approach developed by Rumbaugh [Rum97] is called Object Modelling Technology, or OMT. It is simpler than GRAPES, using object-oriented modelling to think clearly about problems and to draw three kinds of diagrams. The need to study the precise notion of diagrams and the preoccupying attention to software development, however, make this approach difficult for the typical informatization user to understand.
The 6W approach is extensively used in various areas – knowledge management [KMO07], context analysis [Mot00], architectural design [Lan04], etc. The approach involves six questions about the topic – what, where, when, how, why, and who. The approach was created by Rudyard Kipling back in 1902 [Kipl02]. This may seem to oversimplify the complexity of the topic that is being considered, but in reality there is a vast amount of sub-questions which arise from the points of view and expectations of those who are doing the asking.
The author proposes a methodology based on the 6W approach as a top-level structure which can be understood by every client, one which supplies the most typical sub-questions (terms) for further investigation of the relevant topic.

## 3.1 Meta-ontology

The developed Onto6 methodology identifies the object and determines interaction and functionality. The methodology is based on a meta-ontology which creates an instance in accordance with the domain. This instance remains stable and unchanged during the informatization process. The initial ontology is created from the ontology of the instance, and then it is extended during the process of informatization.
The proposed meta-ontology is shown in Figure 1.

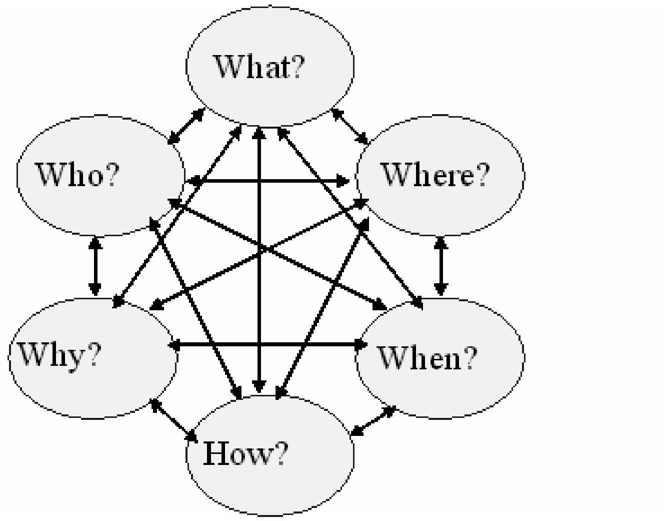


**Fig 1.** The Meta-ontology of the Onto6 Metodology

The meta-ontology contains the most abstract terms and the relations among those terms, presenting them in a simple form. The level of simplicity is based on the level of competence in the sense of the formal notations of the system's users, i.e., the clients of the process of informatization.

The top-level terms that have been chosen by the author are inspired by the 6W approach. Each term has its attributes and sub-terms. There can be vast numbers of sub-terms which characterise each top-level term. Lower-level terms themselves can have sub-terms. This means the emergence of a hierarchy, possible one that is recursive.

The author has proposed a set of terms for top-level term attributes. These reflect the most abstract aspects of each top-level term and should be mapped in relation to the real entities when creating the initial ontology. Table 1 shows the attributes of the top-level terms. The attributes are introduced into the Onto6 meta-ontology with a selective subset of attributes from a number of component ontologies [Goed99, Lep05, Hoss06, Sowa06]. The root metaphor and related concepts described in [Bar94, Mot00, VB01, Fon02, Gaz02, Pul03, Lan04, VZSS04 and Sowa07] are also taken into account. The goal is to create a relatively small set of attributes that can be understood by unsophisticated users and that is sufficient for the conceptualisation of informatization.

**Table 1**. Top-level Terms in the Onto6 Meta-ontology and their Attributes

| **Top level term** | **Attribute** |
|---|---|
| What | Concept<br>Sign<br>Referent<br>Reality<br>Resource<br>Plan<br>Tool<br>Knowledge<br>Information<br>System<br>Model<br>Paradigm |
| Where | Subjective reality<br>Physical reality<br>Location |
| When | Time<br>State<br>Transition<br>Event<br>Life cycle |
| How | Abstraction<br>Concretizing<br>Rule |

| | Role |
|---|---|
| Why | Point of view<br>Perspective<br>Universe of discourse<br>Context<br>Goal<br>Goal-producing context<br>Reason<br>Problem<br>Strength<br>Weakness<br>Opportunity<br>Threat |
| Who | Thing<br>Subjective reality<br>Physical reality<br>Property<br>Relationship<br>Framework<br>Actor<br>Organization<br>Human<br>Action<br>Service<br>User |

In addition to the top-level terms as such, there must also be the definition of the relationships among them. The semantics of these relationships, as defined by the author, are presented in Table 2.

**Table 2**. The Semantics of Relationships in the Onto6 Meta-ontology

| **Relationship between terms** | **Semantics** |
|---|---|
| [What, Where]<br>[What, When]<br>[What, How]<br>[What, Why]<br>[What, Who] | Relationship [T1, T2] defines specific attributes characterizing the cell in two-dimensional informatization grid. The dimensions of the grid are T1 and T2, where T1=What and T2= one of (Where, When, How, Why, Who) |

Relationships are symmetrical, i.e., the relationship [T1, T2] is the same as the relationship [T2, T1].

## 3.2 A Meta-ontology Instance

In order to provide an informatization-based description of a particular domain, an instance of the meta-ontology must be created. This meta-ontology instance reflects the characteristic aspects of a particular domain. The initial ontology of the domain is created on the basis of this meta-ontology instance.
A meta-ontology instance is created by perhaps eliminating some of the top-level terms from the meta-ontology.

### 3.2.1 Input for creating a metaontology instance

The domain of informatization must have a description of the existing situation – usually in terms of a document that is prepared by a working group. This description serves as the main input for creating the meta-ontology. Another input is a documented statement of goals – the vision of the desirable situation in the future.
In situations when there is lack of documents describing the situation and stating the goals the input needed can be obtained by organizing interviews or brainstorming sessions with representatives of interested organizations.

### 3.2.2 Stages in the creation of a metaontology instance

There are several stages in the creation of a meta-ontology instance:
1) Analysis of input documents. At this stage, the matches between the entities of input documents and the corresponding terms in the meta-ontology are determined;
2) Marking of insignificant terms in the meta-ontology. The threshold parameters hw and ht are applied to eliminate insignificant words and terms. The parameter hw is defined as the percentage of appearance vis-à-vis the appearance of the most frequent word. The word w is marked as insignificant if it occurs less often than hw does; The parameter ht is defined as the percentage of appearance vis-à-vis the appearance of the most frequent term. The term t is marked as insignificant if it occurs less often than ht does;
3) Creation of the meta-ontology instance. The terms which remain in the meta-ontology after the elimination of insignificant terms in the second stage are chosen for the meta-ontology instance.
The resulting meta-ontology instances can differ, depending on the values of hw and ht.

## 3.3 The Initial Ontology

The root metaphors of the initial ontology and the relationship among those metaphors must be defined by the informatization planner. The definition process uses the top level terms included in the meta-ontology instance. The attributes of the terms are analyzed – the corresponding instances significance is evaluated. The process of attributes evaluation is similar by its structure to the process of elimination of insignificant terms during the creation of the meta-ontology instance. For the attribute evaluation the same threshold parameter criterion can be used as the parameter used for the elimination of insignificant terms during the creation of the

meta-ontology instance. Usually some weights are applied to the attribute evaluation. Most significant instances are included in the initial ontology.
The visual representation of the initial ontology usually is chosen by the domain experts taking into account their previous representations of facts concerning that particular domain.

### 3.4 Refinements of the Initial Ontology

After the initial ontology is created further work has to be done by informatization implementers refining the initial ontology. The refinement process uses some instance included in the initial ontology. The instance is considered itself as a term with possible attributes. The process of adding the relevant attributes with their instances resembles the process of creating the meta-ontology instance but takes place at more detailed level. The refinement process usually is being applied to several terms leading to several refinements of the initial ontology. In this sense, an ontology cluster which consists of a meta-ontology, a meta-ontology instance, an initial ontology, and the refinements of that initial ontology – these come together to reflect the conceptualisation of informatization in a particular domain. [SB06] describes the creation of initial ontologies and their refinement, as conducted with the participation of this author for several different domains.
The essential phases of the Onto6 methodology proposed by the author are seen in Figure 2.

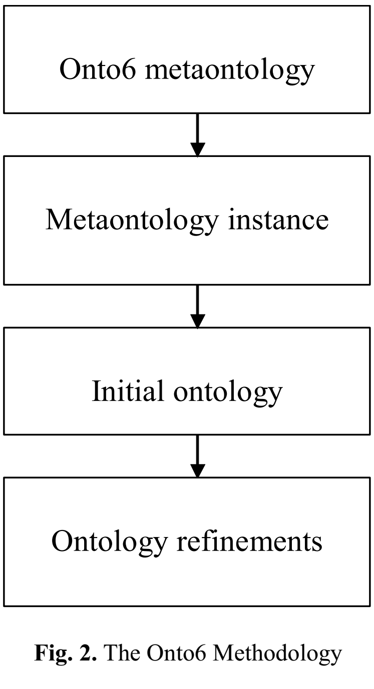


**Fig. 2.** The Onto6 Methodology

# 4 Analysis of Several Domains for Informatization

Next we can review the ontologies which the author has defined for three domains which have not changed during the informatization of the corresponding domain.

## 4.1 Application of Onto6 Methodology to Education Informatization

According to the general Onto6 methodology the informatization conceptualization for a particular domain should be created in several stages:

- Creation of the meta-ontology instance,
    - Analysis of input documents,
    - Marking of significant terms in meta-ontology,
    - Building the meta-ontology instance;
- Creation of the initial ontology,
- Refinement of the initial ontology.

### 4.1.1 Creation of the meta-ontology instance

Education informatization is a significant subprogram of the national “Informatics” program [BBB98]. As the main input document for the meta-ontology instance creation the “Informatics” program document was used. The document consists of 239 pages. For the analysis of the document a full text indexing is required. The indexing can take place either manually or automatically. The author has tried both approaches. Manual indexing is very time consuming. A convenient way of an automatic indexing is to use a Word macro counting the frequency of words in Microsoft Word document. The author has modified such a macro [GenC07] adding the specifics of processing Latvian language words and taking into account the large size of documents to be processed.

The national program “Informatics” document contains 9753 words. Author has taken the decision to consider only the words occurring in the text at least three times. The number of such words is 3303. From the remaining words the redundant words were stripped, such as: a, the, and, at etc. The number of remaining words was 2985. The threshold parameter hw = 10 was applied to the remaining words allowing to obtain the significant words in the input document. There were 28 significant words left. After determining the significant words the matches between the words in the input document and the corresponding terms in the meta-ontology were calculated. In the Table 3 the result of the analysis is presented.

**Table 3**. Significant Words in the Input Document "Informatics"

| Meta-ontology terms | Number of significant words |
|---|---|
| What | 19 |
| Where | 3 |
| When | 0 |
| How | 0 |
| Why | 3 |
| Who | 3 |

From the Table 3 it can be seen that the number of the significant words in term "What" substantially exceeds the numbers of the significant words in other terms. If we apply the threshold parameter ht = 40, we obtain very sparse meta-ontology instance consisting only of the term "What". Taking into account that the initial document describes the whole national program "Informatics" but our task is specifically aimed at a subprogram "Education", a further analysis was required.
Author has analyzed the chapter 6 "Education subprogram" of the national "Informatics" program. The document contains 2190 words. Only 426 words occur in the text at least three times. Stripping the redundant words, such as: a, the, and, at etc. 337 words were left. Again the threshold parameter hw = 10 was applied to the remaining words allowing to obtain the significant words in the input document. There were 73 significant words left. After determining the significant words the matches between the words in the input document and the corresponding terms in the meta-ontology were calculated again. In the Table 4 the result of the analysis is presented.

**Table 4**. Significant Words in the Input Document "Education"

| Meta-ontology terms | Number of significant words |
|---|---|
| What | 41 |
| Where | 17 |
| When | 2 |
| How | 2 |
| Why | 4 |
| Who | 7 |

From the Table 4 it can be seen that the numbers of the significant words in two terms "What"and "Where" substantially exceed the numbers of the significant words in other terms. Therefore we can apply a restricting threshold parameter ht = 40 thus retaining only the most significant terms for the meta-ontology instance. The resulting meta-ontology instance is seen in Fig 3.

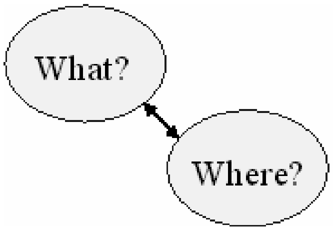


**Fig. 3.** The Meta-ontology Instance for the Informatization of Education

### 4.1.2 Initial ontology creation

The next step was to develop the initial ontology for the informatization of education, basing this on the meta-ontology instance. The requirements of the national "Informatics" programme were taken into account [BBB98]. The resulting ontology is seen in Figure 4.

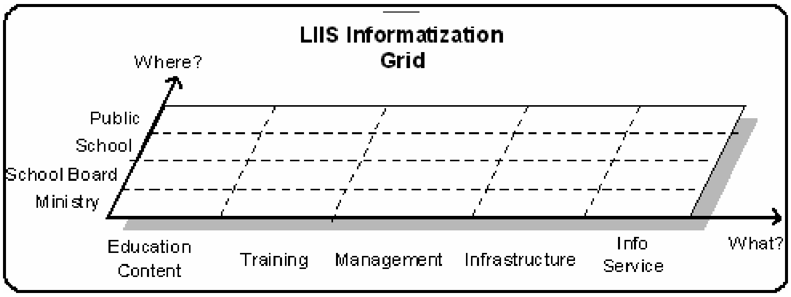


**Fig. 4.** The Initial Ontology for the Informatization of Education

The informatization of education was co-ordinated under the framework of the Latvian Education Informatization System (LIIS) project and on the basis of the initial ontology.

### 4.1.3 Refinements of the initial ontology

The initial ontology was refined several times during the implementation of the project to reflect more concrete aspects of the informatization grid that was presented at first. Figure 5 shows a typical example of how the initial ontology was refined in the context of information services.

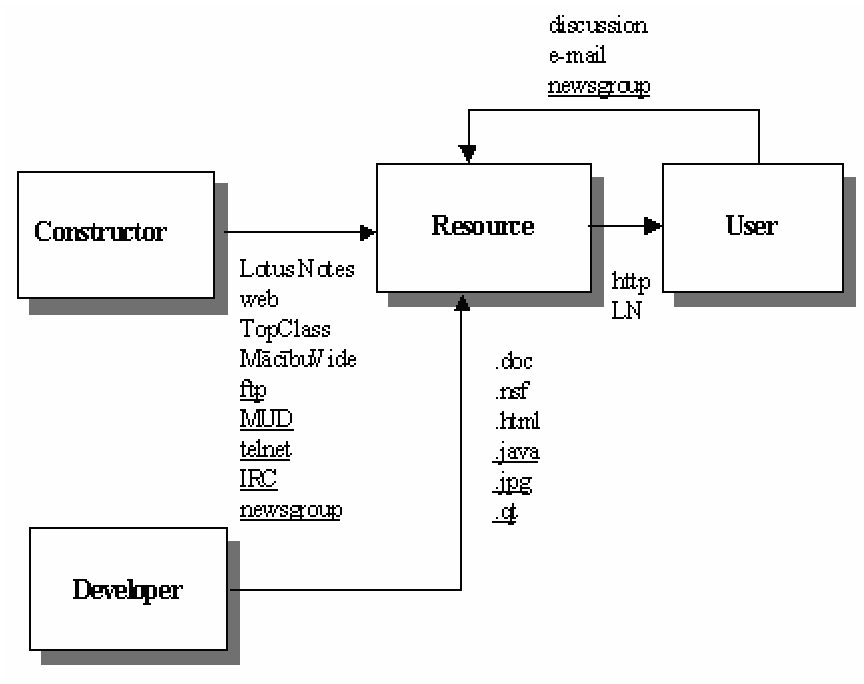


**Fig.5.** Refinement of the Initial LIIS Ontology for Information Services

Another refinement is seen in Figure 6. A structure of the education system is presented on the basis of the initial ontology, taking into account the fact that the same structure of the education informatization system is being implemented at all stages and levels of education.

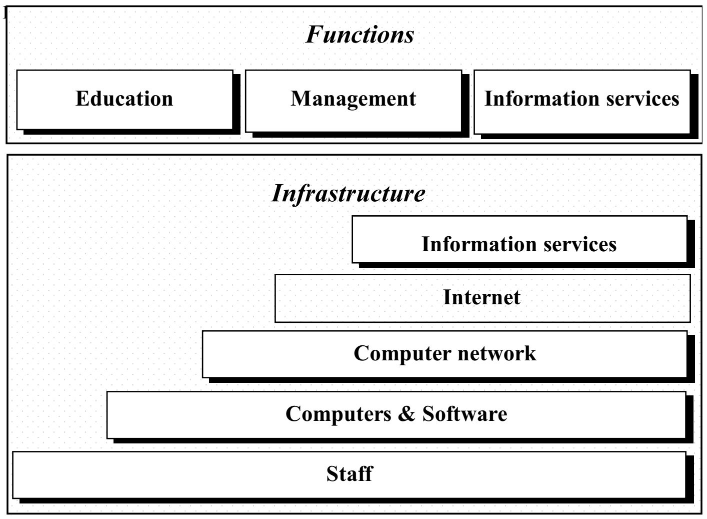


**Fig. 6.** Refinement of the initial LIIS ontology for the structure of the education informatization system

Many different activities are necessary to maintain the structure of the education system. Certain goals must be achieved – measurable improvements to infrastructure, enhancement of management services, supply of teaching materials, etc. Existing resources and routines must be used effectively, and activities aimed at the creation of new resources must be introduced and scheduled. The organisational aspects of planning the use of educational materials can be seen in Figure 7.

