University of Latvia

SERGEJS RIKAČOVS

THE BASE TRANSFORMATION LANGUAGE, ITS IMPLEMENTATION AND APPLICATIONS

Thesis submitted for the degree of Doctor of Science (Ph. D.) in Natural Sciences in the field of Computer Science and Informatics

Subfield of Programming Languages and Systems

Scientific Advisors:

Prof. Dr Habil. Sc. Comp.
JĀNIS BĀRZDIŅŠ

Prof. Dr Sc. Comp.
KĀRLIS ČERĀNS

Riga – 2024

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

This work was developed with the support of the European Social Fund in the project "Strengthening the doctoral capacity of the University of Latvia within the framework of the new doctoral model." No. 8.2.2.0/20/I/006.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

General Description of the Thesis

1.1 Relevance of the Thesis

Nowadays, software systems are constantly becoming more complicated. In the nineties and even earlier it became clear that new software system building methods that could radically facilitate the implementation of a software system are needed. In 2001, a principally new paradigm for software system building – Model Driven Architecture (MDA) emerged. MDA [11] is based on two fundamental concepts: metamodels and transformations of metamodels. In the simplest situation, ideas of MDA can be characterised by classical PIM [11] and PSM [11] models. PIM is a platform-independent model written in a metamodeling language. PSM is a platform specific model, also written in a metamodeling language. As a result, the main software system building process is reduced to transformation from PIM to PSM. The PSM model is implemented universally. For example it could be a Java compiler, in the case if PSM is a Java metamodel. As a consequence, the development of expressive, efficient and easy usable transformation languages became one of the most topical problems in software engineering. A transformation language is a special language that takes instances of one metamodel as input and produces instances of another metamodel as output.

When the MDA paradigm was formulated, development and implementation of model transformation languages began. However, practice has shown that the implementation of model transformation languages is a non-trivial problem that in contradistinction to the case of traditional programming languages, still lacks a commonly agreed solution. The main reason for this is the use of patterns in advanced transformation languages. However, the efficient implementation of patterns can sometimes involve an exhaustive search and in the worst case can be asymptotically exponential. It became clear that the development of efficient compilers for advanced model transformation languages is a non-trivial and rather expensive process.

The author of this thesis became involved in the research of these problems in 2007, when it became clear that solutions, distinctive in some sense from those for traditional

programming languages, had to be found for the efficient and moderately expensive implementation of model transformation languages.

1.2 Aim of the Research

The main goal of the research is to offer an effective solution to the problem of model transformation language implementation and to approbate this solution in practice.

The first task of this work is to develop and effectively implement a base transformation language that supports simple operations for working with metamodels, and with the help of which, using the bootstrapping method, it would be feasible to implement practically usable higher-level transformation languages.

The second task of this work is further development of the base transformation language and its implementation, by supplementing the language L0 with an extension for metamodel processing. We provide an experimental solution for extending the language L0 with undo/redo options, as well.

The third task is approbation of the developed language in practice.

The fourth task is to check if it is feasible to migrate relational data to RDF by using model transformations.

1.3 Hypotheses, Research Issues

- It is possible to create such low level model transformation language, that would be efficiently implementable and at the same time it would be possible to use this language both directly (as a standalone model transformation language) and in the process of implementation of higher-level model transformation languages via bootstrapping process
- It is possible to effectively use model transformation languages for migration of relational data bases to RDF

1.4 Research Methods Used

Following scientific methods have been used in the thesis research:

- Scientific literature analysis
- Comparative analysis of existing solutions
- Experimental development, to validate proposed ideas and solutions.

1.5 Main Results of the Thesis

The main results of this thesis are the following:

1. a new low level textual model transformation language L0 for the processing of metamodel instances has been proposed. An extension of this language – language L0+ for the processing of meta metamodel instances has been proposed as well.
2. Efficient implementation of L0 and L0+ has also been developed. A compiler for L0 language has been implemented. This compiler can produce target code in C++ or Java. Generated code can operate on top of mii_rep [44] and JAPIER [41] data stores in the case of C++, and EMF [37] and JGRALAB [43] data stores in the case of Java.
3. In cooperation with the co-authors, a bootstrapping schema for the implementation of higher level model transformation languages has been developed. This schema is based on L0 language and its metamodel from the one side and an arbitrary higher level language and its metamodel on the other side. Further gradual compilation from high-level transformation language to lower level languages is carried out by transformations (that are models as well) written in L0 language. This schema has been practically used for the implementation of higher level language MOLA [25, 26] in the context of A. Šostak's Ph.D. thesis [52]. The obtained results confirm the efficiency of the proposed schema.
4. Undo/redo implementation principles, based on L0 language have been developed. These mechanisms allow cancelling of the result of execution of certain L0 commands.

5. A series of experiments devoted to the application of model transformations in a principally new domain – migration of relational data bases to RDF has been performed. As a result, a new model transformation-based method for the migration of relational data bases to RDF has been developed. Implementation of this method is based on L0 language. The migration experiments performed have confirmed the practical applicability of model transformation in the domain of relational DB migration to RDF.

1.6 Scientific and Practical Significance of the Thesis

The most important results of the thesis are:

- development of a simple, yet efficiently implemented base transformation language
- development of a bootstrapping scheme (along with co-authors) for effective implementation of higher level model transformation languages
- development of language level undo / redo mechanism implementation principles
- development of a data migration method based on metamodel transformation languages and approbation of it in practice.

The practical significance of the results is confirmed by real usage of the base transformation language in the following contexts:

1. A high-level model transformation language MOLA is implemented through a bootstrapping schema, where L0 is used as a base language [1, 52]. (see also section **7.2** Implementations of High-Level Model Transformation Languages).
2. A graphical tool building platform GrTP [18] is developed on the basis of L0 language. In the context of this platform, language L0 is used for the specification of an internal logic of a concrete tool. It is important to note that the internal logic of the platform itself is also specified in L0. GrTp has been widely used for development of DSL tools. For instance, the following DSL tools [4, 82, 83, 84] were created with the help of the GrTp platform for the needs of the Latvian national economy (see also section **7.3** Tool Building Platform - GrTP).

3. With the help of a method for the migration of relational DB to RDF, developed in this thesis, we performed the migration of medical registries. During this migration process, the data of several medical registries (where information about the main pathologies that threaten the quality of life of Latvian citizens is gathered), which were stored in the relational database, were transformed into a single data warehouse, where the data is stored in the RDF database. This made it possible to use the graphical query construction tool ViziQuer [4] for ad hoc data analysis, which, in turn, made it possible to provide a new quality of end-user interaction with real medical data at the ontology level [4, 5, 6, 9]. The process of this migration was implemented in cooperation with the Centre for Disease Prevention and Control (which maintains the mentioned medical registries) within the framework of VPP project No. 14 "Creation of a single and accessible database for the main pathologies threatening survival and quality of life and the prevalence of their risk factors in the population of Latvia".

The results of the research have seen the following practical applications:

- the L0 language developed in this thesis was used as a base language in the bootstrapping process, within which the efficient implementation of the high-level graphical model transformation language MOLA has been created [52]. Compared to the previous implementation of MOLA, it was possible to achieve a significant performance improvement (up to 65x times better in some examples [52]). This practical application confirms the effectiveness of L0 implementation.
- the L0 language was used to write transformations that define the internal logic of a specific tool in the context of the GRTP platform [18]. These transformations link instances of the domain model to instances of the user interface model. The fact that the L0 language was chosen as the transformation language for specifying the internal logic of the platform is confirmation of the fact that L0 language is practically usable not only as a target language in the process of higher-level language implementation, but also as a model transformation language suitable for the direct development of transformations.

- Several DSL tools [4, 82, 83, 84] were created for the needs of the Latvian economy on the basis of the GrTp platform (L0 language was used for the specification of the internal logic of newly developed DSL tools)
- L0 language was used for the migration of medical relational databases to RDF [5, 6, 9]. During the migration process, the data of several medical registers, which were stored in the relational database, were transformed into a single data warehouse, where the data is stored in the RDF database. This made it possible to use the graphical query construction tool ViziQuer [4] for ad hoc data analysis. ViziQuer allows the creation of non-trivial queries without prior knowledge of data query languages. Previously, when the data was in a relational database, this kind of analysis was not practically possible. The ViziQuer tool was built using the GrTp platform. Data analysis within the aforementioned RDF data warehouse was one of the most serious initial uses of the ViziQuer tool. The further development of the ViziQuer tool is discussed in M. Zviedris' dissertation [85].

1.7 Approbation of the research results

1.7.1 Publications of the Research Results

The main results of the PhD thesis are presented in the following 10 author publications (6 of them in international peer-reviewed journals / conference papers with a calculated citation index (SCOPUS))**:**

1. J. Barzdins, A. Kalnins, E. Rencis, S. Rikacovs, ***Model Transformation Languages and their Implementation by Bootstrapping Method***. Lecture Notes in Computer Science, Springer Verlag, 2008, pp. 130-145. **(SCOPUS)**
   Contribution - 25%.

2. S. Rikacovs, ***The base transformation language L0+ and its implementation***, Scientific Chapters, University of Latvia, "Computer Science and Information Technologies", 2008, pp. 75–102.
   Contribution - 100%.

3. J.Barzdins, S. Rikacovs, ***Towards a Seed Transformation Language and Its Implementation.*** In Alexander Pretschner, editor, Models 2008, Toulouse, France, 28 September - 3 October 2008, Doctoral Symposium., volume 606 of ETH Zürich Technical Report, pages 27–32, September 2008.
   Contribution - 50%.

4. G. Barzdins, S.Rikacovs and M. Zviedris, ***Graphical query language as SPARQL frontend***, ADBIS 2009, Local Proceedings, 2009, pp. 93-107.
   Contribution - 10%.

5. Guntis Barzdins, Sergejs Rikacovs, Marta Veilande, Martins Zviedris, ***Ontological Re-engineering of Medical Databases***, Proceedings of the Latvian Academy of Sciences. Section B. Natural, Exact, and Applied Sciences., 2009, Versita, Warsaw, pp. 156-158. **(SCOPUS)**
   Contribution - 10%.

6. Sergejs Rikacovs, Janis Barzdins, (2010) ***Export of Relational Databases to RDF Databases: a Case Study***, P. Forbrig and H. Günther (eds.), in proc. of Business Informatics Research (BIR 2010), Springer LNBIP 64, pp. 203 – 211. **(SCOPUS)**
   Contribution - 80%.

7. S. Kozlovics, E. Rencis, S. Rikacovs, K. Cerans, ***Universal UNDO Mechanism for the Transformation-Driven Architecture***, DBIS 2010. In Proceeding of the Ninth International Baltic Conference, DB&IS 2010, pp. 325-340, 2010.
   Contribution - 15%.

8. S. Kozlovics, E. Rencis, S. Rikacovs, and K. Cerans ***A kernel-level UNDO/REDO mechanism for the Transformation-Driven Architecture***. In Databases and Information Systems VI - Selected Papers from the Ninth International Baltic Conference, DB&IS 2010, pp.80-93, 2010. **(SCOPUS)**
   Contribution - 15%.

9. S. Rikacovs, ***Export of Relational Databases to RDF Databases by Model Transformations***, in proc. of Business Informatics Research (BIR 2011), Springer LNBIP 90, pp. 158 – 166. **(SCOPUS)**
   Contribution - 100%.
10. S. Rikačovs, L. Lāce, E. Rencis, A. Šostaks, K. Čerāns., ***An Overview of Practical Applications of Model Transformation Language L0***, Baltic J. Modern Computing, Vol. 12 (2024), No. 1, pp. 1-14,. **(SCOPUS)**
    Contribution - 80%.

1.7.2 Presentations of the research results at Scientific Conferences

The results of the thesis have been presented by the author at the following scientific conferences:

1. Report „*Bootstrapping metodes lietojumi modeļu transformāciju valodu realizācijā*" at 64. Scientific Conference of the University of Latvia, Information Technology Section 2006; Riga, Latvia.
2. Report „*Modeļu transformāciju valoda L0+ un tās kompilācija*" at 65. Scientific Conference of the University of Latvia, Information Technology Section 2007; Riga, Latvia.
3. Report *„Towards a Seed Transformation Language and Its Implementation"* at Doctoral Symposium of MODELS (International Conference on Model-Driven Engineering Languages and Systems), 2008; Toulouse, France
4. Report „*Relāciju datubāzu migrācija uz RDF datubāzēm*" at 66. Scientific Conference of the University of Latvia, Information Technology Section 2008; Riga, Latvia.

5. Report „*Export of Relational Databases to RDF Databases: a Case Study*" at 9th International Conference on Perspectives in Business Informatics Research 2010; Rostock, Germany
6. Report „*Export of Relational Databases to RDF Databases by Model Transformations*" at 10th International Conference on Perspectives in Business Informatics Research 2011; Riga, Latvia

1.7.3 Thesis related scientific projects

- VPP project nr. 1 „ Development of model transformations based system building technologies", 2005-2009.
- Project VPD1/ERAF/CFLA/05/APK/2.5.1./000009/004: "Development of a new generation system modeling tool", 2006-2008.
- VPP project No. 14 "Creation of a single and accessible database for the main pathologies threatening survival and quality of life and the prevalence of their risk factors in the population of Latvia", 2006-2009.
- LZP FLPP projekts "Visual queries in distributed knowledge graphs" (Nr. lzp-2021/1-0389), 2022-2024

1.7.4 Applying results in practice

The methods developed in the thesis have seen following practical applications:

- The L0 language developed in the doctoral thesis was used as the base language in the bootstrapping process, within which the effective implementation of high-level graphical model transformation language MOLA has been developed [52]. Compared to the previous implementations of MOLA, it was possible to achieve a significant performance improvement (up to 65x in some cases [52]). This practical application confirms the efficiency of L0 implementation.
- The language L0 was used for development of transformations, defining the internal logic of a specific tool for in the context of the GrTp platform [18]. These transformations link instances of the domain model to instances of the user interface

model. The fact that the L0 language was chosen as the transformation language for specifying the internal logic of the platform is confirmation of the fact that L0 language is practically usable not only as a target language in the process of higher-level language realisation, but also as a model transformation language suitable for the direct development of transformations.

- Several DSL tools [4, 82, 83, 84] were created for the needs of the Latvian economy on the basis of the GrTp platform (L0 language was used for the specification of the internal logic of newly developed DSL tools)
- L0 language was used for the migration of medical relational databases to RDF [5, 6, 9]. During the migration process, the data of several medical registers, which were stored in the relational database, were transformed into a single data warehouse, where the data is stored in the RDF database. This made it possible to use the graphical query construction tool ViziQuer [4] for ad hoc data analysis. ViziQuer allows the creation of non-trivial queries without prior knowledge of data query languages. Previously, when the data was in a relational database, this kind of analysis was not practically possible. The ViziQuer tool was built using the GrTp platform. Data analysis within the aforementioned RDF data warehouse was one of the most serious initial uses of the ViziQuer tool. The further development of the ViziQuer tool is discussed in M. Zviedris dissertation [85].

1.8 Structure of the Thesis

This thesis has the following structure:

- In the first chapter we give a general description of the thesis
- In the second chapter the main ideas of MDA and its fundamentals are given.
- In the third chapter a new low-level textual model transformation language L0 is defined. This language has several important properties: it is very simple, it has efficient implementation, it can be used for the direct development of model transformations and as it will be shown later, this language can be used as a base

language in the process of higher level model transformation language implementation through the bootstrapping method.

- In the fourth chapter we provide a review of related work in the area of low-level model transformation languages. We also highlight the main differences between other low-level model transformation languages and L0 language.
- In the fifth chapter a demonstration of extending L0 language with new features is given. More precisely an L0' language is defined, that supports undo/redo facilities, and the main principles of L0' implementation are given.
- In the sixth chapter we describe a model transformation-based method for the migration of relational DB to RDF.
- In the seventh chapter, applications of L0 language are described. L0 applications in the area of high-level model transformation language implementation by the bootstrapping method and L0 application in the context of a graphical tool building platform GrTP are given. These applications confirm the practical usability of L0 language.
- In the eigth chapter we review the recent trends in the area of model transformations and model based graphical tool buildings platforms
- The ninth chapter concludes the thesis.