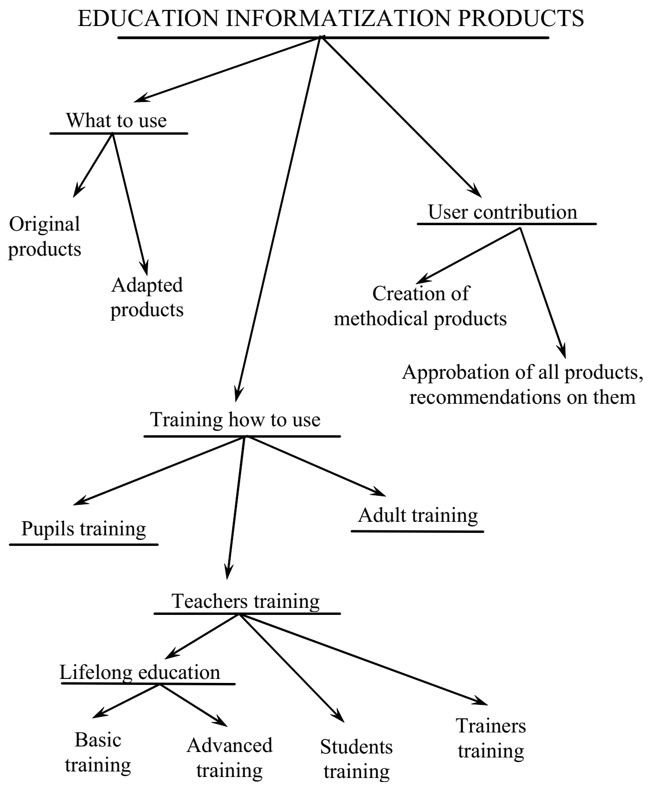


**Fig. 7.** Refinement of the Initial LIIS Ontology for the Organizational Aspects of Informatization

The proposed refinements of the initial otology differ in their form of representation and in their content. The authors recognise, however, that they served as the leading motif during the implementation of the LIIS project between 1997 and 2005. No significant changes were made to the initial ontology instead the fact that the implementation of the informatization permanently creates the needs for further refinements of the initial ontology.

## 4.2 Application of the Onto6 Methodology to the Unified State Library Information System (VVBIS) Project

The goal of the VVBIS project is to establish a harmonised information system for national and public libraries, one which allows them to make use of the opportunities that are offered by modern information technologies [AVGK01]. These are the main functions of the VVBIS:

- Searching for information. The system allows users to seek out information of interest, and it must ensure access to sources of information such as books, documents, publications, government registers, statistics, and transnational sources of information.
- Ordering of information: Once search results are obtained, the user must be able to order the necessary book, report or document, and alternative means for delivering the documents must be established.
- Delivery of information: The system for delivering books, documents, publications, multimedia items, and other sources of information must ensure not only that users can use the information at a library, but also that copies of publications can be supplied in print or electronic form. The world's leading document delivery centres must be used, and supply centres must also be established in Latvia so as to ensure that information sources can be supplied to users in Latvia and beyond This must also involve the services of interlibrary systems.
- Services: The library must be able to provide information services to users, offering information about all kinds of subjects – the law, culture, education, etc. Librarians must become information brokers who are aware of sources for all kinds of information and who can evaluate both the status of the service (basic, value-added, fee-based, limited, etc.) and the scope of the service. Libraries must offer public access to universal information services.
- Creating resources: Libraries must be aware of local information resources which are of lasting value and can be offered to users (including remote users). This is because libraries are a component in the national heritage. The resources must be digitalised, with databases concerning the history of the relevant region, tourism destinations, etc. Unique documents must be copied in electronic form.
- Training of librarians and users: Librarians must be aware of their mission as intermediaries between users and information. They can greatly enhance the process of obtaining information, thus becoming very important members of the Information System. It is vitally important, therefore, to improve the system under which librarians are trained. Librarians must learn how to do the various things that are a part of their job. Libraries must establish user training systems, too, so that users can work more independently.
- Marketing of information: Information is increasingly becoming a product. Research must be conducted to determine the need for value added services, while existing services must be tested and advertised.
- Unified user registration. Users can receive information from many different libraries, and the existing system of user registration must be changed to

reflect this. People must have individual codes which allow for clear identification, examination of their relationship to other libraries, and specification of their right to receive information. Libraries must accept anyone who is seeking universal information services.

- Establishment of a national bibliography and conversion of retrospective data: Because Latvia's informational resources are part of an international framework, they must be catalogued and made available through internationally accepted formats for the exchange of metadata. All materials which have been published in Latvia must be catalogued, with retrospective cataloguing of information resources. This must be done in accordance with demand for such resources.

The meta-ontology instance which was created for the VVBIS on the basis of the aforementioned requirements is shown in Figure 8.

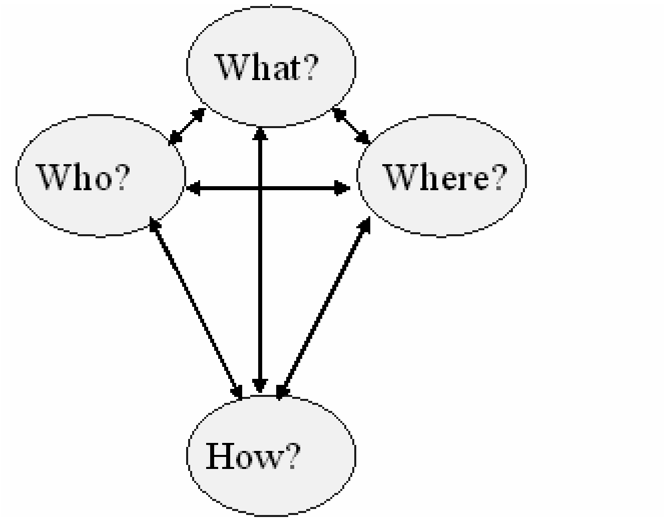


**Fig. 8.** The Meta-ontology Instance for the Unified State Library Information System

Refinements to the initial ontology for data flows in the Unified State Library Information System project are shown in Figure 9. This schema remained unchanged during the creation of the VVBIS concept.

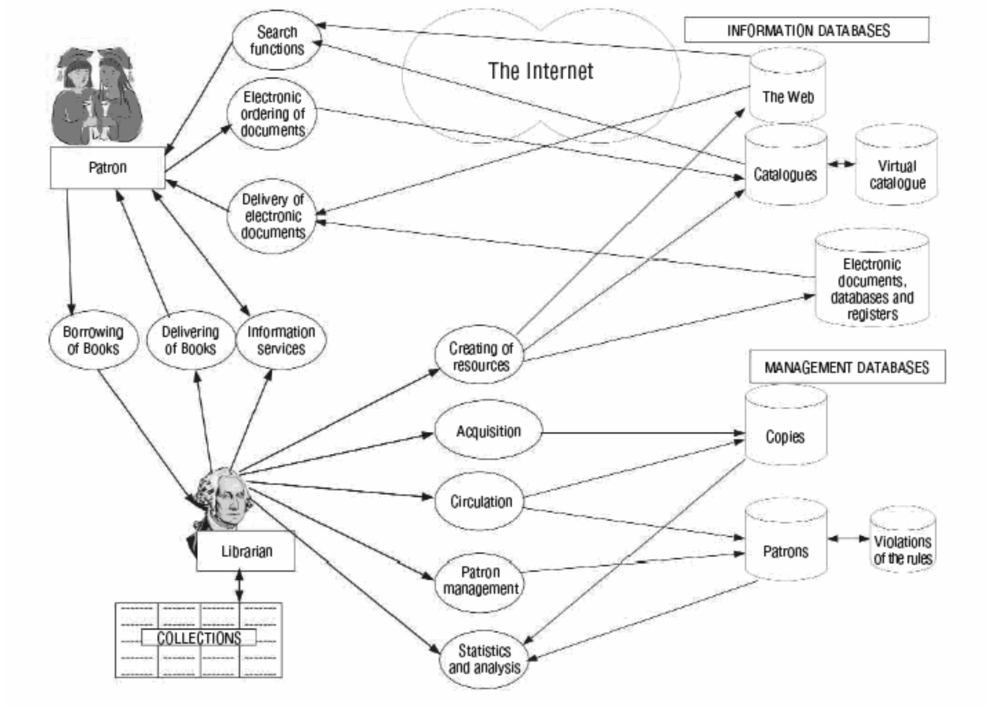


**Fig. 9** Refinement of the Initial VVBIS Ontology for Data Flow

### 4.3 Application of the Onto6 Methodology to the Informatics Curriculum Standard at Latvia's General Education Schools

The standard for the informatics curriculum for Latvia's general education schools was prepared in 2002 by a team which included this author [VBS03]. The standard was based on a questionnaire which determined the frequency of ICT usage at that time and the relevant existing needs. The respondents were divided into three groups – infrequent users of ICT, everyday users of ICT, and ICT professionals. The results were used to propose new content for the teaching of information. The standard for the curriculum is in line with the requirements of the European Computer Driving Licence (ECDL), with a few departures that are based on the needs and traditions which exist in Latvia. Because of the need for activities in support of the introduction of the new curriculum, the development team also created a timetable for a transitional period to run from 2003 to 2005. Special attention was focused on the training of existing and future informatics teachers. Textbooks were written, lesson plans were drawn up and tested at schools, and the ECDL certification system was introduced for teachers.

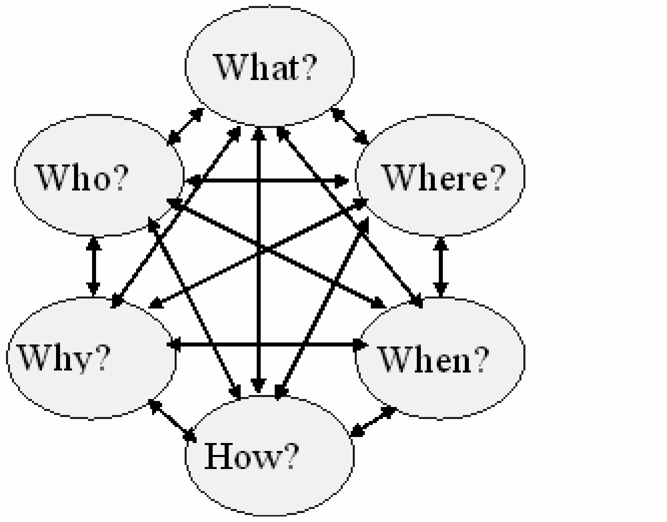


**Fig. 10.** The Meta-ontology Instance for the Informatics Curriculum Standard

The initial ontology for the informatics curriculum standard includes curriculum topics, their relationship to ECDL topics, as well as links to the relevant textbooks and lesson plans.

## 5 Common Methodology for Informatization

This author has dealt with many projects related to informatization, with the scope of domains ranging from the national level to an individual educational institution. The projects have been successful in creating positive changes in the planned timeline.
The author's approach is to analyse the nature of properties in a semi-formal way. The resulting ontology cluster is then checked for comprehensibility and relevance. This approach combines the general methodology for ontological analysis [WG01] with the specific requirements for information systems [2290, DR02, Gre05].
The essence of the proposed Onto6 methodology is to use the meta-ontology to create a meta-ontology instance which is relevant to the problematic domain. Then the initial ontology is developed on the basis of situational analysis and agreement among team members. An incremental plan is set up to refine the various aspects of informatization which are presented in the initial ontology. The result is a cluster of ontologies. Levels of formalisation and unification can differ during the development process, that depends on the potential end user.

## Conclusion

The Onto6 methodology which is proposed in this paper identifies objects, determines their interaction, and specifies their functionality. The methodology is based on a meta-ontology, and it involves the creation of an instance in accordance with the relevant domain. The initial ontology is created on the basis of the instance ontology, and it is then extended during the process of informatization. The planner of informatization must define the root metaphors of the initial ontology and the relationship among them. The implementers of informatization then do more work in refining the initial ontology. This creates an ontology cluster which consists of the meta-ontology, the meta-ontology instance, the initial ontology, and the refinements of the initial ontology. These reflect the conceptualisation and informatization of a particular domain.

## References

[AVGK01] Aldis Abele, Andris Vilks, Arnis Gulbis, Arts Klints, Dzintra Mukane, Gatis Pogulis, Ieva Vitolina, Ivars Indans, Janis Bicevskis, Jurgis Kirsakmens, Margarita Marcinkevica, Sandra Ozolina, Uldis Straujums. A Unified Information System for Latvia's Libraries. In *Baltic IT Review* 1(20), 2001, pp. 40-46.

[BAIMS04] Janis Bicevskis, Agnis Andzans, Evalds Ikaunieks, Inga Medvedis, Uldis Straujums, Viesturs Vezis. Latvian Education Informatization System LIIS. In *Education Media International*, 41:1, Routledge, Taylor & Francis Group, 2004, pp. 43 – 50.

[Bar94] Barsalou, Lawrence, W. Flexibility, Structure, and Linguistic Vagary in Concepts; Manifestations of a Compositional System of Perceptual Symbols. In *Theories of Memory*, edited by Alan F. Collins, Susan E. Gatherhole, Martin A. Convey and Peter E. Morris. Memory Research Unit, Lancaster University, UK, 1994, pp. 29-101.

[BBB98] National Program Informatics (in Latvian). Contributors: R.Balodis, J.Barzdins, J.Bicevskis, J.Borzovs, V.Briedis, E.Karnitis, V.Lauks, J.Mikelsons, A.Virtmanis, K.Zeila. Riga, 1998.

[CM04] Monica Crubézy, Mark A.Musen. Ontologies in Support of Problem Solving. In *Steffen Staab and Rudi Studer (Eds.). Handbook on Ontologies*, Springer-Verlag, 2004.

[DR02] Brian O'Donovan, Dewald Roode.A Framework for Understanding the Emerging Discipline of Information Systems. In *Information Technology & People*, Volume 15, Number 1, 2002, pp. 26-41.

[Fon02] Fonseca, Frederico, Martin, James, Rodriguez, Andrea. From Geo to Eco-Ontologies. 2002, 16p

[Gaz02] Gazendam, Henk, W. M. "Information System Metaphors.", The Journal of Management and Economics, Vol. 3, No. 2, Buenos Aires, Argentina: University of Buenos Aires, 1999, 20p. Revised Edition, 2002, 25p.

[GenC07] Generating a Count of Word Occurrences. Available: http://wordtips.vitalnews.com/Pages/T1510_Generating_a_Count_of_Word_Occurrences.html

[Goed99] Goedvolk, Hans, de Bruin, Hans, Rijsenbrij, Daan. Integrated Architectural Design of Business and Information Systems. Vrije Universiteit, Amsterdam, The Netherlands, 1999, 16p.

[GRA02] The Complex Method GRAPES. 2002. Available: http://www.informatik.uni-bremen.de/gdpa/methods/m-grapes.htm

[Gre05] Peter Green. Ontological Analysis of Business Systems Analysis Techniques, In Business Systems Analysis with Ontologies. UQ Business School, Australia; Queensland University of Technology, Australia, Idea Group Publishing, 2005., 27p.

[Gru95] Thomas Gruber. Towards Principles for the Design of Ontologies Used for Knowledge Sharing. In International Journal of Human-Computer Studies, 43(5/6), 1995, pp. 907-928.

[Gua98] Nicola Guarino. Formal Ontology and Information Systems. In *N.Guarino (ed). Formal Ontology and Information Systems. Proceedings of FOIS'98*, Trento, Italy, 6-8 June 1998. IOS Press, Amsterdam, pp. 3-15.

[Haf03] Nancy J. Hafkin. Gender Issues in ICT Statistics and Indicators, with particular Emphasis on Developing Countries [online]. Statistical Commission and United Nations Economic Commission for Europe (UNECE) Conference of European Statisticians. 2003. Available: http://www.unece.org/stats/documents/ces/sem.52/3.e.pdf

[Hoss06] Hoss, Allysson M. Ontology Based Methodology for Error Detection in Software Design. Louisiana State University, USA, 2006, 147p.

[Kipl02] Kipling, Rudyard. The Elephants Child. 1902.

[KMO07] Knowledge Management Online. Available: http://www.knowledge-management-online.com

[Lan04] Lan, Ju-Hung, A Preliminary Study of Knowledge Management in Collaborative Architectural Design.CAADRIA2004, Seoul, Korea, 2004, p.35-p.47

[Lep05] Leppänen M., An Ontological Framework and a Methodical Skeleton for Method Engineering, Dissertation Thesis, Jyväskylä Studies in Computing 52, University of Jyväskylä, 2005, 708 p. Available: http://dissertations.jyu.fi/studcomp/9513921867.pdf

[Lim01] Lim, S. K. A Framework to Evaluate the Informatization Level. In *Gremberg, W. (ed). Information Technology Evaluation: methods & management*. Hershey: IGP, 2001.

[MABM05] Juris Mikelsons, Agnis Andzans, Janis Bicevskis, Inga Medvedis, Andrievs Niedra, Uldis Straujums, Viesturs Vezis, Leo Truksans. ICT in Latvian Education – LIIS approach. In *The 3-rd International Conference on Education and Information Systems: Technologies and Applications EISTA'05 Proceedings*, Volume II, Orlando, Florida, USA, 2005, pp. 94 – 98.

[Mot00] Motschnig-Pitrik R. Contexts as Means to Decompose Information Bases and Represent Relativized Information. Workshop on Context of CHI 2000, April, Den Haag, Netherlands.

[PCP07] Principia Cybernetica Project. Available: http://pespmc1.vub.ac.be

[PKB93] Podnieks K., Kalnins A., Barzdins J. GRADE V1.0: Modelling and Development Environment for GRAPES-86 and GRAPES/4GL: Language description. Siemens Nixdorf, Munich, Germany, 1993, 246p.

[PR07] Protégé. An ontology editor. [online]. Available: http://protege.stanford.edu

[Pul03] Pulkkinen, Jyrki. The paradigms of e-Education. An analysis of the communication structures in the research on information and communication technology integration in education in the years 2000–2001, 2003, 175p. Available: http://herkules.oulu.fi/isbn9514272463/isbn9514272463.pdf

[Rum97] Rumbaugh, James. OMT Insights: Perspective on Modeling from the Journal of Object-Oriented Programming (SIGS Reference Library). Cambridge University Press, 1997, 412p.

[SB06] Straujums, Uldis, Bicevskis, Janis. Ontologic Aproach to Informatization. In *Databases and Information Systems, Seventh International Baltic Conference on Databases and Information Systems, Communications*, Vilnius, Lithuania, 2006, pp. 276 – 287.

[Sowa06] Sowa, John F. A Dynamic Theory of Ontology. In *Formal Ontology in Information Systems*, edited by B. Bennett & C. Fellbaum, IOS Press, Amsterdam, 2006.

[Sowa07] Sowa, John F. Knowledge Representation. Available: http://www.jfsowa.com/krbook

[Str05] Uldis Straujums. Shaping of Learning Process Using Ontologies. University of Latvia, 63-rd scientific conference, Riga, 2005.

[VB01] De Vasconcelos, José Ângelo Braga. An Ontology-Driven Organisational Memory for Managing Group Competencies. The University of York, 2001, 306p.

[VBS03] Viesturs Vezis, Janis Bicevskis, Uldis Straujums. Computer Literacy Acquisition Strategy in Latvia: problems and solutions. BalticIT&T2003 conference, Riga, 2003

[VZSS04] Antonio Geraldo da Rocha Vidal, Ronaldo Zwicker, José de Oliveira Siqueira, Cesar Alexandre de Souza. Informatization in Brasilian Companies: an exploratory study. University Sao Paolo, 2004, 24p. Available: http://www.ead.fea.usp.br/WPapers/2004/04-001.pdf

[WG01] Christopher Welty, Nicola Guarino. Supporting Ontological Analysis of Taxonomic Relationships. In *Data and Knowledge Engineering*, 39(1), 2001, pp. 51-74.

[Wil04] Ruth Wilson. The Role of Ontologies in Teaching and Learning [online]. JISC Technology and Standards Watch Report TSW0402, 2004. Available: http://www.jisc.ac.uk/index.cfm?name=techwatch_reports_0402

[WW90] Y. Wand, R. Weber. An Ontological Model of an Information System. In IEEE Transactions on Software Engineering, November 1990 (Vol. 16, No. 11), pp. 1282-1292.

[Wys04] Boris Wyssusek. Ontology and Ontologies in Information Systems Analysis and Design: A Critique. In *Proceedings of the Tenth Americas Conference on Information Systems*, 2004., pp. 4303-4308.

LATVIJAS UNIVERSITĀTES RAKSTI. 2008, 733. sēj.:
DATORZINĀTNE UN INFORMĀCIJAS TEHNOLOĢIJAS 261.–288. lpp.

# An Approach to Cadastral Map Quality Evaluation in the Republic of Latvia

Anita Jansone, Juris Borzovs

University of Latvia, Raiņa bulvāris 19, Rīga,. Latvija, LV-1586
anita.jansone@vzd.gov.lv, juris.borzovs@lu.lv

**Abstract.** An approach to cadastral map quality evaluation is proposed, which is elaborated and implemented by State Land Service of the Republic of Latvia. The approach is based on opinion of Land Service experts about cadastral map quality that depends on its usage points. Quality parameters of cadastral map objects identified by experts and its limit values are used for evaluation. The assessment matrix is used, which allow to define cadastral map quality that depends on its usage purpose. The matrix is used to find out, of what quality a cadastral map should be in order to be used for the chosen purpose. The given approach is flexible, it gives a possibility to change sets of quality parameters and their limit values as well as to use the approach for other type data quality evaluation.

## 1 Introduction

Scientific literature identifies several aspects of quality: data quality has several components such as accuracy, relevance, timeliness, completeness, reliability, accessibility, precision, consistency, etc. [1], [2]. There are currently two main research streams, which address the problem of ensuring a high level of data and information quality. One is a technical, database-oriented approach, while the second is a management and business-oriented approach. Engineering of information system brings both streams together and addresses issues related to the design and modeling of information systems [3]. Geographical data are data describing an object's spatial location and various properties. High quality geographical data will include space location and object properties at given times (where-what-when) [4].

Data quality is the degree to which data meet the specific needs of a specific customer. Note that one customer may find data to be of high quality (for one use of the data), while another finds the same data to be of low quality (for another use) [5]. What features do experts working with geographical data (data entry, map drawing, supervision of maps, etc.) use to judge the quality of data? The authors are not aware of any published studies in this area to date. This paper presents an approach to the evaluation of the quality of cadastral map that caters for the differing levels of quality required of various parameters in order to meet different goals.

The subjective assessments of experts in geographical data processing are sought to determine the factors which have the most impact upon the quality of geographical data. When these assessments are evaluated, freed from subjective elements and, classified, it becomes possible to specify parameters for the evaluation of data quality, their values, and the required levels of quality. The result of this is a matrix for quality assessment which can be used to determine the data quality level that is necessary for specific purposes or, alternatively, the specific goals for which data at a specific level of quality may be used.

This paper describes the method that is to be uses in preparing the quality assessment matrix and how this approach is used for cadastral map evaluation in State Land Service of the Republic of Latvia.

## 2 An Approach to Data Quality Evaluation

The discussion of quality must begin with the identification of the objects of interest. Every object will have a number of quality parameters (QP1, QP2, etc.) (Fig.1). Each quality parameter QPn has values taken from one or more sets of values QPnVSk (Table 1), where QPnVS1 may contain the best values. QPnVS2 contains the second best values for some particular goal, etc. [6]

The quality of the object is based upon several or all quality parameters. For instance, an object can belong to the highest level of quality if all of the estimated values of the relevant quality parameters belong to the best sets of values. It belongs to the second level of quality if the values of the relevant quality parameters belong to the second best sets of values, etc.

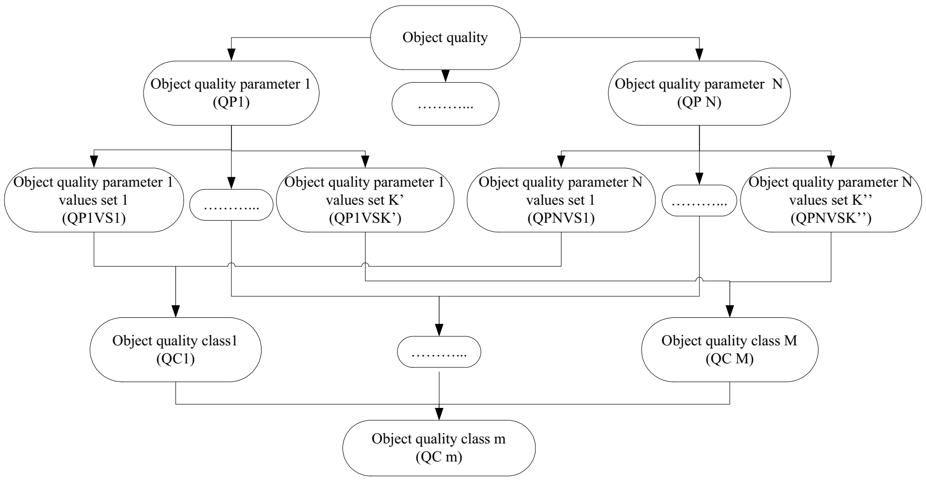

Object quality
Object quality parameter 1 (QP1)
………...
Object quality parameter N (QP N)
Object quality parameter 1 values set 1 (QP1VS1)
………...
Object quality parameter 1 values set K' (QP1VSK')
Object quality parameter N values set 1 (QPNVS1)
………...
Object quality parameter N values set K'' (QPNVSK'')
Object quality class1 (QC1)
………...
Object quality class M (QC M)
Object quality class m (QC m)

**Fig. 1.** An Approach to Data Quality Evaluation

**Table 1.** Quality Parameter Value Set

| Quality parameter (QP) | Quality parameter value set (QPVS) | | | |
|---|---|---|---|---|
| | QPnVS1 (high) | QPnVS2 | ... | QPnVSk (low) |
| QPn | from-until | from-until | ... | from-until |

As a result the object quality evaluation matrix (Table 2) is obtained, which is used to determine, which quality class the object belongs to, as well as to determine, which should be quality parameter values so that the object would correspond to the chosen aim of use.

**Table 2.** Quality Assessment Matrix

| Object quality class (QC) | Quality parameter/ Quality parameter value set | | | |
|---|---|---|---|---|
| | QP1 | QP2 | ... | QPn |
| QC1 (high) | QP1VS1 | QP2VS1 | ... | QPnVS1 |
| QC2 | QP1VS2 | QP2VS2 | ... | QPnVS2 |
| ... | ... | ... | ... | ... |
| QCm (low) | QP1VSk’ | QP2VSk’’ | ... | QPnVSk’’’ |

Quality parameter quality class (QP_QC) depends on a quality parameter value set, to which belongs the quality parameter value (Formulae 1).

$$QPn\_QC=1, \text{ if } QPn \in QPnVS1;\ 2, \text{ if } QPn \in QPnVS2, \ldots, M, \text{ if } QPn \in QPnVSk,\ n=\{1...N\},\ k=\{1...K\} \qquad (1)$$

In its turn, object corresponds to the lowest quality parameter quality class (“hard” principle for object evaluation) (Formulae 2).

$$QC=\text{lowest}\ (QP1\_QC, QP1\_QC, \ldots , QP\ N\_QC,) \qquad (2)$$

The aim of object quality evaluation is to determine, which quality class the object belongs to and which aims it can be used for. In order to evaluate an object (Fig. 2):

a) check the correspondence of an object to quality criterions (Fig. 2, P1), obtain the list or the number (QPn_list, QPn_count) of items not corresponding to the quality criterions,

b) evaluate each object quality according to quality parameters and obtain a quality class:

– calculate object quality parameter values (Fig. 2, P2), obtain QPn,
– determine, which parameter value set (Table 1) it belongs to (Fig. 2, P3), obtain QPnVSk,
– determine, which quality class the value belongs to (Table 2, Formulae 1) (Fig. 2, P4), obtain quality parameter class QPn_QC,
– determine object quality class (Formulae 2) (Fig. 2, P5), obtain Object QC

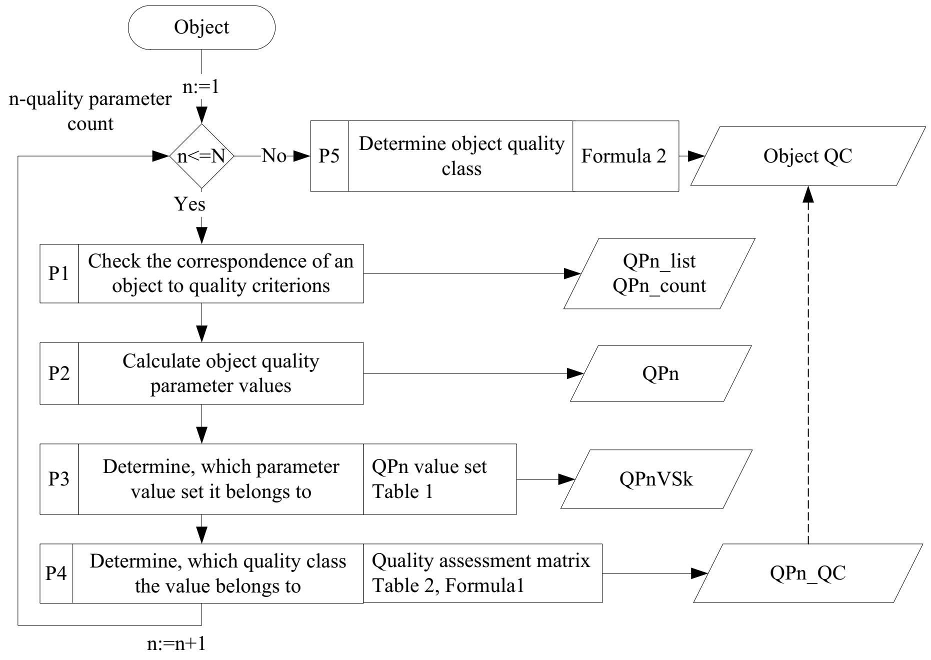

Object
n:=1
n-quality parameter count
n<=N
No
Yes
P5
Determine object quality class
Formula 2
Object QC
P1
Check the correspondence of an object to quality criterions
QPn_list
QPn_count
P2
Calculate object quality parameter values
QPn
P3
Determine, which parameter value set it belongs to
QPn value set Table 1
QPnVSk
P4
Determine, which quality class the value belongs to
Quality assessment matrix Table 2, Formula1
QPn_QC
n:=n+1

**Fig. 2.** Object Quality Class

This approach is implemented in State Land Service (SLS) of the Republic of Latvia for cadastral map evaluation and is based on the defined by field experts quality parameters, which describe the usage purpose of a certain cadastral map.

## 3 Cadastral Map Quality Evaluation in the Republic of Latvia

In the Republic of Latvia, cadastral map (CM) is created in Latvian coordinate system LKS-92 in Transverse Mercator (TM) projection. The following elements are represented in CM: land parcels- boundaries of parcels and their cadastral designations; buildings- outlines of buildings and their cadastral designations; encumbrances- areas occupied by encumbrances of right to use real property and their designations; parts of land parcels- leaseholds and their cadastral designations; boundaries of cadastral territories and cadastral groups. The CM is used to locate cadastral objects with precision so that any changes in boundaries for administrative or other purposes may be accurately described and to describe the relationships between objects for the purposes of environmental and town planning and for various reports. The principles and content of the CM are established by Regulation, which is an ordinance of the SLS of Latvia. The Cadastral IS databases consist of two parts:

the textual part (TP) and the graphical part, which includes the CM in vector graphics form [7].

CM quality depends on the quality of each object, whereof the CM is made. CM can consist of such objects as land parcel, building, encumbrance and part of land parcel. Therefore, in order to evaluate CM quality, firstly, it is necessary to evaluate qualities of land parcel, building, encumbrance and part of land parcel – wherewith the approach described above (Fig. 1) has to be applied for each CM object.

## 3.1 Cadastral Map Objects Quality Parameters

In this article an approach to CM quality evaluation is proposed, which is based on experts' opinions about CM quality that depends on its usage points. Expert opinions are obtained from more than 50 expert interview surveys. Having summarized the results of surveys, such quality criteria are obtained: the CM meets the legal regulation requirements, CM objects are topologically correct, coordinates of CM land parcels are precise, CM objects (land parcels, building, encumbrance and part of land parcel) are in both Cadastral databases and the data is the same – in the TP and in the CM. Quality criteria are given in Table 3.

**Table 3.** Cadastral Map Quality Criteria

| Code | Title |
|---|---|
| C1 | CM meets the legal regulation requirements |
| C2 | CM objects are topologically correct |
| C3 | Coordinates of CM land parcels are precise |
| C4 | Object data in the TP and the CM are identical: |
| C4.1 | A cadastral object (land parcels, building, encumbrance and part of land parcel) has to be in both Cadastral databases – in the TP and in the CM: |
| C4.1.1 | the object marked in a CM has to be in the TP |
| C4.1.2 | the object in a TP has to be marked in the CM |
| C4.2 | Cadastral object data in both Cadastral databases: |
| C4.2.1 | the surveying type for land parcel has to be the same in both databases |
| C4.2.2 | cadastral surveyed land parcels' and parts of land parcels' legal area (indicated in the documents) and area defined by graphical methods (marked in the cadastral map, further in the text – geographical area) cannot be larger or smaller than the acceptable space difference defined in the Regulations |
| C4.2.3 | a building, in both databases, has to be attached to one and the same land parcel |

Experts' opinions about CM quality are subjective and therefore have to be structured and, according to normative acts and existing IT solutions in State Land Service, we obtain cadastral object quality parameters (QPn) (Fig. 1) – for land parcel (LP) 5 quality parameters are defined (LP_QPn, n=1…5) (Table 4), for building (BD) – 4 quality parameters (BD_QPn, n=1…4) (Table 7), for encumbrance (EB) – 2 quality parameters (EB_QPn, n=1…2) (Table 8), for part of land parcel (PLP)– 3 quality parameters (PLP_QPn, n=1…3) (Table 9).

### 3.1.1 Land Parcel Quality Parameters

Land parcel quality is described by 5 quality parameters (Table 4).

**Table 4. Land Parcel Quality Parameters**

| Code | Description | Value High - low | Quality criteria |
|---|---|---|---|
| LP_QP1 | Describes how much (%) of CM land parcels are missing in the TP | 0%-100% | C4.1.1 |
| LP_QP2 | Describes how much (%) of TP land parcels are not marked in the CM | 0%-100% | C4.1.2 |
| LP_QP3 | Describes how much (%) of CM land parcels surveying type differs from TP surveying type | 0%-100% | C4.2.1 |
| LP_QP4 | Describes how much (%) of CM cadastral surveyed land parcels' geographical area is larger or smaller than the acceptable space difference of TP legal area | 0%-100% | C4.2.2 |
| LP_QP5 | Describes how much (%) of CM land parcels are cadastral surveyed | 100%-0% | C3 |

LP_QP1 and LP_QP2 characterize land parcels completeness in Cadastral IS TP and CM databases.

LP_QP1 describes how much (%) of CM land parcels are missing in the TP. Quality parameter values can vary from 0% (all the cadastral map land parcels are also in the textual part) to 100% (none of cadastral map land parcels are in the textual part). Quality parameter value is obtained by applying Formulae 3, where LP_QP1_count – number of cadastral map land parcels, which are not in the textual part, CM_LP_count – number of cadastral map land parcels.

$$LP\_QP1 = LP\_QP1\_count/CM\_LP\_count*100 \quad (3)$$

LP_QP2 describes how much (%) of TP land parcels are not marked in the CM. Quality parameter values can vary from 0% to 100%. Quality parameter value is obtained by applying Formulae 4, where LP_QP2_count – number of land parcels in the textual part, which are not in the cadastral map, TD_LP_ number – count of land parcels in the textual part.

$$LP\_QP2 = LP\_QP2\_count/TD\_LP\_count*100 \quad (4)$$

LP_QP3 characterizes land parcels survey type (Table 5) consistency between in TP and CM and describes how much (%) of CM land parcels surveying type differs from TP surveying type. Quality parameter values can vary from 0% to 100%. Quality parameter value is obtained by applying Formulae 5, where LP_QP3_count – number of cadastral map land parcels, which surveying type does not match the surveying type in the textual part, CM_LP_count – number of cadastral map land parcels.

$$LP\_QP3 = LP\_QP3\_count/CM\_LP\_count *100 \quad (5)$$

**Table 5.** Land Parcel Survey Types

| Survey type in CM | Survey type in TP |
|---|---|
| surveyed land parcels | – instrumental survey<br>– global positioning,<br>– photogram survey |
| allocated land parcels | – allocation<br>– allocation in orthophoto maps<br>– allocation in photoplan |
| designed land parcels | designed land parcel do not have survey type |

LP_QP4 characterize trusted land parcels area. In accordance with the Regulations for CM, the graphical area of a surveyed land parcel listed in the CM (which is calculated on the basis of coordinates) can possibly differ from the legal area of the land parcel shown in the TP (which is declared in legal documents) but within prescribed limits. The admissible level of variation is determined by Regulation (Table 6). LP_QP4 describes, how much (%) of CM cadastral surveyed land parcels' geographical area is larger or smaller than the acceptable space difference of TP legal area. Quality parameter values can vary from 0% to 100%. Quality parameter value is obtained by applying Formulae 6, where LP_QP4_count – number of cadastral map land parcels, which area is smaller or larger than the allowed difference of the legal area, CM_LP_count– number of cadastral map land parcels

$$\text{LP\_QP4} = \text{LP\_QP4\_count}/\text{CM\_LP\_count}*100 \qquad \textbf{(6)}$$

**Table 6.** The Allowed Area Difference of the Surveyed Land Parcel Graphical Area

1)in towns:

| Area (ha) | Up to 0.50 | 0.51-1.00 | 1.01-5.00 | 5.01-10.00 | 10.01-50.00 | 50.01-100.00 | More than 100.00 |
|---|---|---|---|---|---|---|---|
| The allowed difference (%) | ± 3.00 | 2.30 | 1.80 | 1.50 | 1.25 | 1.05 | 1.00 |

2)settlement, summer cottage and gardening areas, country region:

a) the difference, which is determined using formula $\pm 0{,}1\sqrt{P}$ (P – land parcel or part of land parcel area (ha)), if the area is not larger than 1.0 ha;

b) the difference, which is determined using formula $\pm 0{,}25\sqrt{P}$ (P – land parcel or part of land parcel area (ha)), if the area is larger than 1.0 ha;

c) the difference, which is determined using formula $\pm 0{,}3\sqrt{P}$ (P – land parcel or part of land parcel area (ha)), if the area is not larger than 200 ha.