CHAPTER 2

Models, Metamodels and Transformation Languages

Model Driven Architecture (MDA) is an approach to building complex software systems. This approach is based on model transformations [11]. According to MDA (Fig. 1), firstly an implementation independent system metamodel is built. It is called a Platform Independent Model – PIM. Secondly a metamodel for the implementation environment is built, called a Platform Specific Model – PSM. Then with the help of automatic transformation PIM is transformed to PSM that is "understood" by a computer [11].

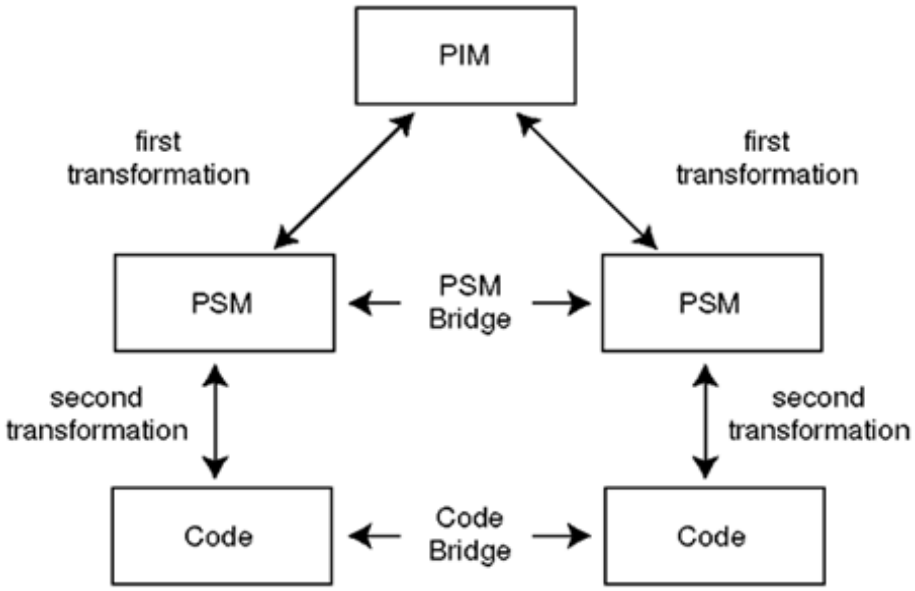

Fig. 1 MDA conceptual schema [11]

Critically important elements of this approach are efficient (from a performance point of view) model transformation languages.

2.1 Models and Metamodels

An important concept that is frequently used in this thesis is the concept of a model. In the context of this thesis we will follow classical definitions from [11] and with the model will understand the description of a system given in a precisely defined language – a language with precisely defined syntax and semantics. Metamodeling is normally used for defining modeling languages. The model describes what elements can exist in a system. In a similar

way, metamodeling language describes what elements can be used while creating a model. For example UML language says that we can use such concepts as “Class” or “Package”, while creating UML models. If we need to formalise a modeling language than we can describe this language with a model, called a metamodel of a language. Model of a language describes those elements that can be used in a language that is being formalised. Every element that a model creator is allowed to use is defined by a metamodel of a language being used. For example, in UML you can use classes, associations, packages and attributes, because in the UML metamodel there are elements that define classes, associations, packages and attributes. Taking into account the fact that metamodel is also a model, metamodel has to be described in some well-defined language. Language that is used to describe metamodels is called meta language [11]. OMG provides a Standard meta language – MOF [13]. For example a UML language is defined in MOF. According to MDA, we have a source model and a target model. Both source and target models conform to specific metamodels. Metamodels are defined in a Standard metamodeling language MOF. Transformation of the source model into a target model occurs according to a transformation definition that is written in a transformation language.

2.2 Transformation Languages

Every model transformation language is a specialised programming language. In contradistinction to general programming languages, model transformation languages are specifically tailored for the processing of metamodels and instances of metamodels. Model transformation languages are a relatively new kind of language. The first standardisation effort in this area was MOF 2.0 Query/Views/Transformations (QVT) request for Proposals (RFP) issued by OMG [15]. By answering this request a specification [16] has been developed. Later, this specification became known as QVT language. QVT defines a standard method of transformation description. According to QVT, source and target models have to conform to the MOF metamodel, and it has to be defined with metamodels written in MOF. For a description of transformations, QVT provides 2 languages:

- QVT Relations
- Operational mappings

Relations language is a high-level model transformation language, where a sufficiently developed concept of a pattern is present. For relations language there is an alternative graphical form. Operational mapping language is very similar to OCL language [12, 17], which is supplemented with new (mostly imperative) constructs, such as loop, condition etc.

Excluding QVT, independent model transformation languages exist: ATL [21], Tefkat [22], UMLX [19], GREAT [20], AGG [24], VIATRA [23], Fujaba [35], Epsilon [69], MOLA [25, 26]. It is believed that model transformations should mainly be specified in graphical form, and combining the graphical form with the textual, when necessary.

Every language that has advanced constructs such as patterns, has some serious problems with its implementation (pattern match implementation problems). One more related problem is the selection of the runtime environment. A standard choice for a biggest part of transformation tools are metamodel based in-memory data stores, such as EMF [36, 37], MDR [38] or similar. These data stores have a universal low-level API for the processing of model elements. As a consequence, the implementation of the pattern matching procedure becomes rather difficult [28].

CHAPTER 3

The Base Transformation Language

3.1 Introduction

The main goal of this chapter is to define the base transformation language that satisfies the following requirements:

- this language is quite simple (easy to learn and easy to use)
- effective implementation of this language exists
- language contains minimal, but sufficient constructs for the development of model transformations. The language has to be usable for direct transformation development.

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 normally used in practice. We will follow this tradition and use a subset of MOF 1.4 [14] seen in Fig. 2 to define metamodeling constructs to be allowed in metamodel definitions. One can notice that packages are not present as an independent concept in this metamodel. In L0, packages are simulated through qualified names.

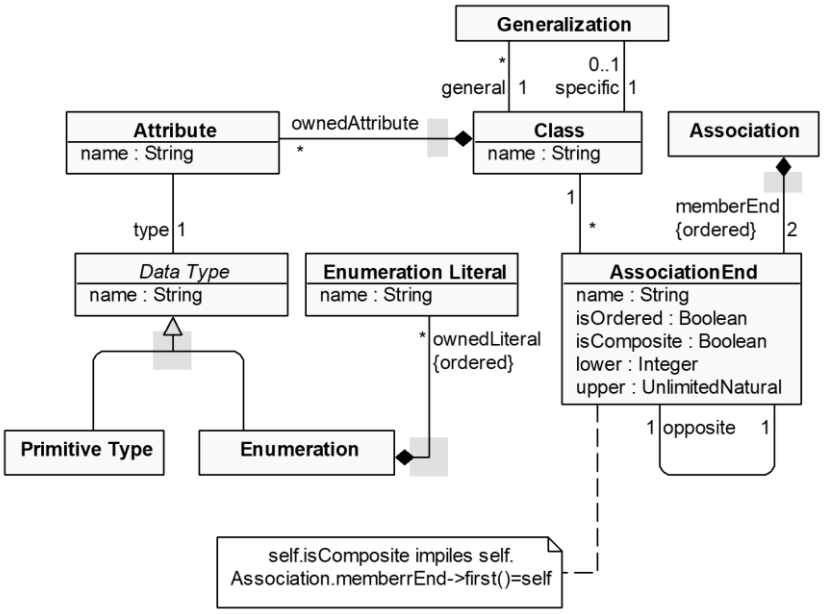

Fig. 2 Meta-metamodel

3.2 L0 Definition

In a broader sense, L0 language consists of 2 parts:

- the part, containing constructs for model instance processing. This language is called L0 (or even more precisely – base L0)
- the part, containing constructs for metamodel instance processing. The base L0 language supplemented with metamodel processing constructs is called L0+. We note that our practice has shown that in real applications it is often necessary to work with both levels (model and metamodel).

In the next section, we will provide a detailed definition of the base language L0, hereinafter referred to simply as L0. A description of the L0+ language will be given after that (see section **3.3** – An extension of a base transformation language L0).

3.2.1 Structure of L0 Transformation Program

A transformation defined in L0 language will be called an L0 transformation program. In the most typical case, the L0 transformation program consists of the following sections (see Fig. 3):

1. transformation header
2. preprocessor directives section
3. global variable definition section
4. a “native” subprogram (function or procedure) declaration part
5. L0 subroutine definition section
6. transformation footer

Additionally, a transformation program can have a native subroutine declaration section, which is used to provide the ability to call subroutines written in another programming language. Typically these are C++ functions adhering to special conventions.

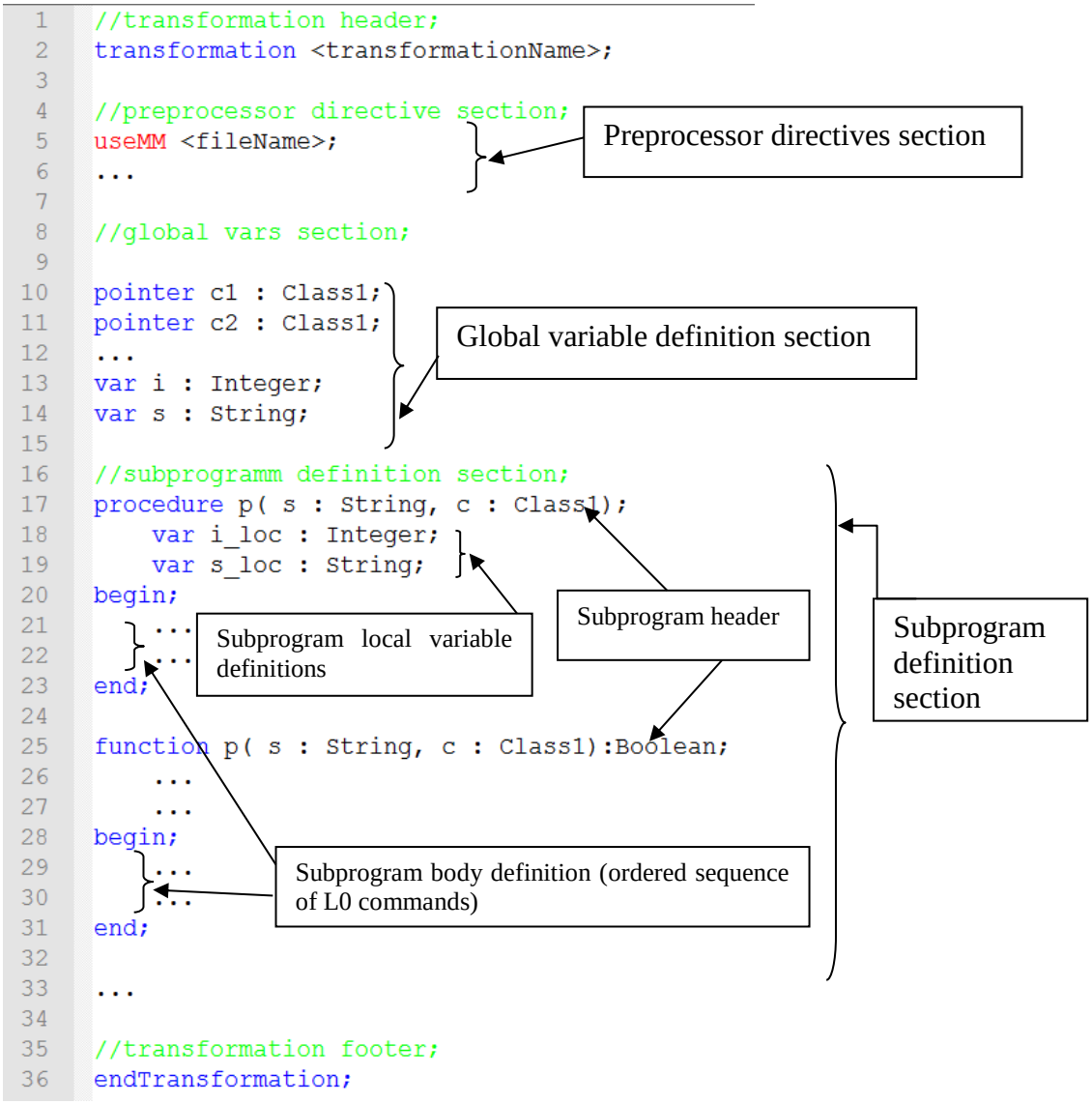

Fig. 3 Structure of a typical transformation program

Let's take a closer look at each of the sections.

1. Transformation header. This section starts the transformation program. The following L0 command is used for this: **transformation** <transformationName>;
2. Preprocessor directives section. As the name implies, this section may contain various preprocessor directives. The most commonly used L0 directive is **useMM** directive. It allows one to provide a path to a metamodel definition file. This file can contain the following commands:
   - **MMDefStart;**

Starts metamodel definition file.

- **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 generalisation relationship between given classes.

- **enum** <enumName>**:{** <enumLiteral1> **,** < enumLiteral2>, … **};**

  Defines an enumeration with given elements.

- **MMDefEnd;**

  Ends the metamodel definition file.

Other more "technical" directives (**useUnit**, **include**, **useLib**) are available in L0 as well. These directives are necessary to organise work with larger programs. They allow the program text to be split into several files, organising the use of external modules (typically C++ function calls) and so on. Let's describe these directives in more detail:

- **useUnit** "<file_path>" – directive allows dividing the transformation program into several units (instead of working with one huge file). After this division has been performed the current unit can only access subroutines from other units after explicitly stating that other units are being used by the current unit (which is done by including **useUnit** directives with a path to outer units in the current unit).

- **include** "<file_path>" – essentially works analogously to a C-style include. In L0, is usually used for including declarations of native subroutines to the current unit.
- **useLib** "<filePath>" – directive allows one to add the already compiled library file (path to this file is given inside <path>) to the executable to be created.

3. A global variable definition part; it is allowed to define both variables of primitive types with the help of the **var** command (see point 4 in section 3.2.2 – The list of L0 commands) and references to object instances with the help of the **pointer** command (see point 3 in section 3.2.2 – The list of L0 commands).
4. A "native" subprogram (function or procedure) declaration part (headers of C++ functions used in the transformation program). As with 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, a possibility exists to call a C++ function. For example, there is a String data type in L0, but the language per se does not define some useful operations such 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.
5. An L0 subprograms definition part. An L0 subprogram definition also consists of several parts:
   - the subprogram header; commands **procedure** and **function**; (see points 5 and 6 in section 3.2.2 – The list of L0 commands)
   - local variable definitions; commands **var** and **pointer**; (see points 3 and 4 in section 3.2.2 – The list of L0 commands)
   - the keyword **begin**
   - the subprogram body definition; a sequence of L0 action commands (see section 3.2.2 – The list of L0 commands)
   - the keyword **end**

   It is expected that exactly one subprogram in this part is labelled with the reserved word **main**, thus defining the entry point for the transformation.

6. A transformation footer. This section ends the transformation program. It consists of just one command, i.e., **endTransformation;**

3.2.2 The List of L0 commands

The L0 language is a textual low-level imperative procedural language. It contains a minimal but sufficient set of commands for processing models. Control flow is organised by using low-level control flow control commands. Access to class instances is organised using the concept of a typed reference – at one particular moment it points to exactly one object (or to the special **null** value). In the program, these typed references are introduced using the **pointer** command. Association instances are accessed using two references that point to the source and target objects of the corresponding link and the name of the association role. Manipulation of attribute values typically occurs by first reading the value of the attribute into a variable of elementary type (introduced with the **var** command), then processing the value of that variable and writing it back to the object's attribute.

All action commands available in L0 can be grouped as follows:

- commands for creating and destroying objects (**addObj**, **deleteObj**)
- commands for creating/deleting links (association instances) (**addLink**, **deleteLink**)
- command for reading / setting attribute values (**setAttr**)
- commands for instance traversing (**first**, **first from**, **next**)
- low-level control flow control commands
    - labels (introduced with **label** command)
    - commands for unconditional transfer of control (**goto**, **return**, **call**)
    - commands for conditional transfer of control (**type**, **var**, **attr**, **link**, **noLink**, **pointer**)
- commands for assigning variable/pointer values (**setPointer**, **setPointerF**, **setVar**)

Before delving into a detailed description of individual commands, it should be noted that the name of the metamodel 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 metamodel 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 metamodel 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 the 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 the 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, otherwise the control is transferred to <labelName>. Instead of equality symbol, the **==** inequality symbol **!=** can be used. Inheritance is not taken into account (i.e., this command works as oclIsTypeOf meaning the result of the comparison is true, if and only if types are identical).