Comment: This parameter relates only to cadastral surveyed land parcels, cadastral unsurveyed land parcels, t.i. cadastral allocated and cadastral designed land parcels, graphical area is not analysed because historically no conditions are proposed to it.

LP_QP5 characterize accuracy of land parcels co-ordinate. The database which includes the graphic component of the cadastral register includes graphic data to various levels of accuracy. The database of land parcels includes data at three different levels of data accuracy – surveyed land parcels, allocated land parcels, and designed land parcels. The coordinates of the surveyed land parcels are obtained by surveying the relevant parcel with the appropriate instruments. Coordinates of allocated land parcels may have been obtained with older measuring instruments that are no longer in use (field compasses, tape measures), or through conversion from other co-ordinate systems which differ from the specified LKS-92 TM coordinate system. The coordinates of designed land parcels are approximate, because they are usually obtained from orthophoto maps, photo plans or other materials. These coordinates are not based on direct land measurement. LP_QP5 describes, how much (%) of CM land parcels are cadastral surveyed. Quality parameter values can vary from 100.00% (all cadastral map land parcels are cadastrally surveyed) to 0% (none of cadastral map land parcels are cadastrally surveyed). Quality parameter value is obtained by applying Formulae 7, where LP_QP5_count – number of cadastrally surveyed land parcels in a cadastral map, CM_LP_count – number of cadastral map land parcels.

$$\text{LP\_QP5} = \text{LP\_QP5\_count}/\text{CM\_LP\_count} * 100 \qquad (7)$$

Comment: This quality parameter gives statistical information – how many land parcels are cadastrally surveyed. The most precise coordinates in the cadastral map and the most arranged textual data are cadastral for the surveyed land parcels, therefore – the more cadastral map land parcels are cadastrally surveyed, the higher the quality of cadastral map data is. However, SLS cannot affect cadastral map quality by this parameter, because it depends only on its owners and dealings with land parcels.

### 3.1.2 Building Quality Parameters

Quality of a building is described by 4 quality parameters (Table 7).

**Table 7.** Building Quality Parameters

| Code | Description | Value High - low | Quality criteria |
|---|---|---|---|
| BD_QP1 | Describes how much (%) of CM buildings are missing in the TP | 0%-100% | C4.1.1 |
| BD_QP2 | Describes how much (%) of TP buildings are not marked in the CM | 0%-100% | C4.1.2 |
| BD_QP3 | Describes how much (%) of CM buildings have different land parcel cadastral designation in TP, to which the building is attached | 0%-100% | C4.2.3 |
| BD_QP4 | Describes how much (%) of CM buildings are cadastrally surveyed | 100%-0% | C3 |

BD_QP1 and BD_QP2 characterize building completeness in Cadastral IS TP and CM databases. Both quality parameters values can vary from 0% to 100%.

BD_QP1 value is obtained by applying Formulae 8, where BD_QP1_count– number of cadastral map buildings, which are not in the textual part, CM_BD_count - number of cadastral map buildings.

BD_QP1 = BD_QP1_count/CM_BD_count*100 (**8**)

BD_QP2 value is obtained by applying Formulae 9, where BD_QP2_count- number of textual part buildings, which are not in the cadastral map, TD_BD_count– number of textual part.

BD_QP2 =BD_QP2_count/TD_BD_count*100 (**9**)

BD_QP3 characterizes building land parcel attachment consistency between TP and CM (in both databases the building has to be attached to one and the same parcel). BD_QP3 describes how much (%) of CM buildings has different land parcel cadastral designation in TP, to which the building is attached. Quality parameter value can vary from 0% to 100%. Quality parameter value is obtained by applying Formulae 10, where BD_QP3_count – number of buildings in the cadastral map, which designation of land parcel does not match with the designation of land parcel in the textual part, which it is attached to, CM_BD_count – number of buildings in the cadastral map.

BD_QP3 = BD_QP3_count/CM_BD_count*100 (**10**)

BD_QP4 characterize accuracy of building co-ordinate. The database which includes the graphic component of the cadastral register includes graphic data to various levels of accuracy. The database of building includes data at three different levels of data accuracy – surveyed building, stereo vectorized building, and vektorized building. The coordinates of the surveyed building are obtained by surveying with the appropriate instruments. A stereo vectorized building contour is marked by a stereo tool, but a vectorized building – by scanned material, the building is not surveyed. BD_QP4 describes, how much (%) of CM buildings are cadastral surveyed. Quality parameter values can vary from 100.00% to 0%. Quality parameter value is obtained by applying formulae 11, where BD_QP4_count – number of cadastrally surveyed buildings in the cadastral map, CM_BD_count – number of buildings in the cadastral map.

BD_QP4= BD_QP4_count/CM_BD_count*100 (**11**)

Comment. This quality parameter gives statistical information – how many buildings are cadastrally surveyed. The most precise coordinates in the cadastral map and the most arranged textual data have cadastrally surveyed buildings; therefore the more buildings in the cadastral map are cadastrally surveyed, the higher the quality of cadastral map data is. However, SLS cannot affect cadastral map quality by this parameter, because it depends only on its owners and dealings with land parcels.

### 3.1.3 Encumbrance Quality Parameters

Encumbrance quality is described by 2 quality parameters (Table 8).

**Table 8. Encumbrance Quality Parameters**

| Code | Description | Value high - low | Quality criteria |
|---|---|---|---|
| EB_QP1 | Describes how much (%) of CM encumbrances are missing in the TP | 0%-100% | C4.1.1 |
| EB_QP2 | Describes how much (%) of TP encumbrances are not marked in the CM | 0%-100% | C4.1.2 |

EB_QP1 and EB_QP2 characterize encumbrance completeness in Cadastral IS TP and CM databases. A CM for encumbrances has been drawn from the 1st of July 2002 and only road servitudes. Both quality parameters values can vary from 0% to 100%.

EB_QP1 value is obtained by using Formulae 12, where EB_QP1_count – number of cadastral map encumbrances, which are not in the textual part, CM_EB_count – number of cadastral map encumbrances.

$$\text{EB\_QP1} = \text{EB\_QP1\_count}/\text{CM\_EB\_count}*100 \qquad (\mathbf{12})$$

EB_QP2 value is obtained by applying formulae 13, where EB_QP2_count – number of encumbrances in the textual part, which are not marked in the cadastral map, TD_EB_count – number of encumbrances in the textual part.

$$\text{EB\_QP2} = \text{EB\_QP2\_count}/\text{TD\_EB\_count}*100 \qquad (\mathbf{13})$$

### 3.1.4 Part of Land Parcel Quality Parameters

Quality of part of land parcel is described by 3 quality parameters (Table 9)

**Table 9.** Part of Land Parcel Quality Parameters

| Code | Description | Value high - low | Quality criteria |
|---|---|---|---|
| PLP_QP1 | Describes how much (%) of CM parts of land parcels are missing in the TP | 0%-100% | C4.1.1 |
| PLP_QP2 | Describes how much (%) of TP parts of land parcels are not marked in the CM | 0%-100% | C4.1.2 |
| PLP_QP3 | Describes, how much (%) of CM cadastral surveyed parts of land parcels' geographical area is larger or smaller than the acceptable space difference of textual part legal area | 0%-100% | C4.2.2 |

PLP_QP1 and PLP_QP2 characterize part of land parcels completeness in Cadastral IS TP and CM databases. Both quality parameters values can vary from 0% to 100%.

PLP_QP1 value is obtained by Formulae 14, where PLP_QP1_count – number of parts of land parcels in the cadastral map, which are not in the textual part, CM_PLP_count – number of parts of land parcels in the cadastral map.

$$\text{PLP\_QP1} = \text{PLP\_QP1\_count}/\text{CM\_PLP\_count}*100 \qquad \textbf{(14)}$$

PLP_QP2 value is obtained by applying Formulae 15, where PLP_QP2_count – number of parts of land parcels in the textual part, which are not in the cadastral map, TP_PLP_count – number of parts of land parcels in the textual part.

$$\text{PLP\_QP2} = \text{PLP\_QP2\_count}/\text{TP\_PLP\_count}*100 \qquad \textbf{(15)}$$

PLP_QP3 characterizes trusted part of land parcels area. The purpose of this parameter is the same as that of quality parameter LP_QP5.

$$\text{PLP\_QP3} = \text{PLP\_QP3\_count}/\text{CM\_PLP\_count}*100 \qquad \textbf{(16)}$$

## 3.2 Cadastral Map Quality Class

In collaboration with experts and in the result of experiments, sets of quality parameter values are defined. There are three sets of values for all the parameters: excellent, good, and bad values (Fig. 1, QPnVSk, k=1...3).

Parameter values of excellent quality are such as ones, which describe that an object meets quality criteria; values of good quality are such as ones, which do not overrun the defined acceptable error rate, but values of bad quality are such as ones, which overrun the defined rate (Table 10). Parameter value of excellent quality to any quality parameter (except for land parcels and buildings) is 0%, but to the surveyed land parcels and buildings – 100%. Value of good quality to any quality parameter (except for land parcels and buildings) is from 0.01% to 5%, but to the surveyed land parcels and buildings – from 99.99% to 10%. Value of bad quality to any quality parameter (except for land parcels and buildings) is from 5.01% to 100%, but to the surveyed land parcels and buildings – from 9.99% to 0%.

**Table 10.** Quality Parameters Values Sets

| Quality parameters | Quality parameter values sets | | |
|---|---|---|---|
| | QPiVS1 excellent | QPiVS2 good | QPiVS3 bad |
| - LP_QP1, LP_QP2, LP_QP3, LP_QP4,<br>- BD_QP1,BD_QP2,BD_QP3,<br>- EB_QP1, EB_QP2,<br>- PLP_QP1,PLP_QP2, PLP_QP3 | 0% | 0.01-5.00% | 5.01-100% |
| - LP_QP5,<br>- BD_QP4 | 100% | 99.99%-10% | 9.99%-0% |

Theoretically, object quality parameters and sets of values can be chosen in thousands of variants, but practically, suitable is only such a variant, where parameters are defined by field experts that depends on what object (in this case – a CM) will be used for.

Taking into account the purpose of a CM and collaborating with experts, three quality classes of objects are defined (Table 11): high, medium and low (Fig. 1) QCm, m=1…3.

**Table 11.** Quality Classes

| Quality class | | Description |
|---|---|---|
| High | 1st quality class (QC1) | A CM can be used for making decisions and other activities, where information from the CM is needed |
| Medium | 2nd quality class (QC2) | A CM can be used for making decisions, but it is necessary to be sure about quality of a certain object, which is used for making the decision |
| Low | 3rd quality class (QC3) | A CM cannot be used for making decisions, it can be used to get primary information |

Having summarized quality parameter sets of values and quality classes, an object quality assessment matrix (Table 12) is obtained. According to quality parameter values, object quality is: High (QC1), if quality parameter value is excellent – appertains to the set of values QPnVS1. Medium (QC2), if quality parameter value is good – appertains to the set of values QPnVS2. Low (QC3), if quality parameter value is bad – appertains to the set of values QPnVS3.

**Table 12.** Object Quality Assessment Matrix

| Object quality class | | Quality parameters value set |
|---|---|---|
| High | 1st quality class (QC1) | QPnVS1 |
| Medium | 2nd quality class (QC2) | QPnVS2 |
| Low | 3rd quality class (QC3) | QPnVS3 |

The main principle of using the quality evaluation matrix – an object corresponds to its quality class, which the worst quality parameter value belongs to.

Land parcel quality class 'LP_QC' (Fig. 3) depends on the lowest quality parameter quality class (Formulae 17).

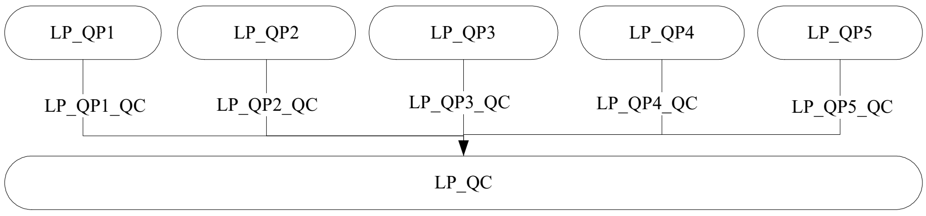


**Fig. 3.** Land Parcel Quality Class

$$LP\_QC = MAX(LP\_QPn\_QC), \quad \textbf{(17)}$$
$$LP\_QPn\_QC = 1, \text{ if } LP\_QPn \in QPnVS1;\ 2, \text{ if } LP\_QPn \in QPnVS2;$$
$$3, \text{ if } LP\_QPn \in QPnVS3,\ n=\{1..5\}$$

In its turn, quality parameter LP_QPn, n={1...5} is calculated according to Formulae 3 – Formulae 7.

Building quality class 'BD_QC' depends on the lowest quality parameter quality class (Formulae 18).

$$\begin{gathered}BD\_QC = MAX(BD\_QPn\_QC),\\ BD\_QPn\_QC = 1, \text{ if } BD\_QPn \in QPnVS1;\ 2, \text{ if } BD\_QPn \in QPnVS2;\\ 3, \text{ if } BD\_QPn \in QPnVS3,\ n=\{1..4\}\end{gathered} \qquad \textbf{(18)}$$

In its turn, quality parameter BD_QPn, n={1...4} is calculated according to Formulae 8 – Formulae 11.

Encumbrance quality class 'EB_QC' depends on the lowest parameter quality class (Formulae 19).

$$\begin{gathered}EB\_QC = MAX(EB\_QPn\_QC),\\ EB\_QPn\_QC = 1, \text{ if } EB\_QPn \in QPnVS1;\ 2, \text{ if } EB\_QPn \in QPnVS2;\\ 3, \text{ if } EB\_QPn \in QPnVS3,\ n=\{1..2\}\end{gathered} \qquad \textbf{(19)}$$

In its turn, quality parameter 'EB_QPn, n={1,2} is calculated according to Formulae 12, Formulae 13.

Part of land parcel quality class 'PLP_QC' depends on the lowest quality parameter quality class (Formulae 20).

$$\begin{gathered}PLP\_QC = MAX\ (PLP\_QPn\_QC),\\ PLP\_QPn\_QC = 1, \text{ if } PLP\_QPn \in QPn\ VS1;\ 2, \text{ if } PLP\_QPn \in QPnVS2;\\ 3, \text{ if } PLP\_QPn \in QPnVS3,\ n=\{1..3\}\end{gathered} \qquad \textbf{(20)}$$

In its turn, quality parameter 'PLP_QPn', n={1...3} is calculated according to Formulae 14 – Formulae 16.

Cadastral map quality class (Fig. 4) depends on the lowest cadastral map object quality class (Formulae 21).

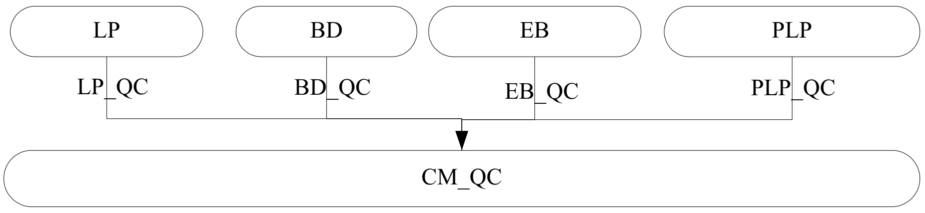


**Fig. 4.** Cadastral Map Quality Class

$$CM\_QC = MAX(\ LP\_QC;\ BD\_QC;\ EB\_QC;\ PLP\_QC) \qquad \textbf{(21)}$$

Now we can evaluate quality of the chosen CM, because we have defined quality parameters (Table 4, Table 7, Table 8, Table 9) and sets of quality parameter values (QPnVSk) (Table 3) and Formulae 2 - Formulae 16, as well as object quality assessment matrix (Table 12) un Formulae 17 - Formulae 21 to calculate quality classes.

## 3.3 Cadastral Map Quality Evaluation Steps

The purpose of cadastral map quality evaluation is to define which quality class a cadastral map belongs to and what purposes the cadastral map cannot be used for. Cadastral map quality depends on the quality of its objects (Fig. 4), that is, on land parcel, building, encumbrance and part of land parcels quality classes (Formulae 21).

To evaluate the quality of the chosen cadastral map, the following steps are made (Fig. 5):

The first step: find out, which objects forms the cadastral map and obtain the binding cadastral map data of the textual part. A cadastral map can be formed by four objects maximum: land parcel, building, encumbrance, and part of land parcel (i={1…4}). The defined method does not depend on cadastral map size – you can choose a cadastral map, which is formed of one land parcel and evaluate it, or choose all possible cadastral maps and evaluate them. Wherewith a cadastral map can be formed by several objects of one type, for example, several land parcels, buildings, etc. Object quality depends on the quality of each object item. Prior to cadastral map evaluation, obtain the number of object items: number of land parcels in the cadastral map 'CM_LP_count', in the textual part 'TP_LP_count', number of buildings in the cadastral map 'CM_BD_count', in the textual part 'TP_BD_count', number of encumbrances in the cadastral map 'CM_EB_count', in the textual part 'TP_EB_count' and number of parts of land parcels in the cadastral map 'CM_PLP_count', in the textual part 'TP_PLP_count' (Fig. 5, P1a, P1b).

The second step: evaluate each object quality by the parameters (Table 4, Table 7, Table 8, Table 9) and obtain object quality class (LP_QC, BD_QC, EB_QC, PLP_QC):

a) check the eligibility of object to quality criterions (Table 3), obtain the number or list of ineligible items QPn_count' or 'QPn_list' (Fig. 5, P2a),

b) obtain object quality class:

- calculate quality parameter QPn values (Formula 3 – Formula 16), obtain QPn (Fig. 5, P2b),
- determine, which quality parameter value set (Table 10) the obtained value belongs to. Obtain QPnVSk (Fig. 5, P2c),
- determine, which is the class of a quality parameter QPn (Table 12), obtain QPn_QC (Fig. 5, P2d),
- determine, which quality class an object corresponds to (Formulae 17 – Formulae 21), obtain LP_QC, BD_QC, EB_QC and PLP_QC. (Fig. 5, P2e).

The third step: determine cadastral map quality (Fig. 4), which depends on the lowest object quality class (Formulae 21), obtain CM_QC (Fig. 5, P3).

For demonstrating the approach for CM evaluation let's choose a map, which contains 19 land parcels, 7 buildings, 2 encumbrances and 1 part of land parcel (Fig. 6). Data in the CM and TP are shown in Table 13- Table 16.

Let's evaluate the quality of land parcels in the chosen CM. We have: five quality parameters for land parcels LP_QPn, n=1…5 (Table 4), three sets of values for quality parameters LP_QPn_VSj, n=1…5, j=1…3 (Table 10) and three land parcel quality classes – high, medium, low LP_QCm, m=1…3 (Table 12), CM and TP data, which are given in Table 13.

Evaluation of a land parcel consists of the following steps:

1st step – acquire the number of land parcels in the chosen CM (Fig. 5): the number of CM land parcels is 19, CM_LP_count=19. Also in the TP the number of land parcels for the chosen region is 19, TP_LP_count=19.

2nd step –

a) calculate how many land parcels do not comply with the proposed criterions, the result is 'LP_QPn_count' or 'LP_QPn_list', n=1...5.

b) then calculate LP_QPn- how many percents it is (Formulae 3 - Formulae 7) and using the sets of values for quality parameters (QPnVSk) and the quality assessment matrix (QAM), acquire quality parameter quality class LP_QPn_QC, n=1...5. Finally, get LP_QC (Formulae 17) (Fig. 3).

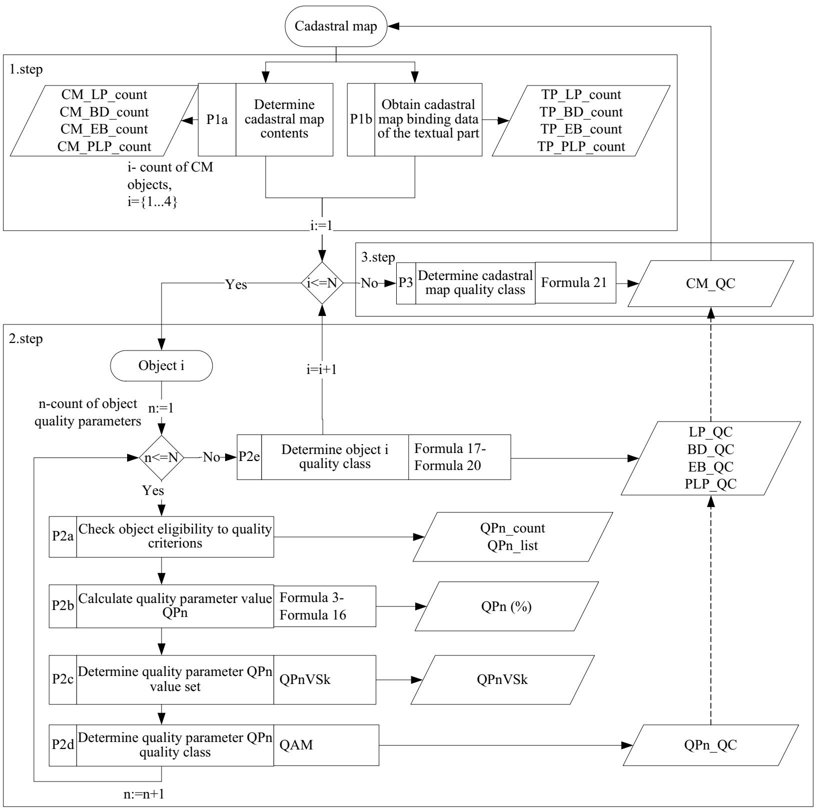

Cadastral map
1.step
CM_LP_count
CM_BD_count
CM_EB_count
CM_PLP_count
P1a
Determine cadastral map contents
P1b
Obtain cadastral map binding data of the textual part
TP_LP_count
TP_BD_count
TP_EB_count
TP_PLP_count
i- count of CM objects, i={1...4}
i:=1
3.step
i<=N
Yes
No
P3
Determine cadastral map quality class
Formula 21
CM_QC
2.step
Object i
i=i+1
n-count of object quality parameters
n:=1
n<=N
No
P2e
Determine object i quality class
Formula 17- Formula 20
LP_QC
BD_QC
EB_QC
PLP_QC
Yes
P2a
Check object eligibility to quality criterions
QPn_count
QPn_list
P2b
Calculate quality parameter value QPn
Formula 3- Formula 16
QPn (%)
P2c
Determine quality parameter QPn value set
QPnVSk
QPnVSk
P2d
Determine quality parameter QPn quality class
QAM
QPn_QC
n:=n+1

**Fig. 5.** Cadastral Map Quality Evaluation

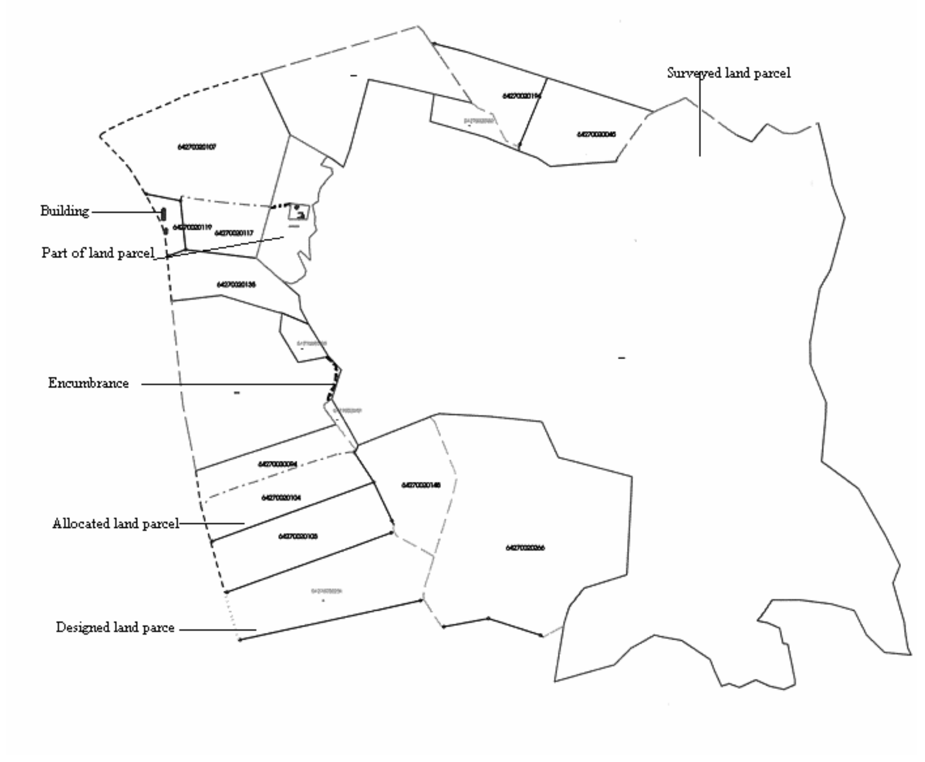


**Fig. 6.** Detail from the *Durbe* Country Cadastral Map

**Table 13.** Land Parcels CM and TP data

| CM | | | | TP | | | |
|---|---|---|---|---|---|---|---|
| Nr | Cadastral number of land parcel | Survey type | Graphical land area m2 | Nr | Cadastral number of land parcel | Survey type | Legal land area m2 |
| 1 | 64270020045 | allocated | 73349 | 1 | 64270020045 | allocated | 82000 |
| 2 | 64270020094 | allocated | 43925 | 2 | 64270020094 | allocated | 51000 |
| 3 | 64270020103 | allocated | 91950 | 3 | 64270020103 | allocated | 91000 |
| 4 | 64270020104 | allocated | 65236 | 4 | 64270020104 | allocated | 59000 |
| 5 | 64270020107 | allocated | 163022 | 5 | 64270020107 | allocated | 158000 |
| 6 | 64270020117 | allocated | 40520 | 6 | 64270020117 | allocated | 38000 |
| 7 | 64270020119 | allocated | 12563 | 7 | 64270020119 | allocated | 15000 |
| 8 | 64270020135 | allocated | 54089 | 8 | 64270020135 | allocated | 64000 |
| 9 | 64270020146 | surveyed | 192035 | 9 | 64270020146 | surveyed | 192100 |
| 10 | 64270020148 | allocated | 81174 | 10 | 64270020148 | allocated | 82000 |
| 11 | 64270020151 | surveyed | 121532 | 11 | 64270020151 | surveyed | 121600 |
| 12 | 64270020189 | designed | 19453 | 12 | 64270020189 | designed | 18000 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 13 | 64270020190 | designed | 12905 | 13 | 64270020190 | designed | 13000 |
| 14 | 64270020191 | designed | 4411 | 14 | 64270020191 | designed | 4000 |
| 15 | 64270020194 | allocated | 49874 | 15 | 64270020194 | allocated | 53000 |
| 16 | 64270020200 | surveyed | 2114825 | 16 | 64270020200 | surveyed | 2115500 |
| 17 | 64270020251 | designed | 119254 | 17 | 64270020251 | designed | 119000 |
| 18 | 64270020266 | allocated | 322332 | 18 | 64270020266 | allocated | 320000 |
| 19 | 64270020317 | surveyed | 2690 | 19 | 64270020317 | surveyed | 2700 |

**Table 14.** Building CM ant TP Data

| CM | | | | TP | | |
|---|---|---|---|---|---|---|
| Nr | Cadastral number of building | Survey type | Cadastral number of land parcel | Nr | Cadastral number of building | Cadastral number of land parcel |
| 1 | 64270020119001 | Vectorized | 64270020119 | 1 | 64270020119001 | 64270020119 |
| 2 | 64270020119002 | Vectorized | 64270020119 | 2 | 64270020119002 | 64270020119 |
| 3 | 64270020119003 | Vectorized | 64270020119 | 3 | 64270020119003 | 64270020119 |
| 4 | 64270020119004 | Vectorized | 64270020119 | 4 | 64270020119004 | 64270020119 |
| 5 | 64270020195001 | Vectorized | 64270020317 | 5 | 64270020195001 | 64270020317 |
| 6 | 64270020195002 | Vectorized | 64270020317 | 6 | 64270020195002 | 64270020317 |
| 7 | 64270020195003 | Vectorized | 64270020317 | 7 | 64270020195003 | 64270020317 |

**Table 15.** Encumbrance CM ant TP Data

| CM | | | TP | | |
|---|---|---|---|---|---|
| Nr | Cadastral number of land parcel | Encumbrance code | Nr | Cadastral number of land parcel | Encumbrance code |
| 1 | 64270020200 | 050301 001 | 1 | 64270020200 | 050301 001 |
| 2 | 64270020146 | 050301 003 | 2 | 64270020146 | 050301 003 |

**Table 16.** Part of Land Parcel CM ant TP Data

| CM | | | TP | | |
|---|---|---|---|---|---|
| Nr | Cadastral number of part of land parcel | Graphical land area m2 | Nr | Cadastral number of part of land parcel | Legal land area m2 |
| 1 | 642700202008001 | 58766 | 1 | 642700202008001 | 55800 |

<u>LP_QP1_QC acquisition</u> (Fig. 7):

a) check, how many land parcels are not in the TP. After the check let us make sure that all land parcels in the CM are also in the TP, therefore LP_QP1_count=0 (Fig. 7, P1, P2),

b) calculate the rate LP_QP1=LP_QP1_count/CM_LP_count*100= 0/19*100=0% (Formulae 3) (Fig. 7, P3). Using the QPVS we see that LP_QP1 value appertains to the set of values LP_QP1_VS1 (Fig. 7, P4) and using the QAM, the value corresponds to the High class LP_QC1, we acquire that LP_QP1_QC=1 (Fig. 7, P5).

<u>LP_QP2_QC acquisition:</u>

a) check, how many land parcels in the TP of the chosen area are not marked in the CM. After the check let us make sure that all land parcels of the TP are marked in the CM, therefore LP_QP2_count=0.

b) calculate the rate: P2_QP2=LP_QP2_count/TP_LP_count*100= 0/19*100=0% (Formulae 4). Using the QPVS we see that LP_QP2 value appertains to the set of values LP_QP2_VS1 and using the QAM, the value corresponds to the High class, LP_QP2_QC=1.

<u>LP_QP3_QC acquisition:</u>

a) check, how many land parcels in the CM have surveying type different from the surveying type in the TP. In the result let us make sure that surveying types in both databases are the same, therefore LP_QP3_count=0.

b) calculate the rate: LP_QP3=LP_QP3_count/CM_LP_count*100= 0/19*100=0% (Formulae 5). Using the QPVS we see that LP_QP3 value appertains to the set of values LP_QP3_VS1 and using the QAM, the value corresponds to the High class, LP_QP3_QC=1.

<u>LP_QP4_QC acquisition:</u>

a) check, how many of surveyed land parcels in the CM have graphical land area larger/smaller than the acceptable difference from legal land is (Table 17):

- calculate the acceptable difference between graphical land area and legal land area (Table 6),
- calculate the actual (fact) difference,
- compare the acceptable difference with the actual area difference. In the result let us make sure that acceptable differences of graphical land area for all land parcels in the CM are within permissible limits, therefore LP_QP4_count=0.

b) calculate the rate: LP_QP4=LP_QP4_count/4*100= 0/4*100=0% (Formulae 6). Using the QPVS we see that LP_QP4 value appertains to the set of values LP_QP4_VS1 and using the QAM, the value corresponds to the High class, LP_QP4_QC=1.

<u>LP_QP5_QC acquisition:</u>

a) calculate, how many land parcels are surveyed in the CM and acquire that LP_QP5_count=4.

b) calculate the rate: LP_QP5= LP_QP5_count/CM_LP_count*100= 4/19*100=21.05% (Formulae 7). Using the QPVS we see that LP_QP5 value appertains to the set of values LP_QP5_VS2 and using the QAM, the value corresponds to the Medium class, LP_QP5_QC=2.

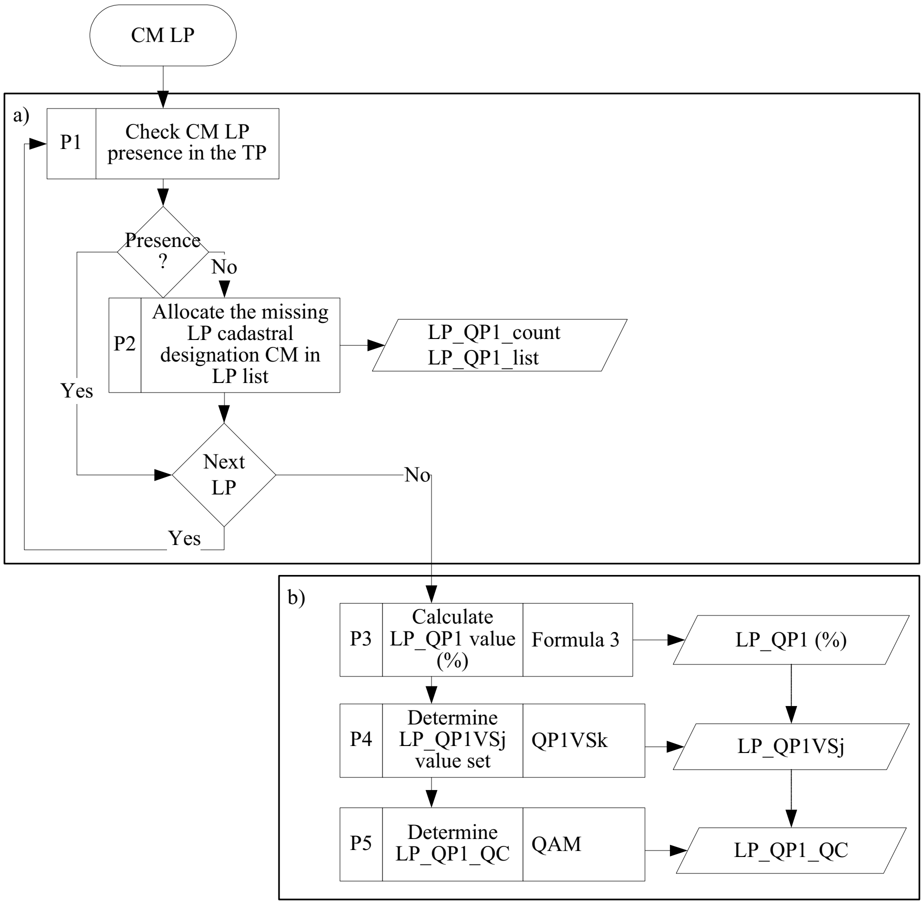

CM LP
a)
P1
Check CM LP presence in the TP
Presence ?
No
Yes
P2
Allocate the missing LP cadastral designation CM in LP list
LP_QP1_count
LP_QP1_list
Next LP
No
Yes
b)
P3
Calculate LP_QP1 value (%)
Formula 3
LP_QP1 (%)
P4
Determine LP_QP1VSj value set
QP1VSk
LP_QP1VSj
P5
Determine LP_QP1_QC
QAM
LP_QP1_QC

**Fig. 7.** Quality Parameter LP_QP1 Quality Class

**Table 17.** Calculation of the Difference between Graphical and Legal Land Area

| Nr. | Cadastral number of land parcel | Graphical land area(a) ha | Legal land area (b) ha | Difference (ha) | | Result |
|---|---|---|---|---|---|---|
| | | | | Acceptable ($\pm x\sqrt{b}$) | Fact ABS(a-b) | |
| 1 | 64270020146 | 19.20 | 19.21 | 1.10 | 0.01 | Acceptable |
| 2 | 64270020151 | 12.15 | 12.16 | 0.87 | 0.01 | Acceptable |
| 3 | 64270020200 | 211.48 | 211.55 | 4.36 | 0.07 | Acceptable |
| 4 | 64270020317 | 0.27 | 0.27 | 0.05 | 0.00 | Acceptable |

Finally, land parcel quality depends on the lowest quality class in every quality parameter: LP_QC=MAX (LP_QPi_QC), i=1…5 (Fig. 3, Formulae 17) and it is Medium class LP_QC=2 - CM (taking into account land parcel quality only), it is permitted to use it for making decisions (Table 11), by ascertaining that land units, which were not surveyed, do not influence the decision. However, if CM usage purpose is not connected with it or land parcels are surveyed (do not take into account LP_QP5, therefore it is not a necessary requirement to be surveyed), then quality of the CM is already High class – LP_QC=1.