    26.1. **type** <pointerName> **==** **(**<stringVarClassName>**)** **else** <labelName>;
27. **var** <varName>**==**<binExpr> **else** <labelName>; If the 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 the condition is not true, the control is transferred to <labelName>. Instead of the 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 the 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 the 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 the == inequality symbol, != can be used. If the condition is not true, the control is transferred to <labelName>. Instead of <pointer2> **null** can be used.

It is easy to see that L0 language 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.

3.2.3 Object-Oriented L0 Constructs

L0 was designed as a low-level language. Practice of application proved that it is possible to use this language for the direct development of transformations without significant loss of development speed.

For example, L0 is used for the development of transformations in the context of the Transformation Based Graphical Tool Building Platform – GrTp [18]. 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 [53], 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 advantage of the 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, the reserved word **this** can be used.

As can be seen, the only difference between the method declaration and the ordinary function or procedure declaration is the fact that the 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.

It should be noted that there are transformation languages providing much more advanced constructs. For example, in QVT Operational Mapping it is allowed to define the so-called mapping operation. This 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.

3.2.4 An Example of an L0 Transformation

Let's consider oriented graphs. Fig. 4 presents one possible metamodel for oriented graphs.

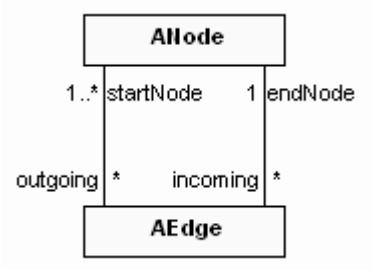

Fig. 4 Oriented graphs metamodel

According to this metamodel, a graph in Fig. 5 corresponds to the instance found in Fig. 6.

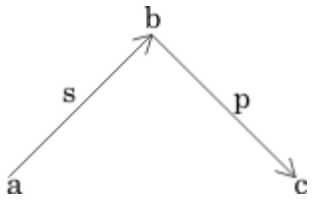

Fig. 5 An example of an oriented graph

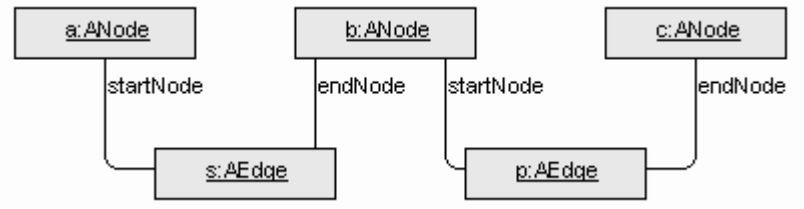

Fig. 6 Instances of the metamodel for oriented graphs

Several other metamodels (for example, the one found in Fig. 7) are also possible for oriented graphs.

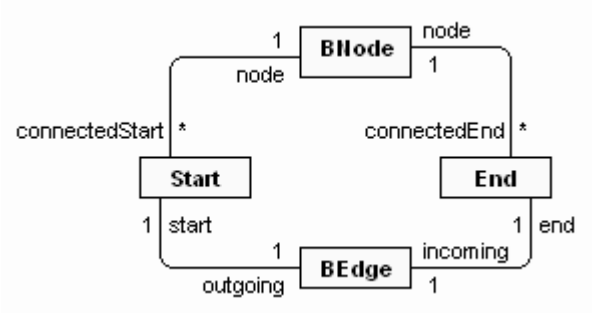

Fig. 7 Another metamodel for oriented graphs

According to this metamodel, a graph found in Fig. 5 will correspond to the instances found in Fig. 8.

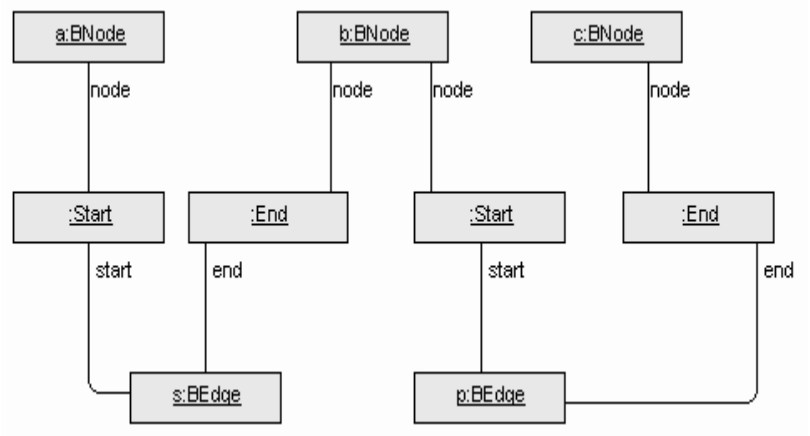

Fig. 8 Instances of another metamodel for oriented graphs

As 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 metamodel A and producing a graph model corresponding to metamodel B. The basic idea is to create one BNode for every ANode and to transform every AEdge to BEdge with the corresponding Start and End.

To simplify this transformation we add a mapping association to the metamodel between classes ANode and BNode (Fig. 9).

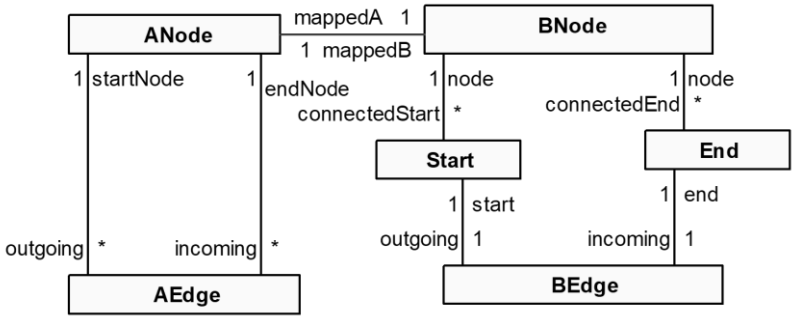

Fig. 9 Metamodel for oriented graphs with a mapping association

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

```
1  transformation Graphs;
2  useMM "graphs.mmd";
3  main procedure Graph2Graph();
4    pointer a : ANode;
5    pointer b : BNode;
6    pointer aEd : AEdge;
7    pointer bEd : BEdge;
8    pointer edgeStart  : Start;
9    pointer edgeEnd    : End;
10   pointer aEdgeStNode : ANode;
11   pointer aEdgeEnNode   : ANode;
12   pointer mapBNode : BNode;
13 begin;
14     //copy nodes;
15     first a : ANode else aNodeProcessed;
16     label loopANode;
17        addObj  b : BNode;
18        addLink a . mappedB . b;
19     next a else aNodeProcessed;
20     goto loopANode;
21     label aNodeProcessed;
22     //copy edges;
23     first aEd : AEdge else aEdgesProcessed;
24     label loopAEdge;
25         addObj bEd : BEdge;
26         addObj edgeStart : Start;
27         addObj edgeEnd : End;
28         addLink bEd.start.edgeStart;
29         addLink bEd.end.edgeEnd;
30         //quit if not found;
31         first aEdgeStNode : ANode  from aEd by startNode else aEdgesProcessed;
32         first mapBNode : BNode  from aEdgeStNode by mappedB else aEdgesProcessed;
33           addLink edgeStart.node.mapBNode;
34         first aEdgeEnNode : ANode  from aEd by endNode else aEdgesProcessed;
35         first mapBNode : BNode  from aEdgeEnNode by mappedB else aEdgesProcessed;
36         addLink edgeEnd . node. mapBNode;
37         next aEd else aEdgesProcessed;
38         goto loopAEdge;
39     label aEdgesProcessed;
40 end;
41 endTransformation;
```

Fig. 10 Transformation for oriented graphs

Let's take a closer look at what is happening in this transformation program. We can start with the overview of different sections of the transformation program. The section containing preprocessor directives can be seen in line 2; in this case it consists of only one directive, **useMM**. With this directive, we say that we will work with the metamodel whose definition is given in the file graphs.mmd. In this file, we give a textual definition (can be seen in Fig. 11) of a metamodel, shown in Fig. 9.

```
1  MMDefStart;
2
3      class ANode;
4      class AEdge;
5      class BNode;
6
7      class BNode;
8      class Start;
9      class End;
10     class BEdge;
11
12     assoc AEdge.outgoing/startNode.ANode;
13     assoc AEdge.incoming/endNode.ANode;
14
15     assoc ANode.mappedA/mappedB.BNode;
16
17     assoc BEdge.outgoing/start.Start;
18     assoc BEdge.incoming/end.End;
19
20     assoc Start.connectedStart/node.BNode;
21     assoc End.connectedEnd/node.BNode;
22
23 MMDefEnd;
```

Fig. 11 Metamodel definition (graphs.mmd) for oriented graph transformation

The global variables section is empty because we are not using global variables for this particular transformation program.

The subroutines section (lines 3-40) consists of only one subroutine, *Graph2Graph*, which is marked with the reserved word **main** to state that execution of the transformation must begin with the call (execution) of this subroutine. The *Graph2Graph* procedure consists of a local variable definition section (lines 4-12) and a section of procedure body definition (lines 13-40). In the body of the procedure a transformation is implemented, which, upon receiving an instance corresponding to the A metamodel (fig. 9), creates an instance corresponding to the B metamodel (fig. 9).

Two steps can be distinguished in the transformation: copying of vertices (*Node*) and copying of edges (*Edge*). Copying of vertices takes place on lines 15-21. In this step, we iterate through all instances of the class *ANode* and for each instance of it we create both: an instance of the class *BNode* (line 17) and a mapping association instance *ANode.mappedB.BNode* between the current *ANode* instance and the newly created *BNode* instance (line 18). We will need instance (link) of this mapping association in the next step to understand which object of class *ANode* corresponds to a newly created object of class *BNode*.

Iterating through instances of the *ANode* class is organised by using **first**, **next**, and **goto** commands to simulate a foreach loop. By executing the command **first** (line 15), the pointer *a* is positioned to the first unvisited instance of class *ANode* and control is passed to the next line (16). If there is no instance of the *ANode* class, the else branch is executed and control is passed to the line labelled with *aNodeProcessed* label. In lines 17 and 18, the loop iteration is executed – the current *ANode* instance is processed. Next, in line 19, the command **next** is executed, which sets the pointer *a* to the next instance of the class *ANode* that has not yet been visited by this pointer. If no such instance exists, the else branch is executed and control is passed to the line labelled with *aNodeProcessed* label. If such an instance exists, control is passed to the next command (line 20), which passes control to the line labelled with the label *loopANode* (16), thus continuing the loop.

Copying of edges takes place on lines 23-39. We loop through all instances of the *AEdge* class. For each instance of *AEdge*, we create an instance of the class *BEdge* (25), create instances of the classes *Start* and *End* (26 and 27), and also create instances of associations (links) *BEdge.start.Start* and *BEdge.end.End* between the newly created instances of classes *BEdge*, *Start* and *End* (28 and 29). Once this is done, we need to attach the newly created instances of the *Start* and *End* classes to precisely those objects of the *BNode* class that were created in the vertex copying step, and which correspond to the *ANode* objects attached to the currently processed instance of the *AEdge* class through links of type *AEdge.startNode.ANode* and *AEdge.endNode.ANode*. We encoded the correspondence

between the *ANode* and *BNode* in the previous step by using the *ANode.mappedB.BNode* association (so-called mapping link).

In the next paragraph, with a-edge we denote the object of class *AEdge,* and with b-edge we denote the object of class *BEdge*; with a-node we denote the object of class *ANode*, and with b-node the object of class *BNode*.

First, we will find the start vertices of the a-edge (the start vertex of the a-edge is an *ANode* object linked to an *AEdge* object with a link of type *AEdge.startNode.ANode*) and end vertices of an a-edge (the end vertex of an a-edge is an *ANode* object linked to an *AEdge* object with a link of type *AEdge.endNode.ANode*). To find the start vertex we need to find the *ANode* object that is bound to the current *AEdge* object via a link of type *AEdge.startNode.ANode*. This is done with the help of the **first from by** command in line 31. Once we have obtained a – starting node corresponding to the a-edge, we must obtain the b-node corresponding to this a-node. This is done by looking up the *BNode* object mapped to the *ANode* object via a link of type *ANode.mappedB.BNode* (see line 32, where **first from by** is used again). After executing lines 31 and 32, we have found the *BNode* corresponding to the starting node of the a-edge. Then we attach this *BNode* to the object *of* class *Start* with a link of type *Start.node.BNode*. Next, in lines 34-36, we proceed analogously to process the end node of the a-edge.

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”.

3.2.5 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, efficient implementation of patterns is one of the main challenges in the implementation of transformation languages [27, 28, 29], and, on the other hand, pattern match can be relatively easily specified with the help of L0 imperative constructs.

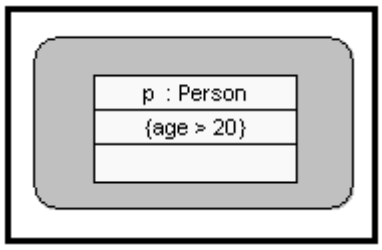

Fig. 12 Example of MOLA pattern

For instance, an example of MOLA **foreach** loop containing a pattern can be seen in Fig. 12. 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 checks 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. This 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.

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

There are use cases for model transformation languages where 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+.

3.3.1 Choosing Metamodel Processing Constructs

To allow transformation developer to process metamodels, we provide constructs to work with concepts defined in the metamodel found in Fig. 2. These are:

- classes
- attributes
- generalisation 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 that 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 generalisation hierarchies as well. In a similar way users can delete classes, attributes, associations, and generalisation hierarchies.

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. Analogous commands are introduced for scanning each kind of metamodel element:

- 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 by 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 the case that the attribute type is not known. One can notice that the type of dynamically created attribute is only unknown at the compile time, but is known to the programmer creating this program. This is why the values of dynamically created attributes can only be manipulated 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;

3.3.2 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.

Metamodel Building Commands

This part of language definition describes commands for dynamic metamodel 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 generalisation relation between the specified classes. If a generalisation 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 generalisation 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>);

Metamodel Element Scanning Commands

This part of language definition describes commands for scanning metamodel 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 the <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, the <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 only takes into account 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 only takes into account 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 only takes into account 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 the <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, the <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 the <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 the <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, the <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 generalisation 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>**;**

3.4 Implementation of L0+

3.4.1 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) [37, 38, 41, 43].

For the implementation of L0 and L0+, the in-memory data store developed at the IMCS was selected, called mii_rep [44]. This repository proved to be reasonably efficient. For instance, in [42] 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 [39, 40].

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 the creation and deletion of metamodel elements, as well as iteration through them;
- model processing functions that can be divided into two subcategories:
  - functions for the 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++ [32] 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 optimisation, because all optimisations 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 optimisation [31].

3.4.2 Compilation Schema

L0 and L0+ compilation process consists of four phases:

- Preprocessing phase, when compiler directives (for example, “useMM”) are analysed;
- Lexical analysis phase, when the 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 the C++ program, we define a C++ class L0_Var_2 with operations corresponding to L0 commands. This class has the following operations:

1. bool moveNext();
2. bool isNull();

3. bool setFirstToRoot(const string & className);
4. bool setFirstFrom(const L0_Var_2 & rhs , const string & assocName);
5. 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 Repository API.

3.4.3 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, the L0 compiler can generate code in debugging mode. When a program generated in this mode runs, it logs the 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, the generated C++ code is appended with some code fragments logging the 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 the 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();
```

".

3.4.4 Implementation Efficiency

Initially, there were 2 main uses cases for L0 language: a base language in the process of implementation of higher-level languages (by bootstrapping), and a model transformation language for specifying the correspondences between domain model and user interface model in the context of a model-based graphical tool building platform. Therefore, one of the most practical ways to ascertain the effectiveness of L0 implementation is to look at how the language has performed in these fundamentally important practical applications. These use cases are described in more detail in CHAPTER 7. Here we will only note that both these use cases have been approbated in practice and approbation results for both these use cases have been successful.

Using L0 as a base language, a high-level MOLA compiler was built through a bootstrapping scheme [52]. The obtained results confirmed the advantages of the bootstrapping approach – the development of the compiler required reasonable resource investments and the developed compiler generates sufficiently optimal (from the performance point of view) code. Compared to the previous MOLA implementation, it was possible to achieve significant improvements in execution speed (in some examples up to 65x order [52]).

In the context of a practically implemented GrTP platform [18], L0 language was used to specify transformations. In tools built with this platform, an L0 transformation is invoked whenever there is a user action that changes the state of the domain model. In the context of this platform, quite high demands are placed on the execution speed of transformations, as they must be performed so quickly that the user does not notice a delay in the operation of the tool. The L0 language fully satisfied these requirements. This is confirmed, for example, by the fact that in a modeling tool built with the GrTP platform, it takes less than 5 seconds to paste a collection of 1000 elements into a diagram consisting of 1000 elements.

These practical applications confirm the effectiveness of L0 implementation.

3.5 Conclusions

This chapter was devoted to the problems of effective implementation of model transformation languages. It was stated that the direct implementation of transformation languages is a difficult and costly process that does not guarantee the effectiveness of the obtained implementation. It is believed that a more promising way to implement high-level model transformation language is to use bootstrapping [1] (see also 7.2). In turn, bootstrapping is not possible without an effective base language.

One of the main results of this chapter is the definition of a new low-level model transformation language L0+ [2], which can be used as a base language in the bootstrapping process. L0 is called a base language, because:

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

One more reason justifying L0+ usage in the 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 the implementation of a high-level language to another target environment (that uses L0+ as a target language), it is sufficient to only port the implementation of L0+.

The second notable result of this chapter is the principles of the effective implementation of this language. According to these principles, the 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 of the constructs (mainly, patterns) found in more advanced languages. Nevertheless, this language is also used for the direct development of model transformations.

Finally, it should be noted that L0 satisfies all the requirements that were set at the beginning of the chapter:

1. It is very simple – approval for the fulfilment of this requirement is given in CHAPTER 4 Related Works, where we compare L0 with other low-level transformation languages.
2. There is effective implementation for this language – approval for the fulfilment of this requirement is given in section 3.4.4.
3. This language can be used for the practical development of transformations, and not only as the target language in the bootstrapping process – confirmation of this requirement is given in CHAPTER 7.

CHAPTER 4

Related Works

4.1 Introduction

The family of model transformation languages is quite broad [57]. The biggest part of model transformation languages is languages that contain relatively high-level constructs (a typical example are declarative facilities for pattern specification). Alongside these high-level languages, low-level model transformation languages exist as well. In this chapter we review low-level model transformation languages that are similar to L0 language. A characteristic feature that is shared by all these languages is the fact that they are used as target languages in the process of compilation of higher level model transformation languages.

4.2 ATL Bytecode

Concerning the time of publication appearance, the main concurrent of L0 language is ATL bytecode that is used in the implementation of ATL language. The first paper describing ATL Virtual Machine and its bytecode was presented in 2006 [58], but the first paper describing L0 language was presented in 2008 [1, 2]. This is why most attention in this review will be given to ATL bytecode analysis.

A precise definition of ATL byte code instructions is given in [59, 60]. For the convenience of the reader we give a concise summary of ATL byte code instructions, presented in [60], in section 4.2.1. Then in section 4.2.2 we give a comparison of L0 instructions with ATL byte code instructions. This comparison is made by compiling L0 commands to equivalent sets of ATL byte code instructions. Finally we provide a conclusion.

4.2.1 Short Description of ATL Bytecode Instruction Set

1. Stack handling instructions	
1.1 The push instruction	
Format	push s
Operand stack	… ⇒ …, s
Description	Push the string constant s onto the operand stack.
1.2 The pushi instruction	
Format	pushi i
Operand stack	… ⇒ …, i
Description	Push the int constant i onto the operand stack.
1.3 The pushd instruction	
Format	pushd D
Operand stack	… ⇒ …, d
Description	Push the double constant d onto the operand stack.
1.4 The pusht instruction	
Format	pusht
Operand stack	… ⇒ …, true

Description	Push the true boolean constant onto the operand stack.
1.5 The pushf instruction	
Format	Pushf
Operand stack	… ⇒ …, false
Description	Push the false boolean constant onto the operand stack.
1.6 The pop instruction	
Format	pop
Operand stack	…, value ⇒ …
Description	Pop the top value from the operand stack.
1.7 The store instruction	
Format	store variableref
Operand stack	…, value ⇒ …
Description	The value on the top of the operand stack is popped from the operand stack, and the local variable referred by variableref is set to value.
1.8 The load instruction	
Format	load variableref
Operand stack	… ⇒ …, value
Description	The value of the local variable referred by variableref is pushed onto the operand stack.

1.9 The swap instruction	
Format	Swap
Operand stack	…, value2, value1 ⇒ …, value1, value2
Description	Swap the two top values on the operand stack.
1.10 The dup instruction	
Format	dup
Operand stack	…, value ⇒ …, value, value
Description	Duplicate the top value on the operand stack and push the duplicated value onto the operand stack.
1.11 The dup_x1 instruction	
Format	dup_x1
Operand stack	…, value2, value1 ⇒ …, value1, value2, value1
Description	Duplicate the top value on the operand stack and insert the duplicated value two values down in the operand stack.
2. Control Instructions	
2.1 The if instruction	
Format	if offset
Operand stack	…, b ⇒ …
Description	Branch if boolean value b is true.

2.2 The goto instruction	
Format	goto offset
Operand stack	No change.
Description	Branch always.
2.3 The iterate instruction	
Format	iterate
Operand stack	…, collection ⇒ …
Description	Delimitate the beginning of iteration on collection elements.
2.4 The enditerate instruction	
Format	enditerate
Operand stack	No change.
Description	Delimitate the end of iteration on collection elements.
2.5 The call instruction	
Format	call signature
Operand stack	…, context, [arg1, [arg2 ...]] ⇒ … when called operation has no return value …, context, [arg1, [arg2 ...]] ⇒ result, when called operation has a return value
Description	Call a method.

3. Model handling instructions	
3.1 The new instruction	
Format	new
Operand stack	…, classifier-name, metamodel-name ⇒ …, reference
Description	Create new object.
3.2 The get instruction	
Format	get name
Operand stack	…, reference ⇒ …, value
Description	Fetch field from object.
3.3 The set instruction	
Format	set name
Operand stack	…, reference, value ⇒ …
Description	Set field in object.
3.4 The findme instruction	
Format	findme
Operand stack	…, classifier-name, metamodel-name ⇒ …, classifier
Description	Fetch a classifier.

4.2.2 Comparison of L0 Commands by Means of ATL Bytecode

For the purpose of comparison of L0 and ATL bytecode we present a definition of L0 commands by means of ATL bytecode.

3.2.2.1 Object creation

Description	A new object of class UML::Class is created.
L0 code	1 ... 2 pointer c : UML::Class; 3 ... 4 addObj c : UML::Class; 5 ...
ATL byte code	1 ... 2 <constant value="Class"/> 3 <constant value="UML"/> 4 <constant value="c"/> 5 ... 6 <localvariabletable> 7 ... 8 <lve slot="2" name="19" begin="0" end="17"/> 9 ... 10 </localvariabletable> 11 ... 12 <push arg="17"/> <!-- Push string constant "Class"> 13 <push arg="18"/> <!-- Push string constant "UML"> 14 <new/> <!--New object of type UML::Class is created> 15 <store arg="19"/> <!-- Store into local variable c --> 16 ...

3.2.2.1 Setting attribute value

Description	Value of object c attribute name is set to "abc".
L0 code	... pointer c : UML::Class; ... setAttr c.name = "abc"; ...
ATL byte code	... <constant value="c"> <constant value="name"/> <constant value="abc"/> ... <localvariabletable> ... <lve slot="2" name="19" begin="0" end="17"/> ... </localvariabletable> ... <push arg="19"/> <!-- Push var reference c> <push arg="21"/> <!-- Push string constant "abc"> <set arg = "20"/> <!-- Set value of attribute "name"> ...

3.2.2.3 Link creation

Description	A link of type "hasAttribute" between object c and a1is created.
L0 code	.. pointer c : UML::Class; pointer a1 : UML::Attribute; ..

	addLink c.hasAttr.a1;
ATL byte code	... <constant value="c"> <!-- 19 > <constant value="name"/> <constant value="hasAttr"/> <constant value="a1"> ... <localvariabletable> ... <lve slot="2" name="19" begin="0" end="17"/> <lve slot="3" name="22" begin="0" end="17"/> ... </localvariabletable> ... <push arg="19"/> <!-- Push var reference c> <push arg="22"/> <!-- Push var reference a1 > <set arg = "21"/> <!-- Create link of type "hasAttr"> ...

3.2.2.4 Class objects traversal

Description	Traverse all objects of class UML::Class.
L0 code	.. pointer c : UML::Class; .. first c : UML::Class else done; label next_c; ... //process current c here; next c else done;

	goto next_c; label done;

ATL byte code	...
	<constant value="UML"/> <!--17>
	<constant value="Class"/>
	<constant value="c"> <!-- 19>
	<constant value="MMOF!Classifier;.allInstancesFrom(S):QJ"/>
	...
	<localvariabletable>
	...
	<lve slot="2" name="19" begin="0" end="17"/>
	...
	</localvariabletable>
	...
	<push arg="17"/> <!-- Push string constant "Class">
	<push arg="18"/> <!-- Push string constant "UML">
	<call arg = "20"> <!- get all object of this class>
	<iterate/>
	...
	//on each iteration, element to be processed is on the top of the //stack;
	//process your elements here;
	...
	<enditerate/>
	...

Let's take a closer look at object creation (first example). In this example an object of class *UML::Class* is created and reference to this newly created object is stored in a pointer named *c*. In the case of L0 this code fragment is rather self-explanatory and laconic. At first we define a pointer, that will hold a reference to a newly created object of class *UML::Class* (line 2), then we create an object of type *UML::Class* and store the reference to this object

in pointer *c* (line 4). In the case of ATL bytecode the same example will be much more verbose.

In lines 3, 4, 5 – necessary records of symbol table are defined (it is assumed that indexes of a newly created symbol table records start with 17). In line 8 we define a local variable named *c* (an argument *name* of *lve* command). In lines 12, 13 we put strings "Class" and "UML" into the operand stack. After that the *new* command will take these two (last inserted) operands and will handle them as a class and metamodel identifier, respectively. The *new* command will store a reference to a newly created object of class *UML::Class* in the operand stack as well. Finally, in line 15 we store a reference to a newly created object in a local variable *c*.

4.2.3 Conclusions

As can be seen from the examples, ATL bytecode is a lower level language compared to L0. One can easily note that on average one line of L0 code is represented with 3-4 lines of ATL bytecode. It can be said, that ATL is closer to an assembler lever and L0 is closer to a procedural high-level programming language (as a C for instance). There are several reasons for it:

- in the case of ATL byte code, the programmer does not have a possibility to work with local variables in a direct way – command parameters are passed through the operand stack and programmer has to "take care" of the state of this stack. In the case of L0 the language programmer can operate with local variables in a direct way
- one more problem is the necessity to work with symbolic constants by using a symbol table. It actually means, that before we can use some string as a command argument we have to define a record in the symbol table and pass an index of a newly created symbol table record to that command
- the last problem that we emphasise is a necessity to write ATL byte code programs in XML syntax. Undoubtedly it requires additional work on a programmer's part. Alternative (non – XML) syntax for ATL byte code is not available. In an indirect way this fact once again confirms that unlike L0, ATL byte code was designed as

a language that will be used as a target language in the compilation process and not as a language that a “normal” programmer will directly write code in.

Program fragments confirming these statements were given above.

4.3 ATC

The next competitor has been developed at approximately the same time as L0. Nevertheless the development of these languages has been carried out independently. The name of this competitor is Atomic Transformation Code – ATC. ATC is an imperative textual low-level model transformation language. In its expressive power, ATC lies somewhere between ATL byte code and L0 language. For instance, this language allows working with local variables in a way that is common for traditional high-level programming languages, unlike ATC byte code that forces a programmer to work with local variables through the operand stack. The basic set of commands is quite similar to that which can be found in L0 or ATL byte code. The authors of ATC divide commands available in this language into the following groups [72]:

- Execution Flow
- Model Transformation
- Query and Pattern Matching
- Arithmetic and Logical Expressions.

Here we will not give a more detailed description of these commands, since it can be found in [72].

Some of the commands available in ATC do not have direct counterparts in L0 (for instance the *foreach* command). ATC is quite a verbose language and the code in this language is both difficult to comprehend and rather cumbersome to write.

To illustrate this we will take a look at a simple example. Let’s assume that we have the procedure *privatizeAttribute(p : Property)*, that assigns the value *VisibilityKind::private* to the attribute *visibility* of the object of type *Property* that is referenced by parameter *p*.

The L0 code that implements this procedure can be seen below:

```
procedure privatizeAttribute( p : Property );
begin;
	p.visibility = "VisibilityKind::private";
end;
```

ATC code that implements the same procedure can be seen below[72]:

```
<atc:AtcTransformation
	xmi:version="2.0"
	[...]
	xmlns:atc="http://boa.opencanarias.com/atc/0.5"
	name="Encapsulation">

<metamodels name="UML2">
	<packagesNsURI>http://mset.opencanarias.com/uml2/1.0.0/UML2
	</packagesNsURI>
</metamodels>

<modelParameters
	name="uml2Model"
	dirKind="inout"
	metamodelId="UML2"/>

main="//@ownedOperations.7">

<ownedOperations
	xsi:type="atc:AtcMapping"
	name="privatizeAttribute">

	<formalParameters
		xsi:type="atc:AtcMappingParameter"
		name="a"
		dirKind="inout"
		typeQualifNm="UML2::Property">
	</formalParameters>


<mBody xsi:type="atc:AtcBlock">
	<atcAtoms
		xsi:type="atc:AtcCreateDataType"
		packageNsURI=http://mset.opencanarias.com/uml2/1.0.0/UML
		dataTypeNm="VisibilityKind"
		sourceString="private"
		targetId="localVar1"/>

	<atcAtoms
		xsi:type="atc:AtcSetStructuralFeature"
		ObjectId="a"
		stFNm="visibility"
		featureVarId="localVar1"/>
</mBody>