If CM usage purpose is connected with involvement of all the objects, it is necessary to evaluate the quality of the other objects. The quality of the other objects is evaluated in a similar way as the quality of land parcels. The quality evaluation of all the objects is given in Table 18.

**Table 18.** Object quality classes

| Land parcel | Building | Encumbrance | Part of land parcel |
|---|---|---|---|
| LP_QP1_QC = 1 | BD_QP1_QC = 1 | EB_QP1_QC = 1 | PLP_QP1_QC = 1 |
| LP_QP2_QC = 1 | BD_QP2_QC = 1 | EB_QP2_QC = 1 | PLP_QP2_QC = 1 |
| LP_QP3_QC = 1 | BD_QP3_QC = 1 | | PLP_QP3_QC = 1 |
| LP_QP4_QC = 1 | BD_QP4_QC = 3 | | |
| LP_QP5_QC = 2 | | | |
| LP_QC = 2 | BD_QC = 3 | EB_QC = 1 | PLP_QC = 1 |

Evaluation for the chosen CM (Fig. 5, 3rd step) is acquired taking into account the lowest quality class of each object: CM_QC= MAX(LP_QC, BD_QC, EB_QC, PLP_QC) (Fig. 4, Formulae 21).

As a result we obtain that quality class of the given CM (taking into account quality of all the objects) is the Low class – CM_QC=3 and it cannot be used for making decisions, it can be used to get primary information.

The evaluation method is based on object usage purpose and 1) if CM usage purpose does not depend on whether a building is surveyed (quality parameter BD_QP4 is not taken into account), then CM quality is of Medium class – CM_QC=2 and it can be used for making decisions, 2) if CM usage purpose does not depend on survey of land parcels and buildings (quality parameters LP_QP5 and BD_QP4 are not taken into account), then quality class is High class - CM_QC=1 and the CM and be used for any purpose.

## 3.4 Cadastral Map Quality Evaluation Software

Cadastral map quality is evaluated according to the defined quality parameters (Table 4, Table 7, Table 8, Table 9) and formulae for calculating their values (formulae 3 – Formulae 16) and used 22 data types (Table 19).

To provide fast and effective data quality evaluation, software for data quality evaluation is developed, which provides:

1) obtainment of data necessary for quality evaluation,
2) quality evaluation according to the defined quality parameters,
3) preparation of data for analysis and quality improvement

As the basis when elaborating data quality evaluation software (DQES) is taken Cadastral Information System Graphical Software (CISGS), which offers the following possibilities (Table 19):

1) make reports (R) on cadastral map and textual part objects,
2) check (C) data quality,
3) search (S),
4) create SQL queries (SQL),
5) save the selected data in MS Excel file.

Functionality of CISGS practically ensures the first step of cadastral map quality evaluation – obtain data about cadastral map content, including data from the textual part (Fig. 5, 1st step). This is provided by CISGS report creating function (report ‘CAD7’). In the report ‘CAD7’ only textual part data about encumbrances are not used, because in the report there are all the textual part encumbrances for the chosen area, but for evaluation only servitudes are necessary (Table 9). That’s why number of textual part encumbrances (servitudes) is obtained by applying an SQL query.

CISGS provides almost all the necessary quality checks (Fig. 5, 2nd step P2a) and selects items ineligible to quality criteria, which can be saved in MS Excel file, but the number of the surveyed objects can be obtained by creating reports ‘CAD1_LP’ and ‘CAD1_BD’. CISGS does not offer two encumbrance quality checks: cadastral map encumbrances not included in the textual part (EB_QP1_list, EB_QP1_count) and textual part encumbrances not included in the cadastral map (EB_QP2_list, EB_QP2_count).

**Table 19.** CISGS Data

| Nr | Data type | Quality parameter | CISGS | Name of data type | Formula |
|---|---|---|---|---|---|
| 1. | Number of LP in CM | LP_QP1, LP_QP3-LP_QP5 | CAD7 (R) | CM_LP_count | 3,5-7 |
| 2. | Number of LP in TP | LP_QP2 | CAD7 (R) | TP_LP_count | 4 |
| 3. | Number of BD in CM | BD_QP1, BD_QP, BD_QP4 | CAD7 (R) | CM_BD_count | 8, 10, 11 |
| 4. | Number of BD in TP | BD_QP2 | CAD7 (R) | TP_BD_count | 9 |
| 5. | Number of EB in CM | EB_QP1 | CAD7 (R) | CM_EB_count | 12 |
| 6. | Number of EB in TP | EB_QP2 | TP_EB_list (SQL) | TD_EB_count | 13 |
| 7. | Number of PLP in CM | PLP_QP1, PLP_QP3 | CAD7 (R) | CM_PLP_count | 14,16 |
| 8. | Number of PLP in TP | PLP_QP2 | CAD7 (R) | TD_PLP_count | 15 |
| 9. | List of cadastral map land parcels, which are not in the textual part | LP_QP1 | LP_QP1_list (C) | LP_QP1_list LP_QP1_count | 3 |
| 10. | List of land parcels in the textual part, which are not in the cadastral map | LP_QP2 | LP_QP2_list (C) | LP_QP2_list LP_QP2_count | 4 |
| 11. | List of cadastral map land parcels with different survey type | LP_QP3 | LP_QP3_list (C) | LP_QP3_list LP_QP3_count | 5 |
| 12. | List of cadastral map land parcels with different area | LP_QP4 | LP_QP4_list (C) | LP_QP4_list LP_QP4_count | 6 |
| 13. | Number of surveyed cadastral map land parcels | LP_QP5 | CAD1_LP | LP_QP5_count | 7 |

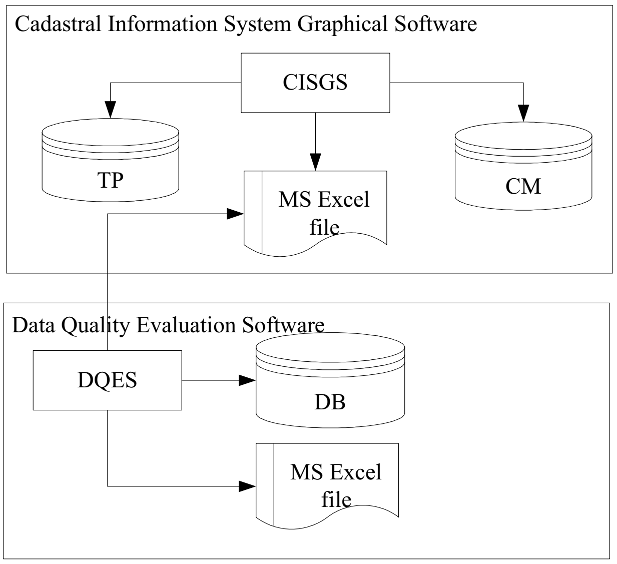
| Nr | Data type | Quality parameter | CISGS | Name of data type | Formula |
|---|---|---|---|---|---|
| 14. | List of cadastral map buildings not included in the textual part | BD_QP1 | BD_QP1_list (C) | BD_QP1_list BD_QP1_count | 8 |
| 15. | List of textual part buildings not included in the cadastral map | BD_QP2 | BD_QP2_list (C) | BD_QP2_list BD_QP2_count | 9 |
| 16. | List of cadastral map buildings, which land parcel designations do not match with the designations of land parcels in the textual part, to which it is attached | BD_QP3 | BD_QP3_list (C) | BD_QP3_list BD_QP3_count | 10 |
| 17. | Number of surveyed cadastral map buildings | BD_QP4 | CAD1_BD (R) | BD_QP4_count | 11 |
| 18. | List of cadastral map encumbrances not included in the textual part | EB_QP1 | CM_EB_list (S) | EB_QP1_list EB_QP1_count | 12 |
| 19. | List of textual part encumbrances not included in the cadastral map | EB_QP1 | TP_EB_list (S) | EB_QP2_list EB_QP2_count | 13 |
| 20. | List of parts of land parcels in the cadastral map not included in the textual part | PLP_QP1 | PLP_QP1_list (C) | PLP_QP1_list PLP_QP1_count | 14 |
| 21. | List of parts of land parcels in the textual part not included in the cadastral map | PLP_QP2 | PLP_QP2_list (C) | PLP_QP2_list PLP_QP2_count | 15 |
| 22. | List of parts of land parcels in the cadastral map with different area | PLP_QP3 | PLP_QP3_list (C) | PLP_QP3_list PLP_QP3_count | 16 |

R- report, C- check , S - search functionality of CISGS

Although CISGS provides the data necessary for quality evaluation and performs almost all quality checks, the software does not provide data quality evaluation.

According to the present situation, obtain, that DQES tasks are (Fig. 8):

1) to import data to DQES data base,
2) to make encumbrance data quality checks,
3) to evaluate cadastral map quality according to quality parameters and quality evaluation matrixes (Fig. 5, 2nd step P2b. – P2e, 3rd step) and to display them in MS Excel file,
4) to prepare data in MS Excel file for analysis and improvement of quality.



Cadastral Information System Graphical Software
CISGS
TP
MS Excel file
CM
Data Quality Evaluation Software
DQES
DB
MS Excel file

**Fig. 8.** CISGS and DQES

### 3.4.1 Data Import and Quality checks

CISGS offers to save data selected in reports, checks, searches and SQL queries in MS Excel file. For data import, using DQES, strictly keep to the definite folder structure (Fig. 9): 1) As cadastral map quality evaluation data (Table 19) are taken on the same date, they are stored in a folder with a name: YYYYMMDD, 2) for each cadastral map object – LP, BD, EB, PLP and the report 'CAD7' a folder is created, into which MS Excel files created with CISGS are placed. The number of files depends on the size of the chosen area and data errors. Work with a cadastral map in SLS is organised in regional departments and department offices. Within offices cadastral maps are created for cadastral areas and cadastral groups. This principle for work with cadastral maps is introduced into CISGS and the data necessary for evaluation are obtained through cadastral areas. For example, information about the *South Kurzeme* regional department can be obtained from CISGS from up to 79 MS Excel files: LP -22, BD -15, EB -24, LPL -15 and CAD7-3.

Prior to data import DQES processes encumbrance data: replacing the number of encumbrances in the textual part in 'CAD7' report with the number of encumbrances in the textual part, which is necessary for evaluation, as well as encumbrance checks. Thus, in the data base are stored only the data, which are necessary for evaluation process. DQES DB imported data are stored in 15 tables respectively.

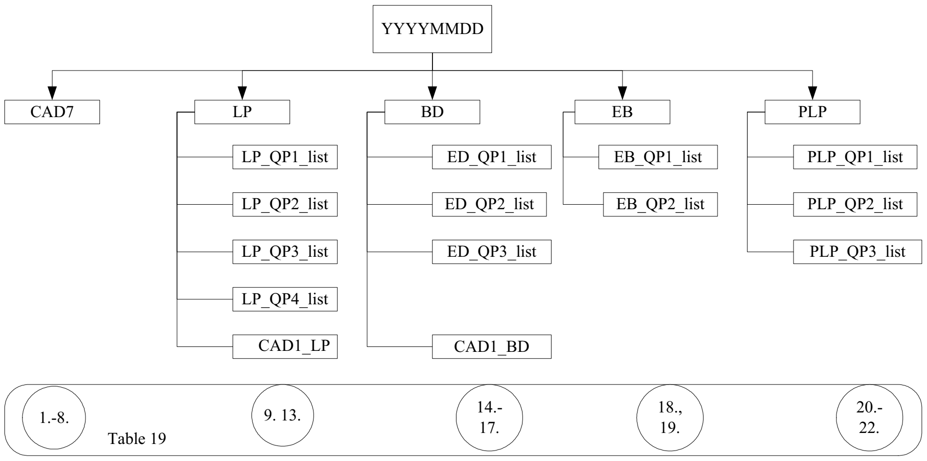


**Fig. 9.** Folder Structure for Data Import

### 3.4.2 Data Quality Evaluation and Mapping of the Results

DQES provides cadastral map quality evaluation according to the definite quality parameters (Table 4, Table 7, Table 8, Table 9), formulas (Formulae 3 – Formulae 16), determines object quality classes according to quality matrix (Table 12) and formulas (Formulae 17 – Formulae 21).

The obtained evaluation results DQES maps in MS Excel file, in the worksheet for each cadastral map object (Table 20), which can contain various cadastral areas,

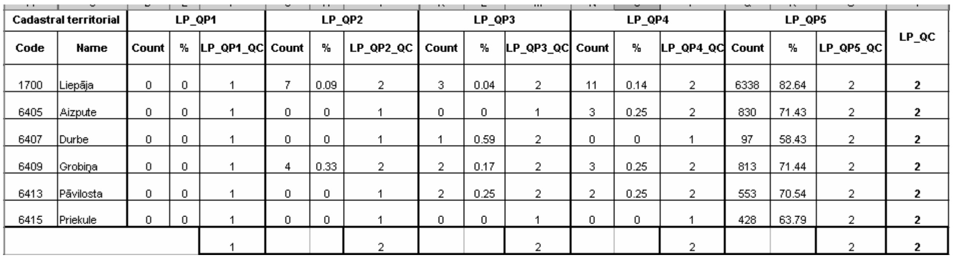
- horizontally (as object item) displayed:
  1-cadastral area code and title,
  2-for each quality parameter (QPn) three data types are displayed: number of items eligible/ineligible to criteria (Count), according to formulae calculated percents of parameter value (%) and the class corresponding to the obtained value (QC),
  3-cadastral area quality class (QC) that depends on each quality parameter quality class,
- vertically (as quality parameter QPn) displayed:
  4-each quality parameter quality class (QPn QC)
  5-object quality class (QC)

**Table 20.** Structure of Evaluation Data Mapping

| Cadastral map territorial code and name | Quality parameter QP1 | | | .. | Quality parameter QPN | | | Quality class |
|---|---|---|---|---|---|---|---|---|
| 1 | 2’ | | | 2’’ | 2’’’ | | | 3 |
| | Count | % | QC | ... | Count | % | QC | Cad. ter. QC. |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| | | | 4’ QP1 QC | 4’’ | | | 4’’’ QPN QC | 5 Object QC |

The maximum a file can contain is 4 worksheets with quality evaluation data – a separate worksheet for each object (LP, BD, EB, PLP).

As an example for land parcel evaluation data mapping (Fig. 10) are given data from *South Kurzeme* regional department in *Liepāja* office for towns cadastral territorial land parcel quality evaluation on August 9, 2007.



| Cadastral territorial | | LP_QP1 | | | LP_QP2 | | | LP_QP3 | | | LP_QP4 | | | LP_QP5 | | | LP_QC |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Code | Name | Count | % | LP_QP1_QC | Count | % | LP_QP2_QC | Count | % | LP_QP3_QC | Count | % | LP_QP4_QC | Count | % | LP_QP5_QC | |
| 1700 | Liepāja | 0 | 0 | 1 | 7 | 0.09 | 2 | 3 | 0.04 | 2 | 11 | 0.14 | 2 | 6338 | 82.64 | 2 | **2** |
| 6405 | Aizpute | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 3 | 0.25 | 2 | 830 | 71.43 | 2 | **2** |
| 6407 | Durbe | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0.59 | 2 | 0 | 0 | 1 | 97 | 58.43 | 2 | **2** |
| 6409 | Grobiņa | 0 | 0 | 1 | 4 | 0.33 | 2 | 2 | 0.17 | 2 | 3 | 0.25 | 2 | 813 | 71.44 | 2 | **2** |
| 6413 | Pāvilosta | 0 | 0 | 1 | 0 | 0 | 1 | 2 | 0.25 | 2 | 2 | 0.25 | 2 | 553 | 70.54 | 2 | **2** |
| 6415 | Priekule | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 428 | 63.79 | 2 | **2** |
| | | | | 1 | | | 2 | | | 2 | | | 2 | | | 2 | **2** |

**Fig. 10**. *Liepāja* Office for Towns Land Parcel Quality Evaluation

## 3.4.3 Data Preparation for Analysis

Quality evaluation data for analysis are displayed in charts. DQES prepares two types of charts: charts describing charts for cadastral map object quality parameter values (%) and charts describing cadastral map and its object quality classes.

For object quality parameters (Table 4, Table 7, Table 8, Table 9) can be 7 charts maximum: PLP – 1, EB – 1, but LP and BD have 2 charts each, because quality parameter LP_QP5 and BD_QP4 value sets (Table 10) are different from other quality parameter value sets, as well as the chart, in which are given quality parameter values, which describe objects – LP_QP1, LP_QP2, BD_QP1, BD_QP2, EB_QP1, EB_QP2, PLP_QP1, PLP_QP2.

As an example of land parcel analysis data mapping are given the data of *South Kurzeme* regional department *Liepāja* office for towns cadastral area land parcel evaluation on August 9, 2007 by parameters LP_QP1, LP_QP2, LP_QP3, LP_QP4 (Fig. 11) and LP_QP5 (Fig. 12).

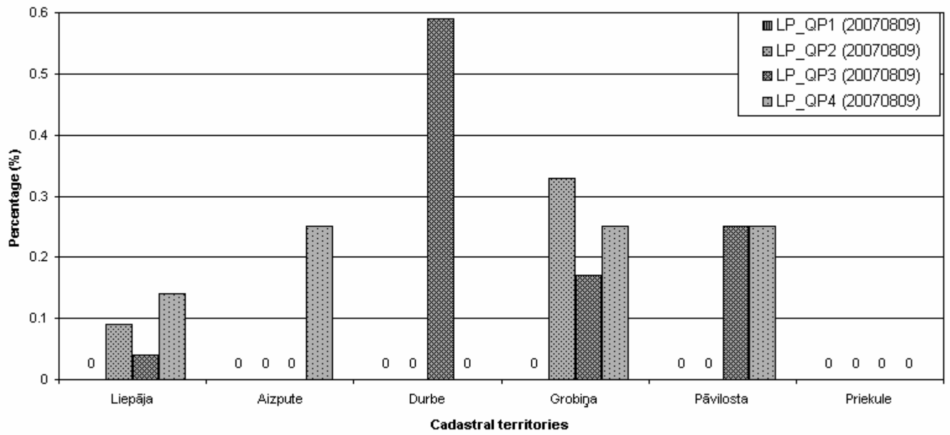


**Fig. 11.** *Liepāja* Office Land Parcel Analysis Data

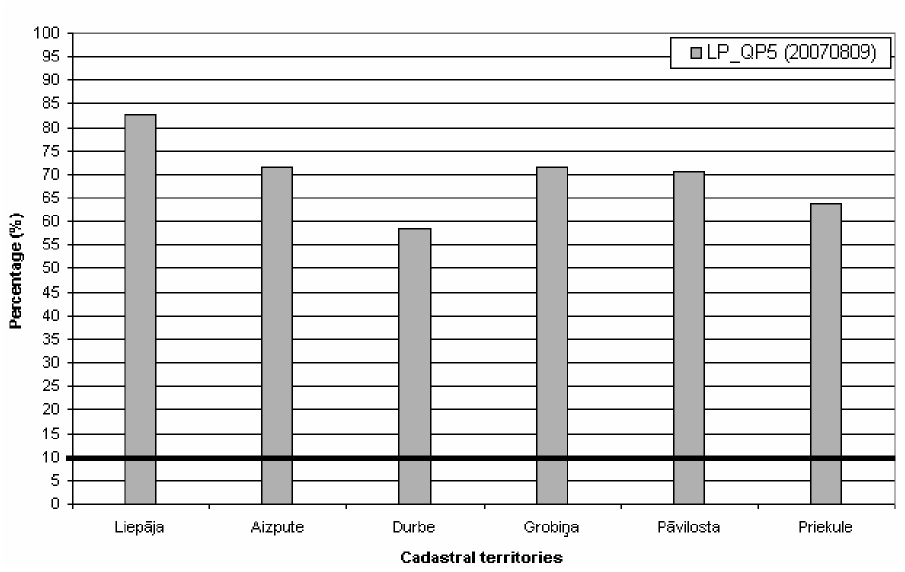


**Fig. 12.** *Liepāja* Office Land Parcel Analysis Data

For quality parameter quality classes can be 5 charts maximum: each cadastral map object quality parameter quality classes and cadastral map quality classes.