</ownedOperations>
<ownedOperations [...]
```

A detailed explanation of this example is given by the creators of ATC language in [72]. For the convenience of the reader we will quote the explanation here: “The example consists of the complete definition of small mapping, where an enumerated type is generated and assigned to a UML2 attribute of a Property instance, which represents the

contextual parameter for the mapping call. To keep things clean we have omitted the Operational Mappings' trace information that stores bindings created between model objects during execution, and that can be queried later on during the same transformation. This information is embedded explicitly in the ATC transformation instances during compilation. Metamodels are defined outside the transformation block. A metamodel is made up of a list of URIs. They refer to Ecore packages in our case. In this example only the URI of our UML 2 metamodel is present. Model parameters for the transformation follow. We can identify the main operation as being the operation number 7. Finally a full list with the transformation functional operations follows. Only *privatizeAttribute* is shown here. Its type is AtcMapping. During compilation, the contextual parameter has lost the privileged position it held in Operational Mappings to become an ordinary parameter, the first on the list. The visibility assignment, which spans a line in Operational Mappings, has ended up being a Block containing two ATC atoms. None of them act as a container for other atoms. The first atom obtains a private instance of the VisibilityKind enumeration type. The second atom will assign it to the visibility attribute of the Property instance, whose name identifier is 'a'. The 'localVar1' variable identifier is used as a key in a map of Java variables in order to store its associated value. It is the link established between both atoms. The behaviour of AtcSetStructuralFeature is straightforward."

It is quite obvious that the ATC program shown above is much more difficult to comprehend and it is substantially more laborious to write programs in this language. The main reason why ATC notation is so verbose is the fact that ATC was designed without really thinking about the readability of the code or programmer's comfort, but about the efficiency of VM that will execute programs written in this language. This kind of approach can be explained by the positioning of ATC as a low-level language that will be used as a target language in the process of compilation of higher level model transformation languages (for instance QVT [73]). The introduction of this kind of intermediate language is needed to hide the details of a specific model repository.

4.4 Epsilon Object Language – EOL

One more language similar to L0, positioned as a base language, is Epsilon Object Language (EOL). The Epsilon language family [76] is built on the basis of EOL [63]. With the introduction of this family of languages its authors try to solve a rather interesting problem. The authors note that in the context of the MDA process developers have to perform different kinds of operations on models: migration, transformation, validation, comparison and so on. These operations can be performed in two different ways: to use one and the same generic model transformation language for all the kinds of operations, or to use the so-called task-specific languages for each individual kind of operation. In the case of generic model transformation language, it is very likely that we will run into problems, because generic model transformation languages are not equally well suited for different kinds of operations on models. As a consequence, it is considered that it is more optimal to perform different kinds of operations on models with task-specific languages that are suited exactly for these specific kinds of operations. But even in this case we have to think about the compatibility as well as mutual consistency and interoperability of these task-specific languages. Authors of the Epsilon family of languages solve this problem by proposing the base language – EOL, that contains generic model processing constructs and building task specific languages by extending the base language [75].

At the given point Epsilon contains the following languages:

- Epsilon Object Language (EOL)[63]
- Epsilon Validation Language (EVL)[66]
- Epsilon Transformation Language (ETL)[69]
- Epsilon Comparison Language (ECL)[64]
- Epsilon Merging Language (EML)[65]
- Epsilon Wizard Language (EWL)[67]
- Epsilon Generation Language (EGL)[68]

Here we will take a closer look at EOL language, as it is the **base** language of the Epsilon family. A detailed and up-to-date description of other languages of the Epsilon family can be found in [76].

EOL is textual model transformation language. EOL programs consist of modules. Modules consist of body and operations. The body in turn is a sequence of statements. [76].

4.4.1 EOL Types

EOL language has the following built-in types:

- super type of all the types – *Any*
- OCL primitive types: *Real*, *Integer*, *String*, *Boolean*
- collection types – *Sequence*, *Bag*, *OrderedSet*, *Set*
- model element types.

Built-in operations are defined for all this types as well. For instance, for type *Any* following operations are defined:

- *isDefined() : Boolean*
- *isKindOf(type : Type) : Boolean*
- *isTypeOf(type : Type) : Boolean*
- *type() : Type*
- *asString() : String*
- e.t.c.

In total, the built-in data type *Any* has 21 predefined operations. Similarly, predefined operations are available for built-in primitive types (available operations are quite close to those found in operations available for primitive types in traditional programming languages).

Available collection operations in EOL closely resemble collection features found in OCL [17]. For example, for every collection type in EOL the following operations are defined:

- add(item : Any)

- remove(item : Any)
- includes(item : Any) : Boolean
- flatten() : Collection
- count(item : Any) : Integer
- e.t.c.

First-order logics operations on collections are also supported:

- select(iterator : Type | condition) :Collection
- collect(iterator : Type | expression): Collection
- closure(iterator : Type | expression): Collection
- forAll(iterator : Type | condition) :Boolean
- exists(iterator : Type | condition) :Boolean.

4.4.2 EOL Statements

EOL has the following statements:

1. variable declaration statement
2. assignment statement
3. if statement
4. switch statement
5. while statement
6. for statement with break, breakAll, continue statements.

To conclude, we can remark that EOL closely resembles OCL language supplemented with constructs found in imperative programming languages. We do not give a more detailed comparison of L0 and EOL, because it is quite obvious that EOL is substantially higher level language and thus could not satisfy requirements for the base transformation language presented in chapter 2.

4.5 Conclusions

In this chapter, L0 language, developed by the author, was compared with three different low-level model transformation languages – ATL byte code, ATC and EOL.

Historically, the first low-level model transformation language was ATL byte code. A more detailed comparison of this language with L0 language was given in section 4.2.2. This comparison (and the relevant publications) shows that ATL byte code language is not well suited for the direct development of model transformation programs. This conclusion follows from a number of observations highlighted in section 4.2.3. Taking into account the said observations, it is rather obvious that the level of ATL byte code constructs is substantially lower than that of L0. ATL Byte code is only meant as a target language in the process of higher language compilation.

The next reviewed language was ATC. A short description of its main constructs was given in section 4.3. We stress that ATC cannot be used for direct transformation development either. As was the case with ATL byte code, ATC does not have user-friendly concrete syntax (notation used to write down programs in XML form cannot be considered as such). By its nature, ATC is more orientated towards being efficiently interpreted rather than towards being easily usable for transformation development. In comparison with ATL byte code, ATC is more user-friendly, it allows working with local variables without using operand stack and does not force the user to work with symbolic literals through the use of a symbol table.

One more piece of evidence confirming that both these languages are not suited for practical model transformation development follows from the fact that these languages are only used as target languages. In fact, both ATL and ATC authors had to introduce new supplementary code generation languages, to simplify higher level language compiler creation. For instance, ATL compiler to ATL byte code is written not in ATL byte code itself but in a special generation language – ATL Code generation language (ACG) [59, 61].

Authors of ATC language follow the same pattern; they create Generative ATC (GATC) to simplify the creation of the compiler from QVT to ATC, as programming in ATC is too cumbersome [74].

Language that comes closest to L0 language is Epsilon Object Language (EOL) [63]. It should be noted that the development of these languages (from an implementation and paper publication point of view) was independent and more or less simultaneous.
The first paper describing Epsilon Object Language was published in 2006, but it only contained brief enumeration of the main EOL constructs, without further details. The first paper describing L0 and its related family of languages in detail was published in 2008.
Both EOL and L0 are imperative model transformation languages. L0, as well as EOL have concrete textual syntax. The main difference is that EOL provides several rather high-level constructs that are not present in L0. For instance, EOL has the following statements: *if*, *for* and *while*. But the biggest difference is that EOL provides support for OCL-style collections and the according operations.

In conclusion, we can stress that in fact none of the reviewed competitor languages fully satisfy the requirements that were given in section 3.1 and have to be satisfied by a newly developed language L0 (language has to be both easily implementable and user-friendly to a degree that allows the creation of real world model transformation programs in it). Both ATL byte code and ATC are too low-level languages to be used not as a target language, but as an independent transformation language. EOL in turn can definitely be used as an independent model transformation language, but it is very unlikely that it is easily implementable, because of all the high-level constructs it contains (mainly OCL-like collections and the corresponding operations). Similarly it would be rather problematic to use EOL language as a target language in the compilation process. This is because it will be more difficult to port EOL to a different target environment (target programming language + target repository) compared to L0 that does not have these high-level constructs.

The fact that L0 language satisfies the above-mentioned requirements (it is both easy implementable and practically usable in real-world model transformation tasks) is demonstrated in CHAPTER 7, where two important use cases of L0 language are described (usage of L0 language as a base language in the process of high-level model transformation language MOLA implementation by means of bootstrapping method, usage of L0 language as an independent language for development of model transformations in the context of GrTP platform).

CHAPTER 5

Incorporating Undo/Redo in L0

5.1 Introduction

Undo/redo is an important feature in modern software tools. An interesting question that naturally arises regards whether it is possible to incorporate undo/redo features directly into the L0 language (if we port a tool that was built with L0 to some other repository, undo/redo features should still be functioning). In this chapter a new language L0' is created on the basis of L0 language. This new language can be used, for instance, to solve undo/redo problems in the context of GrTP – like platforms [18].

5.2 Basic Idea

We extend language L0 by adding 3 new commands to L0 language (newly obtained language is called L0'[3]):

- *CreateCheckpoint* – establishes a checkpoint;
- *Undo* – rolls back all changes made by a transformation after the last checkpoint was established;
- *Redo* – cancels the previously executed undo.

Before we describe principles of L0` implementation, let us illustrate a context in which a need for this kind of language arises.

Let's consider a model transformation-based graphical tool building platform, e.g., [18]. These kinds of platforms have the following core elements:

- *Hardcoded* presentation (GUI component) metamodel
- Domain metamodel specific to a concrete tool
- *Hardcoded* presentation engine – C++ program that can interpret instances of presentation metamodel, capture user events (also called user actions) and transfer these events to the transformations specific to a concrete tool. These

transformations define the inner logic of this specific tool (ways of how this tool reacts to user actions)

When the tool is started, execution control is passed to the presentation engine (C++ program), which remains active till the moment when the tool is closed. At the moment when the user performs some action, for example, creates a new class if we are talking about class diagram editor, this action is captured by the presentation engine. If the presentation engine cannot process this action by itself it calls transformation that is defined for processing this action. In fact every non-trivial user action is processed by calling a transformation. From the presentation engine, this transformation receives parameters that represent user action. If transformation needs to pass some information to the presentation engine, transformation can write this information into the instance of a presentation metamodel. The presentation engine, in turn, only reads information from a repository that contains instances of presentation and domain metamodels. Further we assume (without loss of generality), that every user action is processed by calling corresponding transformation (called user action processing transformation). It means that presentation engines can only change the state of the repository by means of calling these kind of transformations.

As was previously mentioned, the architecture of the platform requires that every user action is processed by a distinct transformation call. Thanks to this fact, the undo/redo functionality can be added in the following way:

1. add **CreateCheckpoint** command of L0` language before every user action processing transformation call;
2. add undo/redo buttons in a toolbar; these buttons will call transformations consisting of only one command of L0` language – **Undo** and **Redo** respectively. It is assumed (L0` is implemented in such a way), that **CreateCheckpoint** commands create a stack of check points that can be traversed with undo/redo commands.

5.3 Implementation of L0`

As to the implementation of L0`, there are several possibilities. One of the most popular approaches would be to use a data store that already has built-in undo support (EMF for example). The problem with this approach stems from the fact that not every repository provides this kind of built-in undo support. As a consequence, if at some point we start using undo features of one certain repository, we can run into serious problems if we want to migrate to some other repo that does not have this kind of feature.

The author`s proposed idea is to implement L0` by encoding all the information that is needed for execution of the three above-mentioned L0` commands in the metamodel. It means that we will create a compiler from L0` to L0, that will be able to create the corresponding L0 program P for every L0` program P`. In P, all undo specific commands will be substituted with a sequence of corresponding L0 commands.

Let’s describe the main ideas of this compiler. To be able to cancel changes made by transformation it is necessary to store information about changes in model (it is assumed that undo/redo will only be available for model-level actions). In model transformation language L0, the state of the model corresponding to some metamodel M can be changed by the following actions:

- creation/deletion of link
- creation/deletion of object
- attribute value changing.

To store information about these actions, we supplement metamodel M with a fragment seen in Fig. 13. Without loss of generality we can assume that all classes present in a user

metamodel are subclasses of the class **Object**. In this case it should be noted that this fragment does not depend on a specific metamodel.

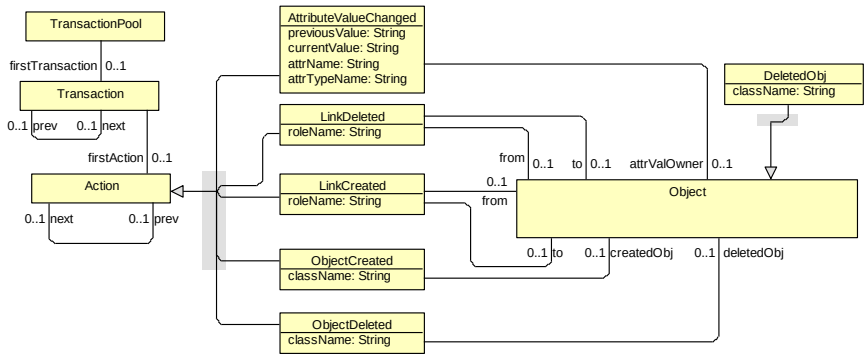

Fig. 13 Classes for undo/redo implementation.

Let's describe what new elements are added to user metamodel and what the purpose of each element is:

- **TransactionPool** – contains a pointer to a first transactions list element
- **Transaction** – contains a pointer to a first element of a list of transaction actions
- **Action** – abstract super class for actions that can be part of the transaction
- **AttributeValueChanged** – stores information about a certain object (association "attrValOwner") certain attribute value change
- **LinkDeleted** – stores information about the deleted link of a certain type (attribute "roleName") between given objects (association "from", association "to")
- **LinkCreated** – stores information about created link
- **ObjectCreated** – stores information about created object
- **Object** – abstract super class for pointers to created and deleted objects (all classes in the user metamodel are subclasses of this class)
- **DeletedObject** – stores information about deleted objects

When creating the L0` compiler, special attention has to be paid to the compilation of L0 commands that change the state of the model. These commands are the following:

- addLink
- deleteLink
- addObj

- deleteObj
- setAttr.

A compilation schema of L0` to L0 is sketched in Table 1.

Table 1. Schema of L0` compilation to L0

L0` command		Description of a corresponding L0 code
CreateCheckPoint		Object of the class “Transaction” is created and is appended (via “next/prev” association) to the list of already existing transactions.
L0 commands needing special processi	addObj <pointerName>: <ClassName>	Object o of the type <ClassName> is instantiated. Object a of the type “ObjectCreated” is instantiated. Link of type “createdObj” between objects a and o is instantiated. Object a is appended to the list of already existing actions inside the current transaction.
	deleteObj <pointerName>	Register attribute value changes. Delete instances of user defined associations in which this object takes part (association “deleteLink”). Create object do of the type “DeletedObject”. Create object a of the type “ObjectDeleted”. Repoint all instances of “undo specific” associations starting from object to be deleted to object do. Append object a to the list of already existing actions inside the current transaction.
	addLink <pointer1>. <assocname>. <pointer2>	Link of the type <assocname> between <pointer1> and <pointer2> is instantiated. Object a of the type “LinkCreated” is instantiated. Link of the type “fromObj” between a and object pointed to by a <pointer1> is instantiated. Link of the type “toObj” between a and object pointed to by a <pointer2> is instantiated. Object a is appended to the list of already existing actions inside the current transaction.
	deleteLink <pointer1>. <assocname>. <pointer2>	Object a of the type “LinkDeleted” is instantiated. Link of the type fromObj between a and object pointed to by a <pointer1> is instantiated. Link of the type “toObj” between a and object pointed to by <pointer2> is instantiated. Link of the type <assocName> between objects pointed to by <pointer1> and <pointer2> respectively is deleted. Object a is appended to the list of already existing actions inside the current transaction.

n g	setAttr <pointer>. <attrname> = <expr>	Object a of the type “AttributeValueChanged” is instantiated and populated with according values. Link of the type “attrValOwner” between a and object pointed to by <pointer> is instantiated. Object a is appended to the list of already existing actions inside the current transaction.
Undo		For each item in a list of actions of the current transaction (starting from the last one), perform an inverse action.
Redo		For each item in a list of actions of the current transaction (starting from the first one), perform a specified action.

We proceed by illustrating this schema with a simple example. Let’s recall the oriented graph metamodel, presented earlier (Fig. 14).

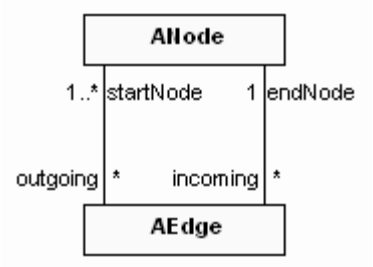

Fig. 14 Oriented graph metamodel

Assume that the following L0' program is being executed:

```
1. createCheckpoint;

2. addObj a : ANode;
3. setAttr a.label = "a";

4. addObj b : ANode;
5. setAttr b.label = "b";

6. addObj c : ANode;
7. setAttr c.label = "c";

8. addObj s : AEdge;
9. setAttr s.label = "s";

10. addObj p : AEdge;
11. setAttr p.label = "p";

12. addLink s.startNode.a;
13. addLink s.endNode.b;
14. addLink p.startNode.b;
15. addLink p.endNode.c;