As an example of land parcel quality parameter quality class analysis data mapping are given the data of *South Kurzeme* regional department *Liepāja* office for towns cadastral area land parcel evaluation on August 9, 2007 by parameters LP_QP1, LP_QP2, LP_QP3, LP_QP4 and LP_QP5 quality class (Fig. 12).

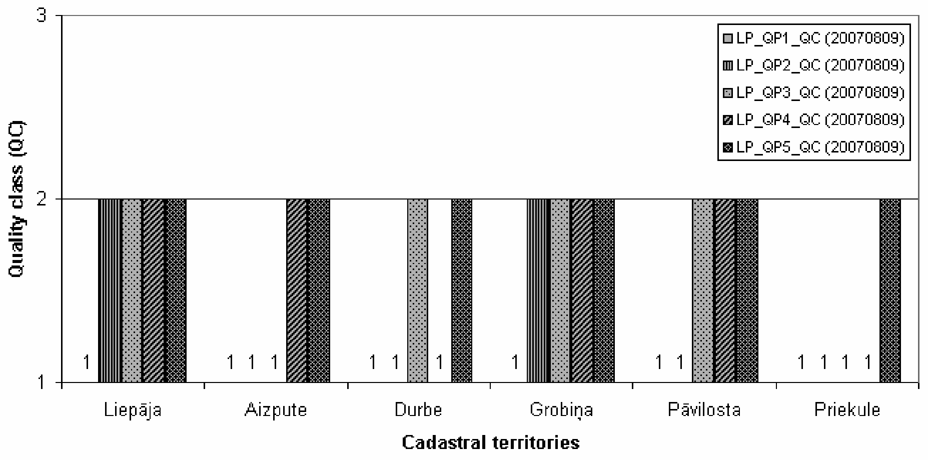


**Fig. 13.** *Liepāja* Office Land Parcel Quality Parameter Quality Classes

As an example for cadastral map object quality class analysis data mapping are given the data of *South Kurzeme* regional department *Liepāja* office for towns cadastral area land parcel evaluation on August 9, 2007 (Fig. 14).

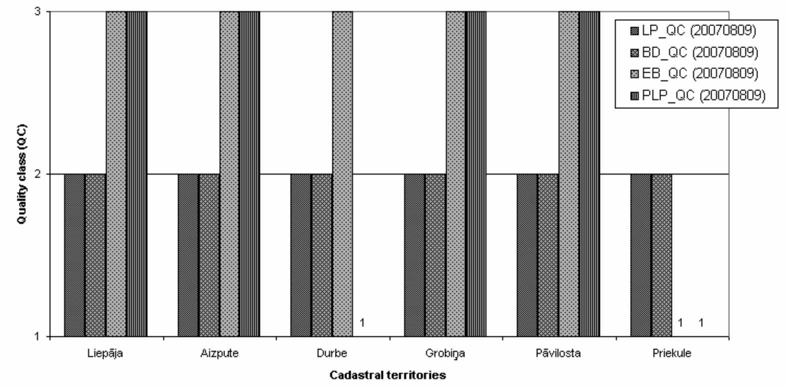


**Fig. 14.** *Liepāja* Office Cadastral Object Quality Classes

### 3.4.4 Storage of Quality Evaluation Data

DQES maps quality evaluation data and analysis data in MS Excel file. The file, if it is necessary, can be stored in the chosen location and name.

In general MS Excel file created by DQES can contain 16 worksheets maximum: 4 worksheets with quality evaluation data, 7 worksheets with charts for object quality parameter values and 5 worksheets with charts for quality classes (object quality parameter and for the cadastral map.

For development of DQES such tools are chosen: My SQL, MS Visual Basic and MS Excel. DQES DB contains 30 tables (15 data tables, 6 classifiers, 4 data quality evaluation result tables and 5 support tables), the interface consists of 5 display forms, but program code contains approximately 4300 rows. Software specification and design are made by A.Jansone, but code is made by K.Grietēns.

The defined method does not depend on the size of each cadastral map – you can choose a cadastral map with one land parcel with existing objects- and evaluate it, as well as you can choose all the cadastral maps in the data base and evaluate them.

## 4 Conclusion

The described approach can be applied to any CM. Quality assessments can be obtained not only for CM of small territories but also for big areas, e.g., cities, regions. The example given in this paper is an assessment of a portion of the Latvian country *Durbe* and reveals where the weaknesses of the map may be.

The insights gained from this analysis are varied. For example, lists of land parcels for which data quality is poor and where data quality needs to be improved in order to be useful for given purposes. In particular, approximate calculations can be done to estimate the time and financial commitment required to bring a CM to a desired quality; for example, to carry out border adjustments in particular territories.

The elaborated method can be used for quality evaluation of objects of any type and the main steps of the method are: firstly, from experiments obtain subjective opinion about object quality descriptive parameters - which value depends on object usage purpose. Secondly, perform structuring of expert subjective opinion and define object quality parameters and their values, according to object binding normative documents and existing IT solutions in the company. Thirdly, together with experiments define object quality classes depending on object usage purposes and what quality parameter values create each quality class, consequently, obtain object quality evaluation matrix, which is used to evaluate the use of an object for the chosen purpose.

This paper presents an object corresponding to the lowest quality parameter quality class - “hard” principle for object evaluation. Other principles (for example, “soft” principle) are going to describe in coming research papers.

In order to make everyday use of a cadastral map easy and simple, support software (Data Quality Evaluation Software) is elaborated for calculating values of quality parameters and for quality class determination, as well as for obtaining charts to analyse data and to elaborate a plan for improving data quality. If without DQES data quality evaluation of one regional unit (i.e. *South Kurzeme* regional unit) required 2-3 days, now the needed time is 1-2 hours. DQES data quality evaluation algorithms tested in practice can be used for supplementing CISGS.

Continuing research is aimed at identifying more quality parameters and ensuring that extracted quality parameters conform to the initial subjective opinions of experts.

## 5 Acknowledgment

The authors would like to thank the SLS experts A.Sideļska, I.Rudzīte, I.Pauliņa, and all Regional Offices experts who took part in the interviews for their assistance in giving freely of their time and expertise with CM, as well as K.Grietēns, master student of Liepāja Academy of Pedagogy, for coding DQES.

The research was partly funded by the European Social Fund and the Latvian Science Council.

## References

1 Olson, J. E.: Data Quality: The Accuracy Dimension, pp 24-27. Morgan Kaufmann Publisher (2003)
2 Batini, C., Scannapieco, M.: Data Quality: Concepts: Methodologies and Techniques, pp 19-49. Springer (2006)
3 Eppler, M. J., Helfert, M., Pernici, B.: Preface. In: 16th Conference on Advanced Information Systems Engineering (CAiSE’04), DIQ’04 Workshop Chairs, pp 3-4. Rīga (2004)
4 NCGIA Core Curriculum in Geographic Information Science, http://www.ncgia.ucsb.edu/giscc/units/u100/u100_f.html
5 Redman,T.,C.: Data Quality: The Field Guide, pp 223, Digital Press, (2001),
6 Borzovs, J., Jansone, A.: An Approach to Geographical Data Quality Evaluation. In 7th International Baltic Conference Databases and Information Systems, pp 125 – 131, Vilnius (2006)
7 Cadastral Template a Worldwide Comparison of Cadastral Systems, http://www.geo21.ch/cadastraltemplate/countrydata/lv.htm

# METHODOLOGY

LATVIJAS UNIVERSITĀTES RAKSTI. 2008, 733. sēj.:
DATORZINĀTNE UN INFORMĀCIJAS TEHNOLOĢIJAS 291.–302. lpp.

# An Outstanding Example of University-Industry Partnership: the Latvian Case[1]

Juris Borzovs
University of Latvia, Riga Information Technology Institute, Exigen Services DATI
Raiņa bulvāris 19, LV-1586 Rīga, Latvija
juris.borzovs@lu.lv

**Abstract.** The Latvian ICT sector is unquestionably the leader in co-operation with educators. In the year 2000, leading Latvian professional ICT associations established the Council of Professional Education. The Council on behalf of employers and industry has coordinated the development of requirements for needed professionals, has established requirements for qualification exams, and approves membership of qualification commissions. Based on these requirements and on ACM/IEEE Computing Curricula, the University of Latvia performs an innovative computing study programme that organically comprises academic and professional education, and covers all the five ACM/IEEE Computing Curricula disciplines: computer science, software engineering, information systems, information technology, and computer engineering.

**Keywords:** computing, study programme, university-industry partnership.

## 1 Introduction

Relations between academy and industry are nearly always somewhat contradictory. The former relies on long-term research and education process, while the latter needs immediate solutions and narrowly, however, deeply educated employees. Overcoming the gap between the two parties is by no means easy.

The ICT sector is unquestionably the leader in co-operation with educators in Latvia. In the year 2000, leading Latvian professional ICT associations established the Council of Professional Education. The Council on behalf of employers and industry has coordinated the development of requirements for needed professionals, has established requirements for qualification exams, and approves membership of qualification commissions. Based on these requirements and on ACM/IEEE Computing Curricula [1], the University of Latvia performs an innovative computing study programme that organically comprises academic and professional education, and covers all the five ACM/IEEE Computing Curricula disciplines: computer science, software engineering, information systems, information technology, and computer engineering.

[1] This paper originally was presented at the 2nd IT STAR Workshop on Universities & the ICT Industry (UNICTRY '07), Genzano di Roma, 26 May 2007.

Levels of ICT education are described in section 2. Current situation in ICT sector is covered in section 3. Latvian professional higher education system is explained in section 4. The unique employer' s ability to influence Latvian universities' study programmes and their presentation is given in section 5. The University of Latvia computing study programme as an innovative approach to education is presented in section 6. Bologna process movement in the right direction is questioned in section 7.

## 2 Levels of ICT Education

The Organisation for Economic Co-operation and Development (OECD) has recommended that skills related to information and communications technologies (ICT) be classified into three major categories:
Professional IT skills: The ability to use complex IT tools and/or to design, repair or create such tools;
Applied IT skills: The ability to use simple IT tools in general places of employment (not ones related to IT);
Basic IT skills: The ability to use IT tools for simple tasks and as an educational tool.

At this point it is worth explaining that the concept of ICT is interpreted in Europe as referring to information technologies, telecommunications or electronic communications, and electronics. In other words, it regards a sector of economy in which various kinds of electronic equipment are manufactured, including computers and electronic communications equipment, communications networks are established, software and information systems are designed, and relevant services are provided. Among these, of course, the most common ones are telephone and Internet services. The concept of "IT", however, has several meanings. In the narrowest sense, it refers to the design of computer software or information systems, while the concept of IT skills refers to the ability to use this software. However, in a broader sense, IT and ICT are synonyms.

ICT is a special sector because its products and services are used by almost everyone and everywhere. In this sense it reminds one of the infrastructure sectors-transportation. It is no accident that until 2003, the ICT sector in Latvia was governed by the Ministry of Transport. Not everyone has to know how to build motor vehicles and roads, but people do have to study before they can drive a motor vehicle. Not everyone must know how to design ICT tools, but there is usually a need to study them before use

There are 13 institutions of higher education in Latvia where one can pursue a degree in IT – in Rīga, Daugavpils, Liepāja, Jelgava, Rēzekne, Ventspils, Valmiera, Jūrmala and Jēkabpils. There are also some 10 professional IT high schools. 80% of those who receive a degree in ICT come from the Riga Technical University, the University of Latvia and the Transport and Communications Institute.

In Latvia applied IT skills can be learned through a programme known as the European Computer Driving Licence (ECDL). The programme was introduced in Latvia in 2001 by the Riga Information Technology Institute (RITI). The programme licence holder is the Latvian Association of Information and Communications Technologies (LIKTA). Currently the programme is being implemented by the University of Latvia, but certification exams can also be taken in Latvia's regional centres. It must be emphasised that a certificate issued in Latvia is valid in more than 40 countries, including all the European Union member states .

Latvia is the first country in the world to introduce the ECDL programme in the general national education programme. Since the autumn of 2003, basic IT skills have been taught at the elementary school level, while applied skills to satisfy all ECDL requirements are taught at high schools. There are several companies that offer training in this area on a commercial basis. Still, we are at the beginning of the road, because only about 2,000 certificates have been issued so far. In nearby Sweden, by comparison, nearly one million certificates have been issued.

## 3 What Is Happening in the ICT Sector?

The days when anyone who knew how to switch on a computer could hope for a salary of USD 100,000 per year in America are long gone, and they are irretrievable. Neither is it true any longer that naïve investors are in a hurry to invest all their money in any company that has anything to do with information technologies. Economic stagnation in America, Germany and many other "engines of the global economy" in the early part of the 21st century made people think about every dollar and euro before it was spent. Research and development budgets were the first to be cut, and there was also less spending on the development of information systems and on outsourcing. These, however, are the three major pillars of the entire ICT sector. The economies have recovered, and the pillars are back in place, but there are far fewer pointless investments and thoughtless spending projects in the area of information systems. National economies require experts with in-depth knowledge, skills and experience in the area of ICT and and in its relevant areas. "Soft" skills will also be of key importance – the ability to read, speak and write in several languages, dedication, responsibility, the ability to manage others, etc. It will be very hard to find job without the aforementioned "soft" skills and professional experience. University students need to think about professional and "soft" skills while they are still at school, and they should accumulate as much professional experience as possible.

We understand increasingly that there are two possible routes in the ICT profession – the "deep" and the "broad" route. In the first case, the professional has very detailed technical skills and knowledge in a fairly narrow sector in which he or she will always be able to find job – although not always in Latvia. In the second case, the knowledge and skills will be broader, but the professional will not always have sufficiently detailed or precise skills for a specific job. Compensation for this will be provided by a wealth of "soft" skills. As one person humorously put it, those who know how will always have work, and those who know why will always be their bosses.

There are several places on the Internet [2, 3] where one can learn about the specific knowledge, skills and properties that are needed in the ICT sector. The first of these was established by a consortium of prominent European ICT companies so as to encourage universities to adapt their ICT study programmes to the demands of the labour market to a greater extent. The second site was established by the Professional Education Administration of the Latvian Ministry for Education and Science. The PEA is the institution which maintains professional standards. These standards are defined by the state so that employers can inform educators about the kinds of workers and qualifications that are required No professional education programme may be launched in Latvia before the relevant professional standard has been implemented. ICT is the only sector in the economy which, thanks to the sector's Professional Education Council, has drafted all the necessary standards (Table 1).

**Table 1.** The standards of the ICT profession

| | **IT** | **Telecommunications** | **Electronics** |
|---|---|---|---|
| 5th-level qualification | IT project manager; Systems analyst; Software engineer | Telecommunications engineer | Electronics engineer |
| 4th-level qualification | Software developer; Tester; Computer network administrator | Telecommunications specialist | Electronics specialist |
| 3rd-level qualification | Software technician; Computer systems technician | Telecommunications technician | Electronics technician |

It is worth looking at the level of education among European ICT specialists. Research shows that the proportions vary from one EU member state to another, but on average 50 to 70% of ICT specialists hold at least a bachelor's degree, while 30 to 50% have the so-called sub-degree education. In Latvia, this applies to people who have pursued their education at a college or a professional secondary education institution. There is a view that in quantitative terms, demand for ICT specialists is currently satisfied in Europe, but there is a need for a greater proportion of specialists with a college diploma or bachelor's degree. The education structure in Latvia's ICT world is dominated by bachelor's degree programmes, and there are more master's degree students than there are college students. This structure would be considered mistaken in Europe, but in Latvia it may be quite commendable. The labour market for our ICT specialists cannot be limited to Latvia alone. Specialists will be able to compete abroad only if they have higher education – a master's or doctoral degree.

Latvia is a very small country, and its ICT market will not be worth more than EUR 60 million per year in the foreseeable future. This means that no more than 2,000 specialists will find work in the near future. Others will have to find jobs in state, local government and other organizations, taking care of their information systems. Perhaps several thousand specialists can find work in these areas. This is not good news, given that each year more than 1,000 specialists graduate from higher education institutions. The good news, however, is that since May 2004, the ICT market has become 200 times larger, because Latvia is now a member state of the European Union. Of course, without preparedness and only with Latvian and Russian language skills no one is going to find a job outside Latvia, but another bit of good news is that the proportion of small companies in Latvia will triple and draw closer to the European level. This will mean a major increase in the use of information technologies at such companies. If one ICT specialist can provide services to 10 small companies, that will mean a need for approximately 8,000 specialists .

## 4 Professional Higher Education

Over the last 15 years, massive changes have taken place in the Latvian economic system, and one of the negative side effects to this was the breakdown of connection between the economy and the education system. New study programmes were often based on the capacities of education institutions, not on the requirements of the labour market. There was no one to formulate that demand in any event. During the previous era, internship in industrial setting was an integral part of study programmes, but that was no longer true. Employers regularly complained about the fact that the knowledge and skills of graduates were not in line with modern requirements, particularly in the field of information technologies and other areas of engineering. There was concern about the fact that Latvia, in comparison to the "old" countries of Europe, had proportionally low numbers of students at the so-called post-secondary non-tertiary level of education (the phase between secondary and higher education, as defined in UNESCO ISCED-97 [4], the 4th level, and in Latvia's case, approximately at the level of former "tehnikums"). This suggests that there were some areas in which graduates suitable for the market could be trained in a shorter period of time – just a few years after graduation from high school.