16. Undo;
```

Before execution of the L0` program, the user metamodel has to be supplemented with a metamodel fragment presented in Fig. 13. After this supplementation, the metamodel of oriented graphs will look like the following (Fig. 15):

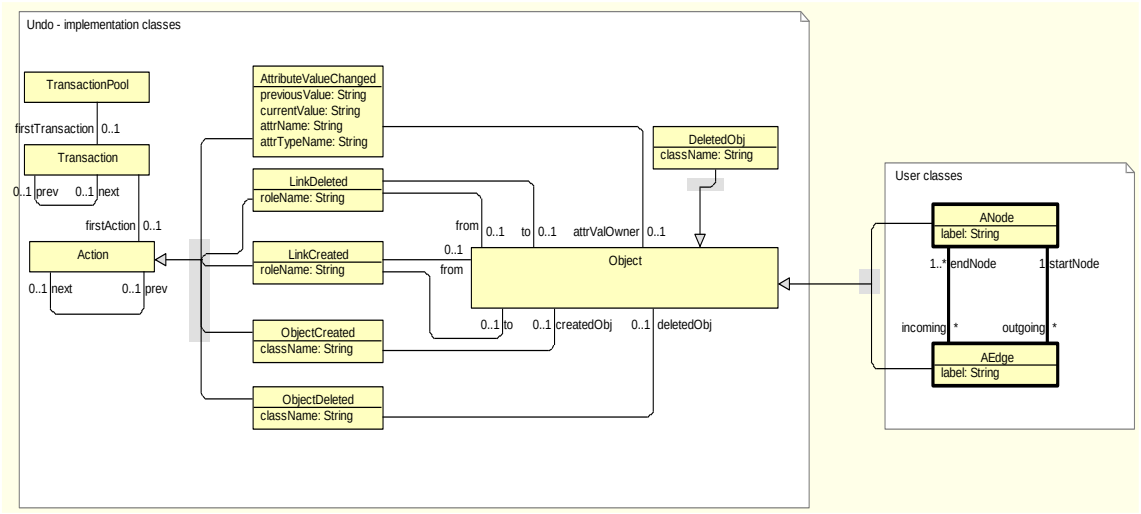

Fig. 15 User MM after addition of undo/redo implementation classes

Now we will illustrate the execution of this program step by step. While demonstrating states of the model we will adhere to the following agreements:

- objects from user part – yellow background with a bold frame
- links from user part – highlighted with bold
- object from undo part – green background without a bold frame
- links and objects, added after the last executed command, are portrayed with a dashed line

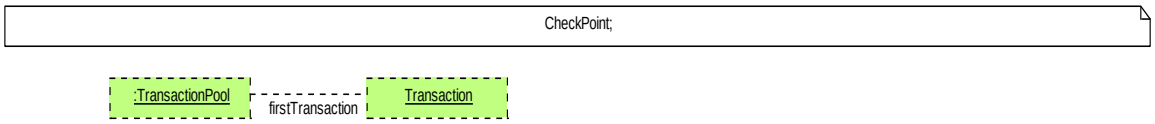

Fig. 16 Model state after execution of step 1

CheckPoint command is executed. According to the compilation schema an object of class *TransactionPool* is connected (via link of type *firstTransaction*) with the newly created object of class *Transaction*.

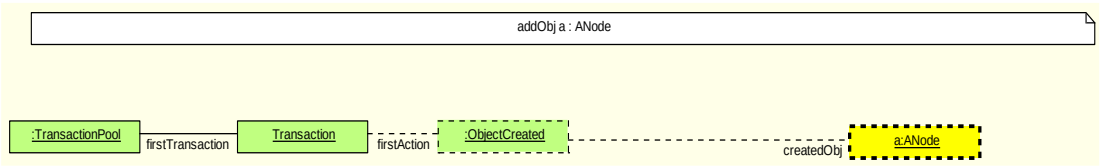

Fig. 17 Model state after execution of step 2

addObj a : ANode; command is executed. Object of class *ANode* is created in a user part, in the undo part information about the creation of a new object is stored – an object of class *ObjectCreated* is created and is linked to a transaction as a first action of it (association *firstAction*).

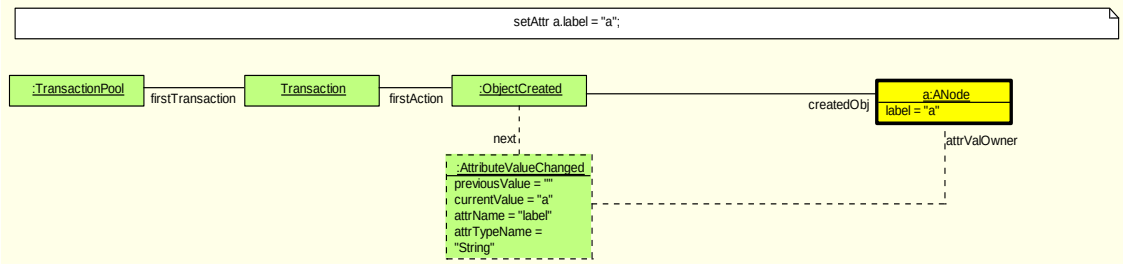

Fig. 18 Model state after execution of step 3

setAttr a.label = "a"; command is executed. In the user part a value of attribute "*label*" of a corresponding object is changed to "a". In the undo part, information about attribute value change is stored – new object of class *AttributeValueChanged* is created and linked to a current transaction (association *next*) and in the values of this object attributes information about values of the attribute *label* before and after *setAttr* command execution is stored.

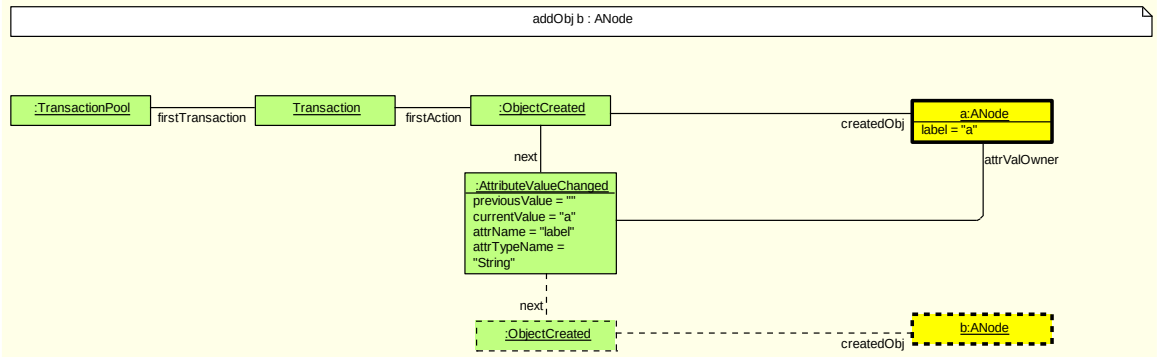

Fig. 19 Model state after execution of step 4

addObj b : ANode; command is executed. The situation is similar to that which happened after step 2.

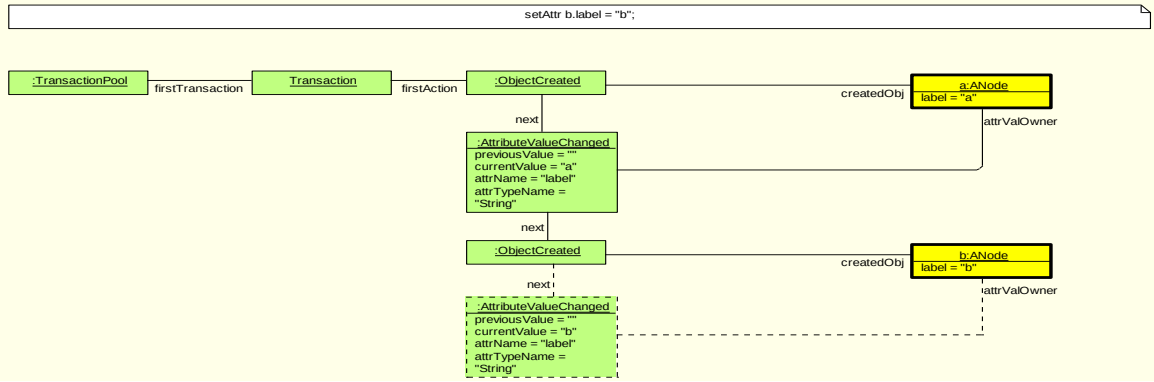

Fig. 20 Model state after execution of step 5

setAttr b.label = "b"; command is executed. The situation is similar to that which happened after step 3.

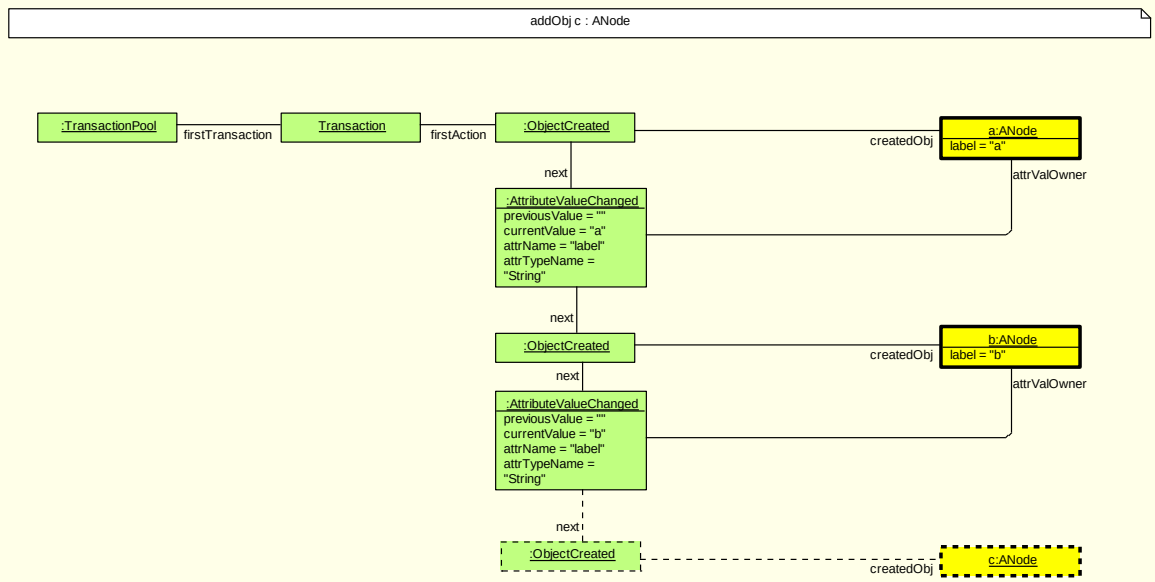

Fig. 21 Model state after execution of step 6

addObj c : ANode; command is executed. The situation is similar to that which happened after step 2.

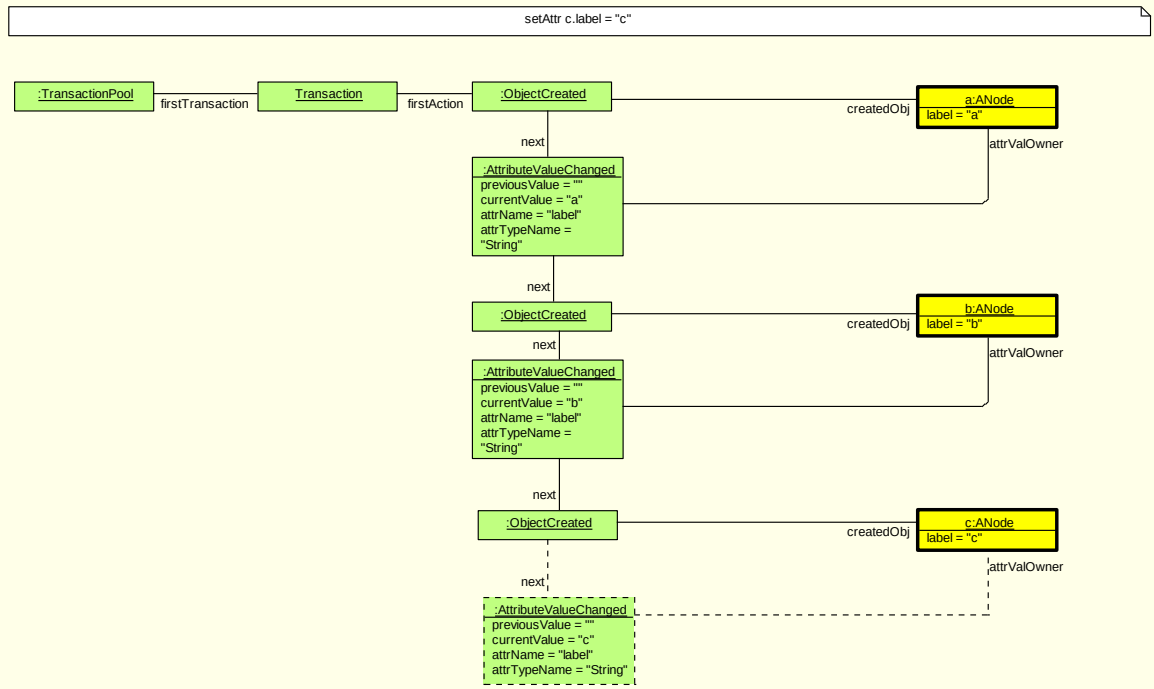

Fig. 22 Model state after execution of step 7

setAttr c.label = “c”; command is executed. The situation is similar to that which happened after step 3.

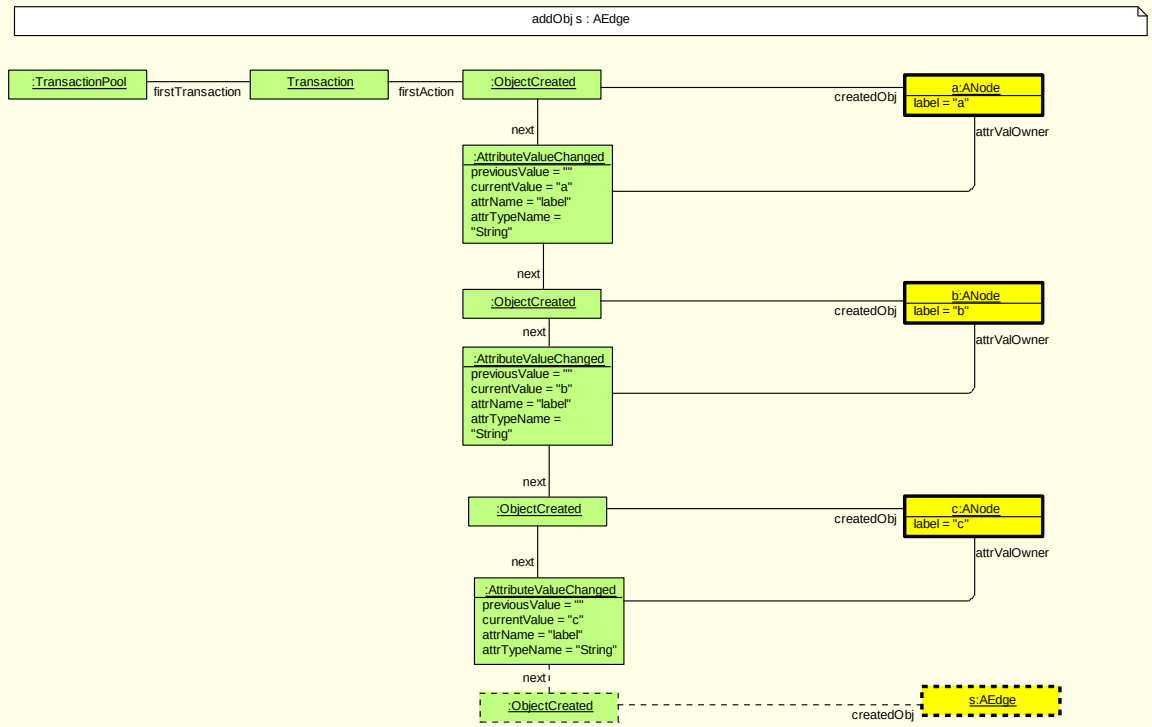

Fig. 23 Model state after execution of step 8

addObj s : AEdge; is executed. The situation is similar to that which happened after step 2.

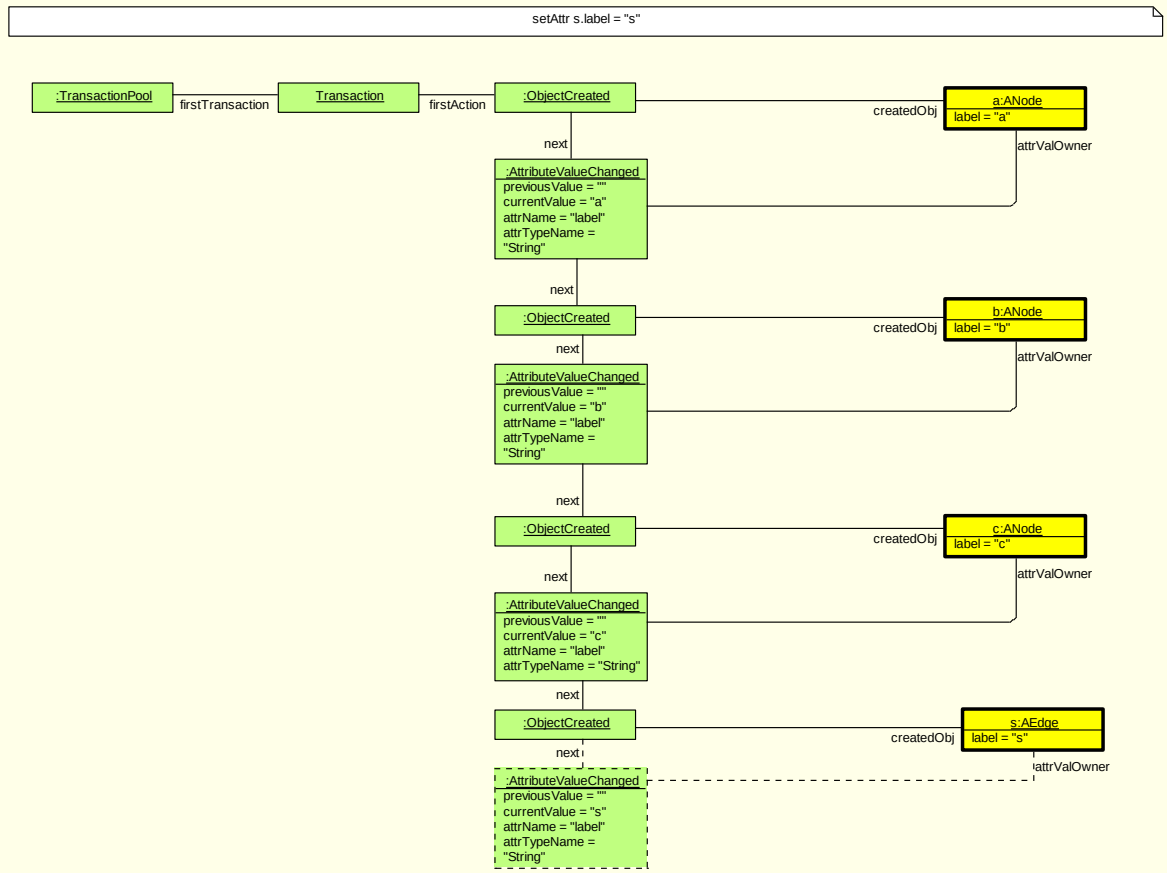

Fig. 24 Model state after execution of step 9

setAttr s.label = “s”; command is executed. The situation is similar to that which happened after step 3.

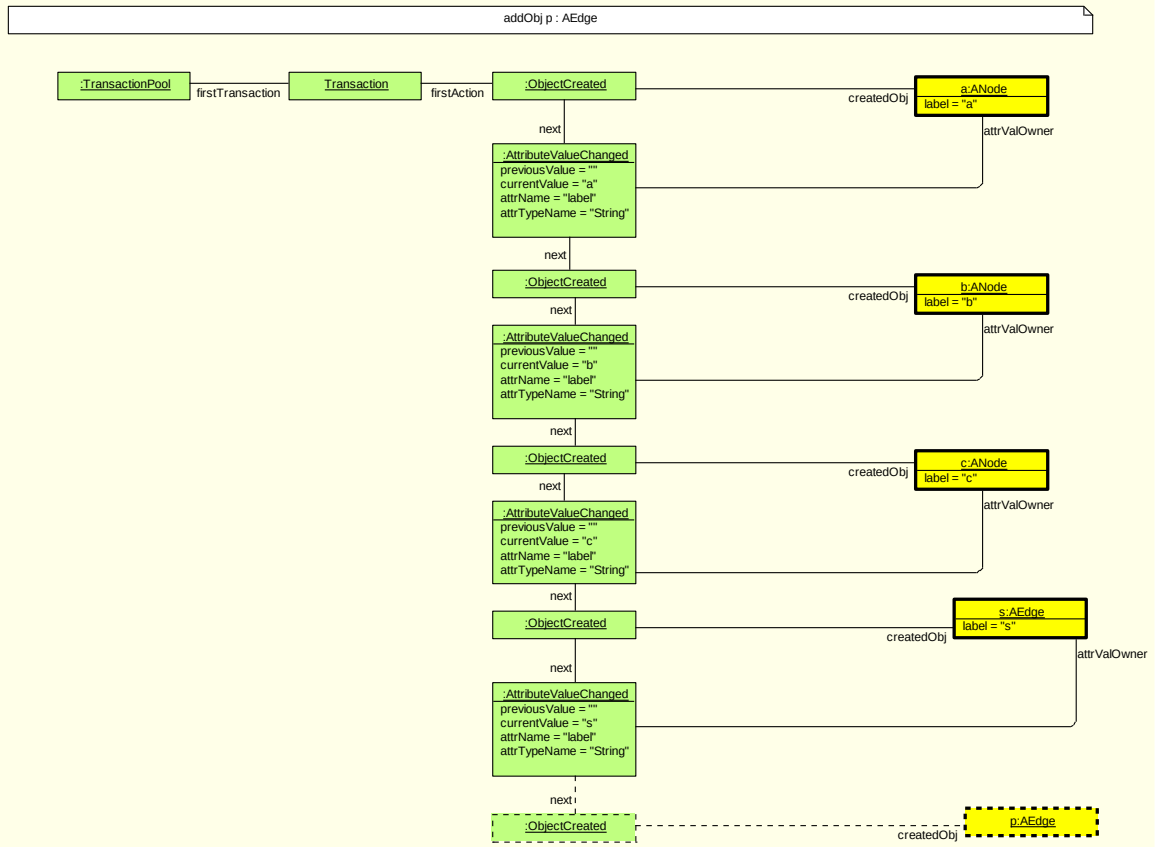

Fig. 25 Model state after execution of step 10

addObj p : AEdge; command is executed. The situation is similar to that which happened after step 2.

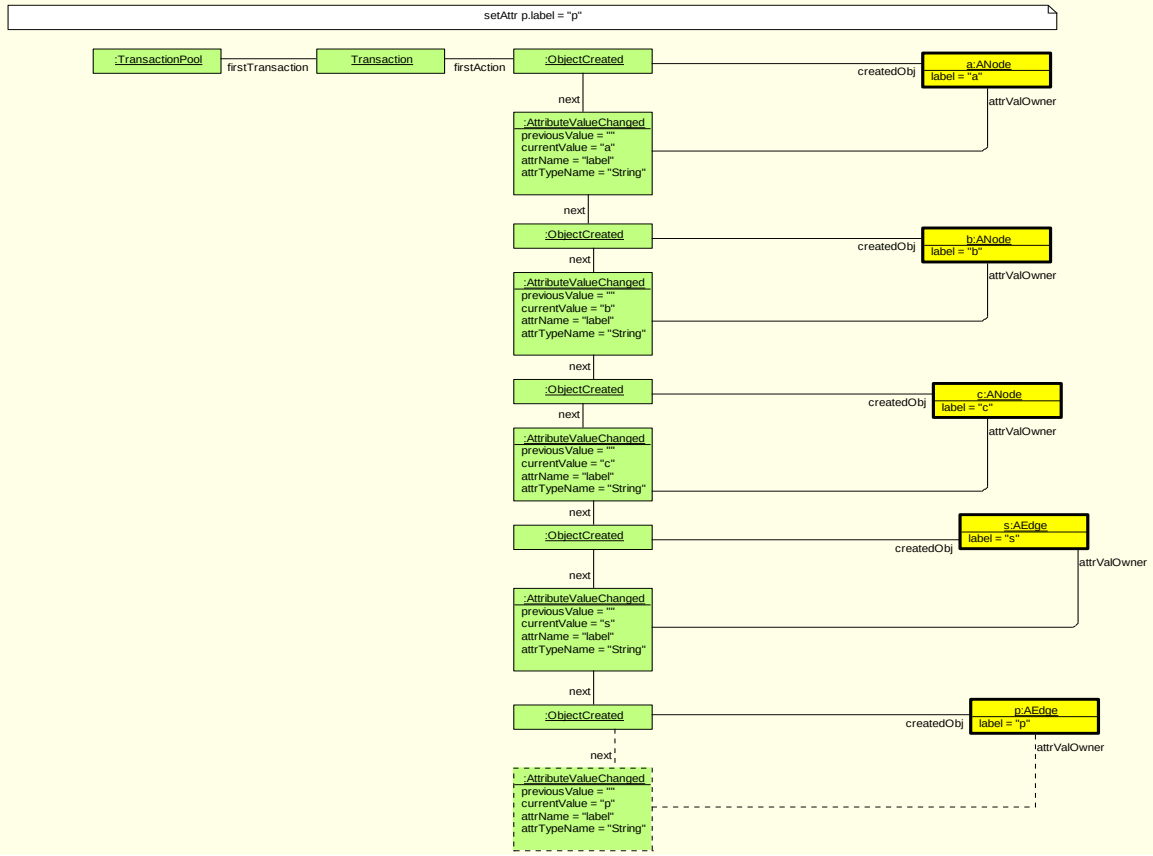

Fig. 26 Model state after execution of step 11

setAttr p.label = "p"; command is executed. The situation is similar to that which happened after step 3.

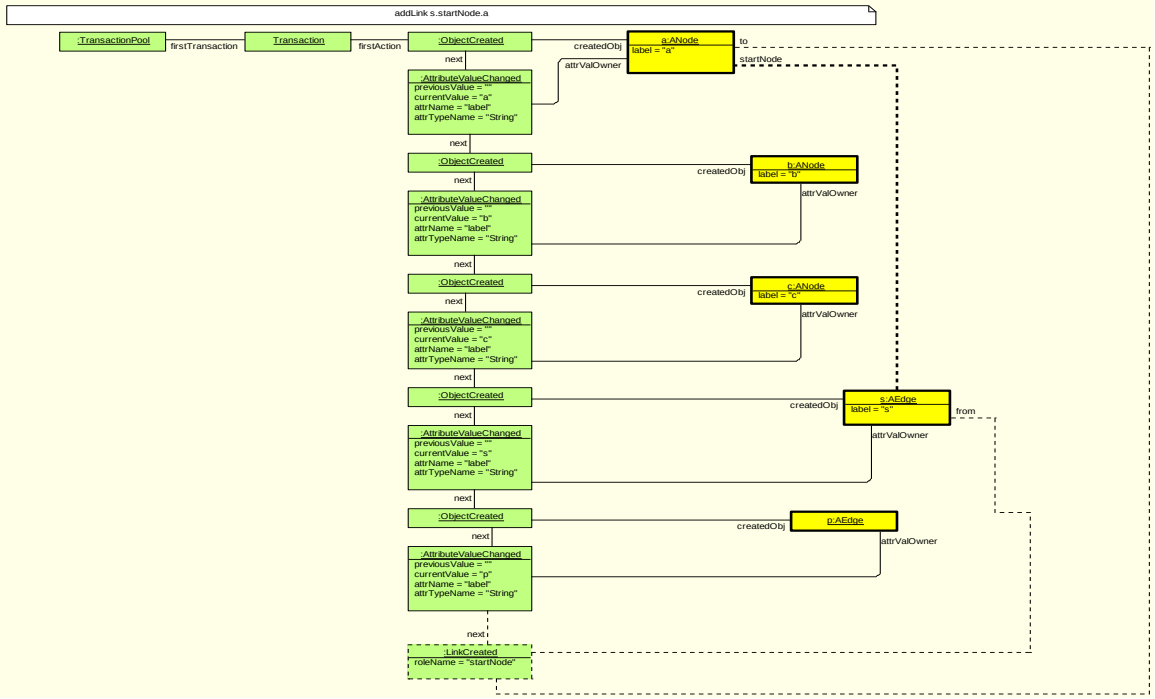

Fig. 27 Model state after execution of step 12

addLink s.startNode.a; command is executed. In the user part the link of type *startNode* between objects *s:AEdge* and *a:ANode* is created. In the undo part the object of class *LinkCreated* is created. This object will hold info about a newly created link. In the attribute *LinkCreated.roleName* of this object, a type of the newly created link is stored. Starting and ending objects of this link are connected with associations *to* and *from* accordingly. Object of class *LinkCreated* is added to the list of transaction actions.

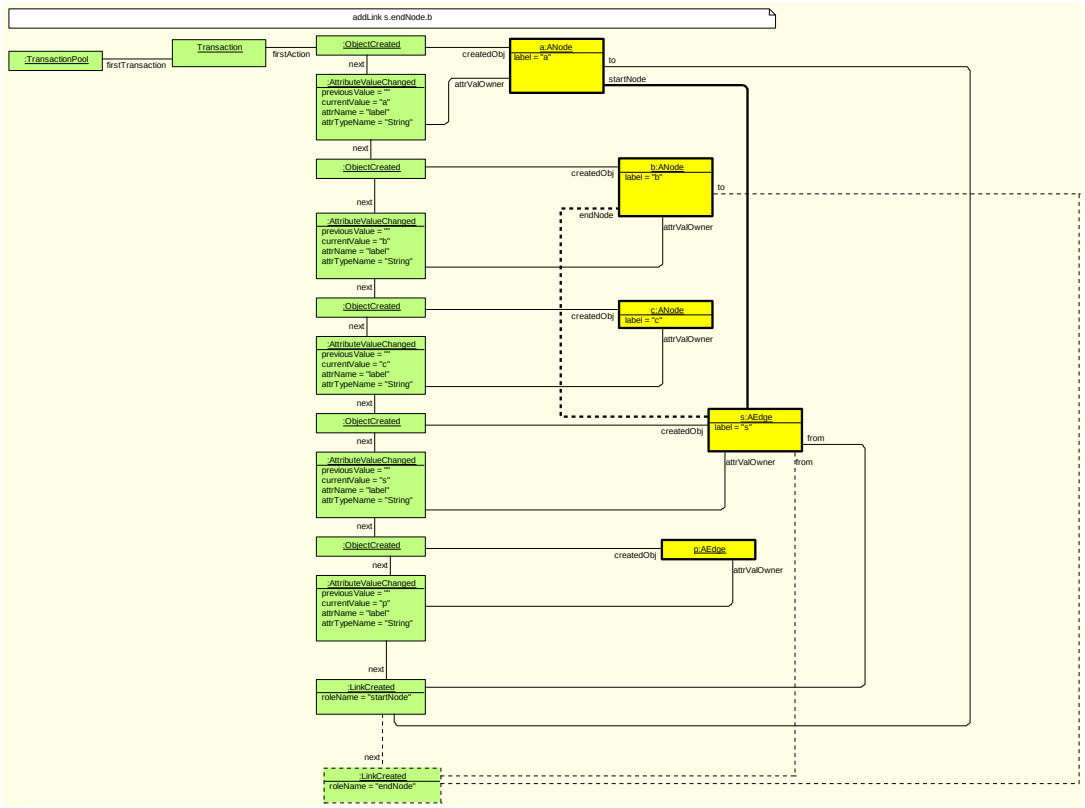

Fig. 28 Model state after execution of step 13

addLink s.endNode.b; command is executed. The situation is similar to that which happened at step 12.

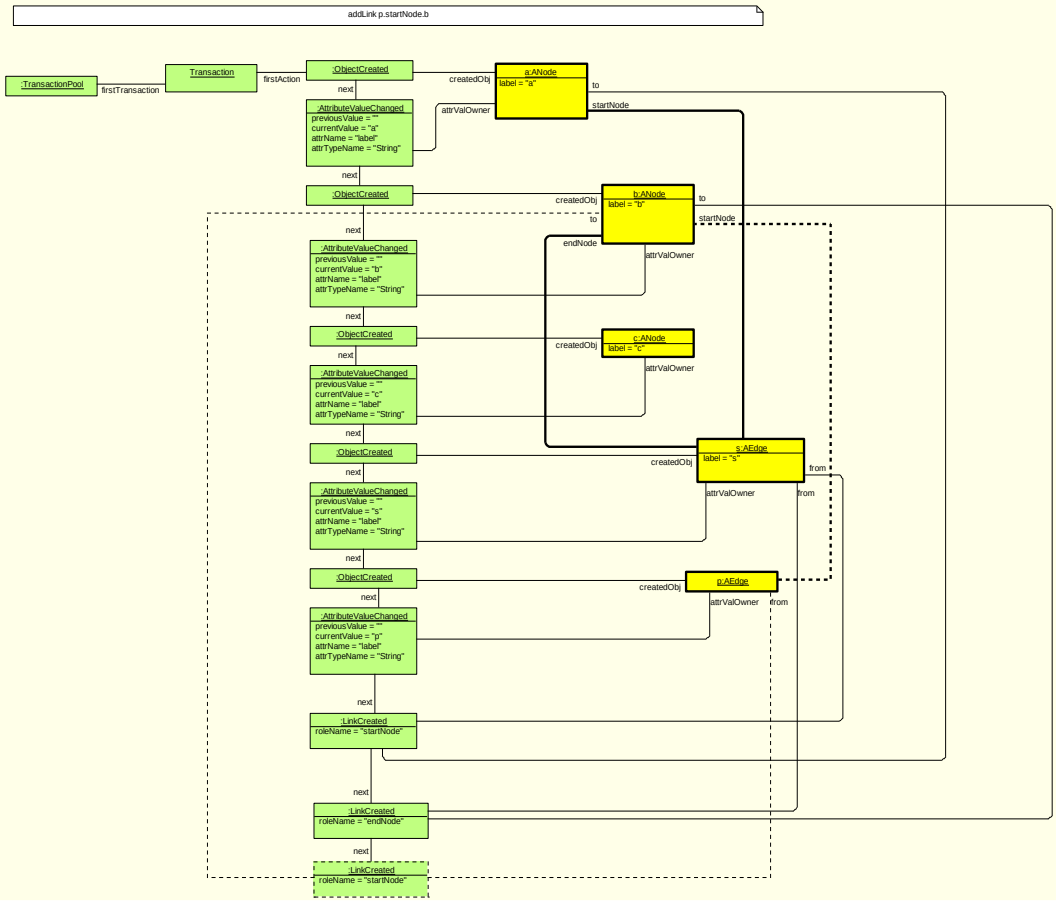

Fig. 29 Model state after execution of step 14

addLink p.startNode.b; command is executed. The situation is similar to that which happened at step 12.

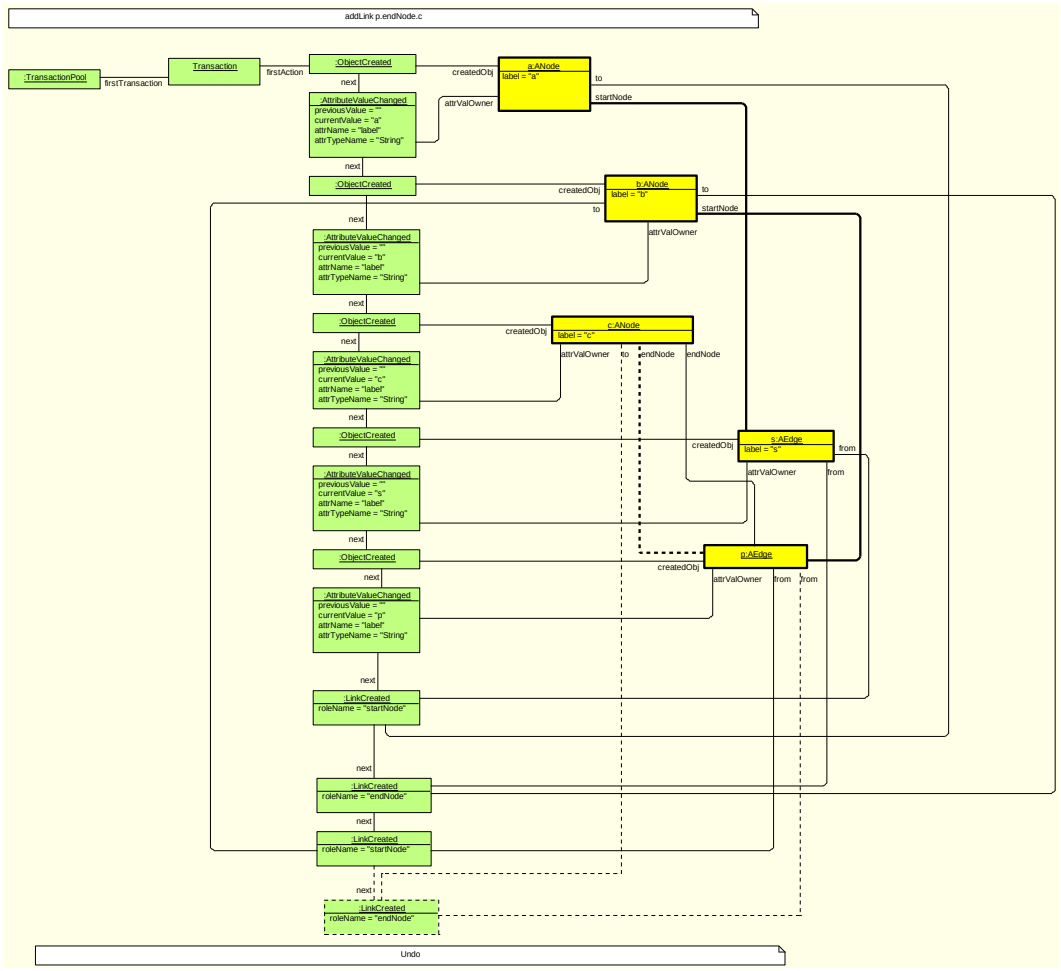

Fig. 30 Model state after execution of step 15

addLink p.endNode.c; command is executed. The situation is similar to that which happened after step 12.

After we have executed **addLink p.endNode.c;** command in step 15, we execute step 16 – **Undo;** command. Semantically it means that we perform an inverse action for each item in a list of action of the current transaction (starting from the last one). For example, in our case the last action of the current transaction is LinkCreated. An inverse action in this case will be the deletion of the link that is specified by the value of the attribute *LinkCreated.roleName* and associations *LinkCreated.to* and *LinkCreated.from*.

5.4 Conclusions

Traditionally the Undo/Redo mechanism is implemented on the repository level. It means that when we migrate our applications to a new repository we will need a new implementation of Undo/Redo. In this chapter a new method for the implementation of Undo/Redo has been proposed. This method does not depend on the features offered by a concrete repository and is based solely on the base transformation language L0. It means that in the process of migration to a new repository we will only have to provide the implementation of the base language L0 and that (as it was shown in section 3.4) is not usually an overly complicated task. After that, Undo/Redo implementation on this new repository will be obtained automatically (with a standard compilation of L0` to L0).