Here it should be explained that the issue back then was not the present-day trend of insisting that everyone pursue a three-year bachelor's degree. Instead, the aim was to ensure that high school graduates pursued areas of specialisation that were in demand in the labour market. Later this process unofficially became known as college education – the first level of a professional higher education. Officials at various universities were afraid of the competition and the possibility that their students might be tempted to attend colleges instead. Making use of competition among the various departments of the Ministry of Education and Science, they achieved the inclusion of colleges in the system of higher education. This was completely in opposition to the initial goals of the reforms, as well as to the intentions of the minister of that time, Jānis Gaigals. The numerical disproportion between students pursuing higher education, professional secondary education and post-secondary non-tertiary education has expanded, not shrunk. Latvia has become a country in which there is basically no opportunity to pursue education that corresponds to the aforementioned 4th level of the UNESCO ISCED-97.

We know this now, but we could not know it in 1999, when the author of this article invited several leading specialists from IT companies to visit the Riga Information Technology Institute (RITI, a research institute belonging to the stock company Exigen Services DATI) so as to draft requirements related to the professional qualifications of specialists in the area of software development and design of information systems.[2] First to respond were Valdis Lauks from Fortech, Ivo Odītis from the Bank of Latvia, Jānis Plūme from IT Alise, and Uldis Sukovskis from RITI. Employers wanted to dissociate from fruitless criticisms of the education system and its universities and to become involved instead in the restructuring of study programmes in a practical way. There was a certain degree of serendipity. Shortly after the working group was assembled, I was sought out by Aleksandra Joma, a project director for the Professional Education Development Programme. She was looking for people who could handle a PHARE-financed programme, “Professional Education 2000”. The working group immediately became involved in what proved to be an enormously successful and sustainable project, “Establishing a Structure of Professional Qualifications”. Our aim was to study the condition of the information technology sector (the construction industry was also studied), to consider the professions that are needed therein, to select one or two most highly demanded areas (we sensed that this could be in line with the so-called fourth professional qualifications level in the understanding of the law on professional education, with employees of this kind trained by the intended colleges), to draft descriptions or standards for the professions and to prepare sample study programmes. We hoped that the methodology that we were designing and testing would serve as an example for similar standards in other professions and sectors.

2 This was not the only Exigen Services DATI and RITI initiative in the area of education and research. One can cite, for instance, the co-operation between RITI and DATI with the University of Latvia Institute of Mathematics and Computer Science on the design of the well-known GRADE system. The RITI introduced the European Computer Driving License programme in Latvia in 2001. It served as a cornerstone for research and development in the IT industry, including the writing of doctoral dissertations. DATI guaranteed Hansabanka loans for students before the national government began to do so. We have worked with the Latvian Education Fund and its “For Education, Science and Culture” programme to award scholarships to doctoral students and prizes to the authors of the best master’s degree, bachelor’s degree and engineering papers. DATI holds conferences for computer science students each year. The Latvian Academy of Sciences works with DATI and the Latvian Education Fund to award the Eižens Āriņš prize. There is ongoing support for informatics olympiads for schoolchildren at the Latvian, Baltic, and global level. RITI is also the “seat” for the Sub-Commission on Information Technologies and Telecommunications of the Terminology Commission of the Latvian Academy of Sciences, and of the Council on Professional Education in the Fields of Information Technologies, Telecommunications and Electronics.

By September 2000, we had prepared the description for the information technology, telecommunications and electronics (now known as the ICT) sector [5], as well as a professional standard for the category "Software Developer" [6]. We had also prepared a sample programme of study. The work was done by a group of educators under the leadership of Professor Jānis Grundspeņķis of the Riga Technical University. All that remained was a seemingly petty issue – ensuring that project director Aleksandra Joma would not be concerned about whether the ICT sector would declare the results to be good and about who would be able to do so in the name of the entire sector. At that time the author was the president of the Latvian Information Technology and Telecommunications Association (then the LITTA, now the LIKTA), and he had involved people from three other professional organisations in the sector in the work that was done. These were the Latvian Association of the Electronics and Electronic Technologies Industry (LETERA), the Latvian Telecommunications Association (LTA), and the Latvian Computer Technology Association (LDTA). It was not therefore difficult to reach agreement with the fellow presidents (the well known Inārs Kļaviņš, Pēteris Šmidre, and Dzintars Zariņš) on how to evaluate and approve the project results. This gave the green light for the Sub-Council on Tripartite Professional Education and Employment Co-operation (PINTSA) to give its approval as well. In January 2001, as a result of this, the Minister for Education and Science officially confirmed Latvia's first professional standard on behalf of the government. The four association presidents also reached agreement on the establishment of a Professional Education Council for the sector. It was entrusted with representing the sector in the area of education – coordinating and confirming professional standards, coordinating and confirming requirements for examination of qualifications, and confirming experts who would represent employers when those examinations occurred. It is easier to walk down a beaten path, and so it was far easier to draft the following standards after the first one was in place. Today there are 14 standards which have been implemented with the direct or coordinating participation of the Professional Education Council. The ICT sector is unquestionably the leader in co-operation with educators. Since 2006, the Professional Education Council has been chaired by the vice president of Exigen Services DATI, Uldis Smilts.

## 5 The Employer's Ability to Influence Study Programmes and Their Presentation

As was mentioned in the previous section, employers, via the offices of the Professional Education Council established by professional associations, participate in the preparation of standards in the profession, but that is not the only way how they participate. According to government rules, at least a half of members of examination commissions must represent employers, and that includes the chairperson of the commission. Membership of commissions is approved by the Professional Education Council.

The same commissions also award professional qualifications. To make sure that the work of the commissions is not arbitrary, the Professional Education Council approves qualification requirements that are afterwards examined by members of the commission. Educators will prepare students to satisfy the qualification requirements.

Another way to influence the study process is to establish study programme councils at universities on the principle of parity between educators, students, and employers. Each major change to the curriculum must be first discussed and confirmed by this council. That does not mean that university Senates will automatically approve changes, but there have to be very fundamental arguments to get the Senate to disagree with the solution of such councils.

Employers make an enormous investment by offering internships to many students. These internships last for four to six months. Industry experts with higher education are often asked to serve as academic advisors to final theses, particularly at colleges and at the bachelor's degree level.

At the national level, an important annual event is the meeting between members of the Professional Education Council and directors of ICT study programmes at universities and colleges. Education issues are also usually on the agenda of the annual LIKTA conference, as well as of the international "Baltic IT&T" conference. The most important requirements of the ICT sector are included in LIKTA declarations which are then submitted to the government.

Universities and colleges are willing to include elective courses in their curricula which are provided by ICT companies. Companies are expected to provide the necessary equipment, software, textbooks and lector. Alternatively, they can provide financing for the course. This approach has led to the fact that many Latvian universities and colleges have the Microsoft IT Academy, CISCO Academy, and study courses provided by the Exigen Services DATI, the Baltic Technology Group, Tilde, IBM Latvia, etc.

A particularly high level of academic co-operation involves doctoral dissertations written on subjects that are of interest to companies in the ICT sector. Authors can use the infrastructure and information base of these companies as they write their dissertations.

## 6 The University of Latvia Computing Study Programme as an Innovative Approach to Education

The ICT sector in Latvia and the world has experienced a very rapid growth over the last 10 years. According to the Ministry of Economics of Latvia, the sector produces 5 to 6% of Latvian GDP, and exports are worth nearly EUR 150 million. The sector has been declared a national priority by several governments, but in the Latvian language, sadly, it does not have a single name. The terms that are used, as translated into English, include "computer studies", "informatics", "information technologies" and "information and communications technologies." On April 4, 2006, the Cabinet of Ministers approved the order No. 267 [7] to announce that the word "datorika" is to be used as a translation for the word "computing". The term "computing", as we know [1] refers to a thematic part of education – the one which covers computer sciences, information technologies, information systems, software engineering, and computer engineering. The objectives of higher education in terms of supporting the further development of the computing sector are the following:

1. Prepare highly qualified and export-capable specialists for practical work at companies and government institutions – specialists who not only can design and produce complex information systems, but also manage projects and independently learn about new technologies in the rapidly changing environment of computing;
2. Prepare academically educated specialists who are prepared to do scientific work in the computing sector – research projects in the computer sciences in Latvia, as well as expert participation in the evaluation of new technologies and systems.

These are contradictory requirements because an academic higher education is based on science, while the knowledge that is needed in practice is based on engineering and the study of production processes.

The proposal is to train computing specialists on the basis of a four-level pyramid:
1) The college level, which trains software designers and computer network administrators with a level of knowledge and skills equal to the first-level higher professional education standard;
2) The bachelor's level, where students learn not only about software design, but also about the design and development of complex software systems;
3) The master's level, where students learn to analyse and design large systems, and to run projects;
4) The doctoral level, where highly qualified specialists are trained to work with major and complex projects and to work at universities.

This proposal regarding the training of computing specialists has been approved at several meetings of company representatives and university representatives (the first in 2000, with the participation of Ministers of Economics and Education, another during the November 2004 LIKTA conference, the third organised by Exigen in March 2005, etc). The focus on the demands of the Latvian economy for highly qualified computing specialists is very different from other exact science study programmes at the University of Latvia, because these programmes are focused, at least formally, on the training of scientists and teachers.

The objective of the proposed programmes [8] is not only to ensure that specialists are trained at all four higher education levels, but also to ascertain that there are opportunities to pursue all five areas of specialization (disciplines):
1) Computer science (CS), where the programme covers the mathematic processes of computer science, system modelling and issues related to artificial intellect;

2) Information technologies (IT), covering the design and use of computer networks and clusters, as well as sound and image processing;
3) Information systems (IS), focusing primarily on database management systems, as well as the design, implementation and maintenance of information systems;
4) Software engineering (SE), focusing primarily on software design and production of software, including embedded systems;
5) Computer engineering (CE), which covers the design and manufacturing of electronic equipment.

In what sense are these study programmes original?
1) *All areas of computing are covered in one programme at each level of higher education*. During the first two study years, students can choose to pursue SE to receive the qualification of "Software designer", or they can study IT and receive the qualification of "Computer network administrator". This is a choice that has to be made at the beginning of the second study year. During the first two years, the two areas of specialisation differ only in terms of internships that are worth 24 ECTS credits (16 weeks). The internships are organised in the fourth semester. There are also 12 ECTS credits for writing the thesis, and work on that begins in the third semester. Those specialising in SE need software design practice and a qualification thesis in software design. Those pursuing IT must engage in an internship focused on computer network administration and also write a paper of the same kind. Internships can begin before the fourth semester. In the third study year (the fifth semester), students can choose any of the five areas of specialisation, irrespective of the diplomas or qualifications that they already have: CS (more theoretical), SE (more focused on software design), IS (more focused on the design and maintenance of information systems), IT (more focused on computer networks), or CE (more focused on the construction of electronic equipment). This study programme organically merges the study of fundamental aspects of the profession with vast opportunities for specialisation and theoretical study. It is the only programme of this kind in Latvia.

2) To a certain extent, this programme represents a return to the Soviet system, which provided for a *mandatory semester-long internship outside of the educational institution at the conclusion of the 2nd year of studies*. This internship allows young people to decide whether they have made the right choice in terms of their study programme and their selection of a profession. They also begin to accumulate professional experience, and that is often the first criterion for hiring a new employee.

3) There are *very close links to the industry* – representatives of employers are on the councils of study programmes and on the commissions which test people's qualifications. There are more than 50 contracts on internships and specialised courses of study provided by leading IT companies such as Microsoft, CISCO, the Exigen Services DATI, the Baltic Technology Group, etc.

4) *There are still powerfully academic and research-based studies* in the upper years of bachelor's degree studies and, of course, at the master's and doctoral level. Instructors at the University of Latvia are equally strong in theory and practice. The science citation index of peer reviewed scholarly publications produced by instructors

at the University exceeds the total number of science citation index of such publications produced by all other instructors and scientific researchers in Latvia, Lithuania and Estonia taken together.

5) The study plan is structured so that *when a student receives a diploma, he or she can pursue further studies in any other area of specialisation without having to take a catch-up course*. The specialization does not have to be selected before the young people start their studies – often they have a fairly unclear understanding of the programme of study and of their future profession. The first choice – software designer or computer network administrator – must be made at the beginning of the second year, while the second – CS, SE, IT, IS or CE – must be made at the start of the third year. The first university diploma (college level) is received at least one year sooner than elsewhere in Latvia.

## 7 Is the Bologna Process Moving in the Right Direction?

Once we come to the firm understanding that there are very different areas of knowledge and that those which are of use in theology are not of use in physics and vice versa, it will be easier for us to understand that a unified process is not possible without specific exceptions. Europeans tend to move toward three-year bachelor's degree programmes, which may be all right for humanities, but is certainly unacceptable for those areas of study which cannot be imagined without serious internships (medicine and engineering, including computing). One cannot understand at what expense Europeans are trying to achieve the main goal of the so-called Bologna process – to compete with the United States and to surpass the USA in terms of the level of higher education. Do we have far better instructors and far more talented students so that we can achieve in three years what the Americans achieve in four?

This means that in the higher education system related to ICT, there should be no study programmes which allow students to receive a diploma without a serious internship. Three-year programmes with no internship opportunities are at all absurd. The so-called academic study programmes in engineering produce hundreds of young people each year – young people without the slightest industrial experience.

**Acknowledgments.** The author would like to extend his gratitude to colleagues that participated in creation of ICT professional standards and qualification requirements: Vitālijs Aišpurs, Inna Allena, Baiba Apine, Sandijs Aploks, Anda Ādamsone, Elmārs Beķeris, Normunds Bergs, Modris Bērzonis, Vilnis Bēts, Zane Bičevska, Jānis Bičevskis, Jānis Bikše, Juris Binde, Harijs I. Bondars, Jānis Brants, Liene Brīvule, Solvita Brokule, Ziedonis Bunžs, Ivars Ciesalnieks, Inese Cvetkova, Pēteris Čaurs, Harijs Čivkulis, Dainis Dosbergs, Ainārs Dubra, Avroms Dušanskis, Renata Frolkova, Ainārs Galvāns, Mārtiņš Gills, Egils Ginters, Irvins Gustsons, Sergejs Ilnickis, Andris Jakubovskis, Aleksandra Joma, Jānis Grundspeņķis, Jānis Kaģis, Edīte Kalniņa, Helga Kaukule, Arts Klints, Inārs Kļaviņš, Andris Kovaļevskis, Pēteris

Krastiņš, Jānis Kručeks, Didzis Kukainis, Inga Kullesa-Lapiņa, Edvīns Laucis, Gunārs Lauks, Valdis Lauks, Jānis Lelis, Inese Lenša, Edīte Liekmane, Sanda Linde, Andrejs Lūsis, Maija Ļaksa, Jekaterina Ļevčenkova, Juris Majors, Arnis Mamajs, Ģirts Mamontovs, Gunārs Matisons, Ainis Mūsiņš, Boriss Obermanis, Ivo Odītis, Ilmārs Osmanis, V.Pētersons, Romass Pauliks, Anda Pavlova, Voldemārs Plavoks, Jānis Plūme, Māris Puriņš, Inese Purmale, Guntis Romanovskis, Olga Sabanska, Nadežda Semjonova, Jānis Siliņš, Uldis Smilts, Romualds Smiļģis, Arnis Smirnovs, Ilze Spektore, Uldis Stūre, Uldis Sukovskis, Dainis Susejs, Natālija Suvorova, Normunds Svētiņš, Darja Šmite, Ziedīte Šmite, Agris Šnepsts, Evija Tauniņa, Sandra Tene, Ilona Valdate, Larisa Zaiceva, Anna Zalucka, Dzintars Zariņš, Viesturs Zūlis. This work was partly supported by the EU PHARE Program's "Professional Education 2000" project "Establishment of a Professional Qualifications Structure".
The innovative computing study programme at the University of Latvia was designed mainly by Guntis Arnicāns, Guntis Bārzdiņš, Jānis Bārzdiņš, Jānis Bičevskis, Jānis Cīrulis, Kārlis Čerāns, Rūsiņš-Mārtiņš Freivalds, Ēvalds Ikaunieks, Audris Kalniņš, Paulis Ķikusts, Kārlis Podnieks, Uldis Straujums, Māris Treimanis, Juris Vīksna, Māris Vītiņš. This work partly was supported by European Union Structural Fund project VPD1/ESF/PIAA/04/APK/3.2.3.2/0069/0063 "Modernization of Computer Science Degree Programs in the University of Latvia".

## References

1. Computing Curricula 2005: The Overview Report, 2005, p. 62 See http://www.acm.org/education/curric_vols/CC2005-March06Final.pdf . Last viewed 18 September 2007.
2. Curriculum Development Guidelines. New ICT curricula for the 21st century: designing tomorrow's education. Luxembourg: Office for Official Publications of the European Communities, 2001, p. 58 See http://people.ac.upc.edu/toni/papers/CurrITEng.PDF . Last viewed 18 September 2007.
3. See www.izmpic.lv . Last viewed 18 September 2007.
4. International Standard Classification of Education ISCED-97. United Nations Educational, Scientific and Cultural Organisation, May 2006, re-edition, p. 48. See www.uis.unesco.org/TEMPLATE/pdf/isced/ISCED_A.pdf . Last viewed 18 September 2007.
5. Lūsis, A., Siliņš, J., Sukovskis, U., Zariņš, D., Bikše, J., Borzovs, J., Ginters, E., Kaģis, J., Kļaviņš, I. and Lelis J. Informācijas tehnoloģijas, telekomunikācijas un elektronikas nozares apraksts (A Description of the Information Technology, Telecommunications and Electronics Sector). Rīga: PIAPA and PIC (2000), p. 51.
6. The professional standard "Software developer". Registration No. PS 0001. Approved by order of the Ministry of Education and Science, No. 145, 12 March 2001, amended by order No. 649, 29 December 2003. See http://www.izmpic.lv/index2.html . Last viewed 18 September 2007.
7. The Republic of Latvia education classification, approved by order of the Cabinet of Ministers, No. 267, 4 April 2006. See http://izm.izm.gov.lv/normativie-akti/mknoteikumi/932.html . Last viewed 18 September 2007.
8. See http://www.aiknc.lv/lv/prog_view.php?id=5361 . Last viewed 18 September 2007.

LATVIJAS UNIVERSITĀTES RAKSTI
733. sējums, DATORZINĀTNE UN INFORMĀCIJAS TEHNOLOĢIJAS

---

Latvijas Universitātes Akadēmiskais apgāds
Baznīcas ielā 5, Rīgā, LV-1010
Tālr. 67034535