Naturally, a question may arise – why is it necessary to define a new language L0`, if we only want to add some commands to the existing language L0. Why can't we just change the already existing compiler of language L0? In this case, the language L0 serves as a natural abstraction of the target environment, which hides the implementation details of the execution environment from higher level languages. Therefore, when writing a compiler from language L0` to L0, we only need to know the specifics of one target environment (of L0 language itself). If we wanted to add a new feature to the L0 language itself (and wanted to change the L0 compiler), then we would need to know the specifics of all the target languages for which the L0 language is compiled.

The figure below shows the implementation scheme of the L0` language.

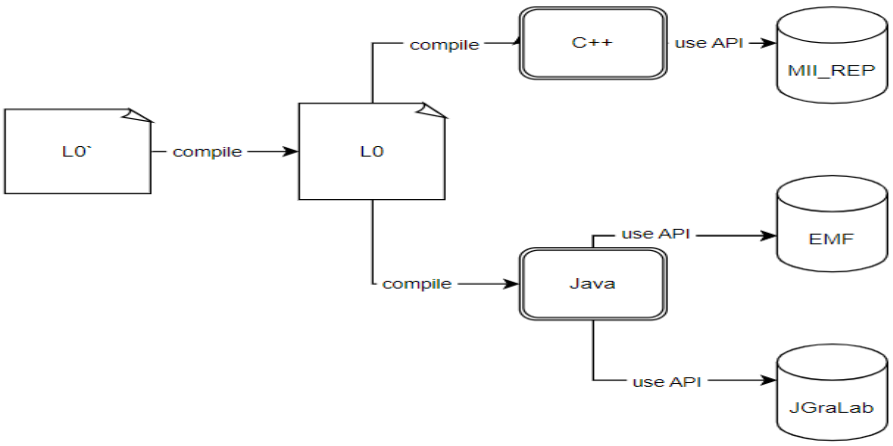

Fig. 31 Implementation scheme for language L0`

This scheme, when new features are not added directly to the language L0, but by defining higher level languages that are further compiled to L0, is universal, and relatively cheap from the point of view of the programming efforts needed to build a compiler, and a relatively efficient mechanism for obtaining the implementation of a new transformation language with an extended range of features (compared to L0). One of the most serious applications of this scheme is the implementation of high-level transformation languages with the bootstrapping method, which is described in section 7.2 Implementations of High-level Model Transformation Language.

Above we have shown the principal scheme for how the language L0 can be supplemented with new features (in this specific case we chose to supplement L0 with undo/redo options). Therefore, when talking about the implementation of undo/redo, we focused on a relatively simple implementation option, which allows one to obtain the working implementation of L0` that would be usable in practice, but ignores some minor technical nuances.

A much more detailed version of undo/redo implementation, which has been implemented and is used in the GrTp platform, is presented by the author of this thesis and co-authors in [7, 8].

There we use a proxy approach to implement undo/redo functionality. Thanks to tighter integration with the GrTp platform, in most cases the platform itself can automatically call an analog of the CreateCheckpoint function and create a sequence of user actions to cancel. In addition, we provide an opportunity to create several sequences of undoable actions and to define dependencies between them. We also provide an API for creating irreversible actions and irreversible states.

CHAPTER 6

A Novel Application of Model Transformations – Method for Migration of Relational DB to RDF

A vast amount of business information is nowadays stored in relational databases. For the Semantic Web vision to become a reality, we need ways of exploiting this data in the form of RDF triples. A universal and commonly accepted solution to this problem still does not exist. In most cases, mapping languages are used for the specification of correspondences between OWL [34] ontology and DB schema. At the same time, these languages are generally not well suited for the specification of mappings in cases when there is a substantial difference between OWL ontology and DB schema. Below, we describe a new model transformation-based method for the specification of correspondences between the elements of DB schema and OWL ontology. We also present our experience of using this method in a real world use case and sketch the direction of future research.

6.1 Introduction

A traditional approach for storing data is in the form of relational databases. This approach has its own advantages and drawbacks. A new approach to data management that is promising to overcome some of the limitations of relational databases is RDF framework [55].

For example, there are several medical statistics databases in Latvia covering the main pathologies endangering the quality of life by the spread of risk factors to residents of Latvia [5]. These databases are relational databases. As a consequence, users of these databases have faced the following problems:

- For a typical medical researcher, it is difficult to understand what data are contained in a database. The reason is the fact that ER schemas usually present data in a form that is not quite understandable for a non-specialist.
- The second problem is a data retrieval problem. For a non-proficient user, it is too difficult to create SQL queries.

Our solution to these problems is to use semantic web technology: relational database schema is transformed to OWL ontology and visualised in UML, which is a format already readable by a medical researcher. The transformed OWL data can be queried through the standard SPARQL [56] query language by a programmer. For medical researchers, we have developed a graphical front-end ViziQuer [4, 54] for composing SPARQL queries directly from UML-like visualisations.

These problems are discussed in more detail in [4, 5]. However, there is an important problem, the solution to which is only briefly sketched in mentioned papers – how to perform the export of relational databases to RDF databases.

Below, we give a review of existing tools for exporting data of relational databases to the RDF database and explain our motivation for developing a new method for this kind of migration. At the time of the study, several methods already existed for the migration of relational data to RDF. A survey of existing approaches is given in the W3C work group report [46]. These methods can be divided into two groups: those providing the direct creation of RDF dumps, and others allowing the defining of views on relational data [47, 48, 49, 50].

The most mature implementations of mapping languages are Virtuoso RDF View [48] and D2RQ [47]. Virtuoso RDF View provides its own Quad map language that allows the defining of RDF views on relational data. A potential drawback of this language is the absence of strict control of target ontology. It is not prohibited to map some relational table to a non-existent element of ontology. This can result in some very obscure bugs. D2RQ provides a declarative language for defining RDF views on relational data. The main strength of mapping languages lies in the specification of mappings for typical situations. However there are situations when the limited expressive power of mapping languages makes the specification of sophisticated mappings rather difficult or even impossible.

The W3C standard R2RML [86] supported by various transformation tools, and an ontology-based data access approach (OBDA) [87, 88] have also been adopted for the specification of mappings between knowledge graphs structured in the form of RDB and RDF.

As for model-based approaches to the migration of relational data to RDF, a quite interesting approach is discussed in [45] where the authors provide a mapping language

that is compiled to a model transformation language. This approach is relatively similar to ours in a certain way. The main difference is that in our case, the transformation from the relational model to the domain model is fixed and cannot be parameterised. In the case of [45], it is exactly the opposite – the authors propose to use some ORM framework to specify the relationship between the relational DB and the object-oriented domain model. Both this object-oriented model and the ORM representation from the relational DB to it must be specified by the user independently. In our case, the object-oriented domain model is obtained automatically from the relational DB. From this point of view, our proposed method looks simpler to use (namely, the specification of representations from relational DB to ontology should be more convenient), but less flexible in the part that concerns import from relational DB (the one used with ORM has the most control over the import process).

It would also be interesting to compare the speed of the two proposed methods. Especially the effect that using an ORM has on the speed of the migration process. Unfortunately, no information can be found in [45] about the use of the proposed approach on a sufficiently large real-life example.

6.2 Model Transformation-based Migration Method on the Basis of the Metamodel-based Data Store

In this section, we describe a model transformation-based approach for migrating relational databases to RDF databases and illustrate it with a simple example developed by the authors of [71]. There are two main differences of the proposed approach compared to the methods mentioned above:

- the proposed method is based on UML/OWL profile;
- correspondences between according elements (group of elements) of database schema and elements (group of elements) of ontology are specified by using a graphical high-level model transformation language.

The conceptual schema of our method is shown in Fig. 32.

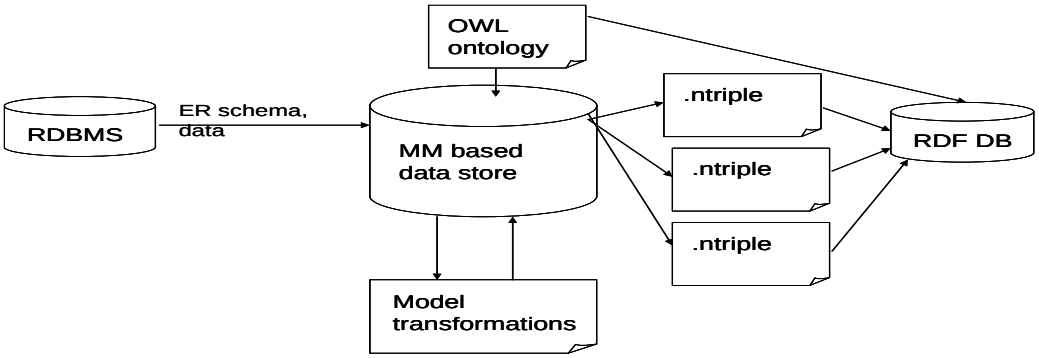

Fig. 32 The conceptual schema of the proposed method

According to this schema, the process of data migration consists of the following steps:

1) We start by importing data and ER schema of the source relational database into a metamodel-based data store (for example [41]).
2) Then, we import target ontology (OWL ontology) into the metamodel-based data store.
3) Then, domain expert specifies correspondences between the elements (group of elements) of the source ER model and according elements of the target ontology. These correspondences are specified in a high-level graphical model transformation language MOLA [26].
4) After the execution of transformations created in the previous step, files containing serialized RDF triples are generated according to rules specified by a domain expert. These files contain the representation of relational data in the form of RDF triples. Serialization of RDF triples is possible in several formats such as RDF/XML, Turtle, N3, N-Quads, RDF/JSON, N-Triples [77, 78, 79, 80, 70, 81]. We chose to serialize them in N-Triples format.
5) Finally, newly created NTriple files are imported into the RDF database.

It should be stressed that all of the steps listed above, excluding only one step – the specification of correspondences between the source ER model and the target ontology, are universal steps – they do not depend on a concrete source ER model or target OWL ontology or correspondences between them. It means that the only step that needs to be taken care of by the user of this method is the specification of correspondences between the source ER model elements and elements of the target OWL ontology.

In the following subsections, we will explain the aforementioned steps in more detail on the basis of a simple example. In this case, it will be a mini-university example.

6.2.1 Universal Steps

6.2.1.1 Importing ER schema and relational data:

1) Import of relational tables – for every table *T* in the source relational model, a class *C* in a metamodel based repository is created. For every record of table *T*, an according object of class *C* is created.
2) Import of table columns – for every column *Col* of a table *T* excluding FK columns and PK columns, an appropriate class attribute *A* is created. Values of attribute *A*, are imported from the values of column *Col*.
3) Import of foreign key relations – for every foreign key relation, an association between appropriate classes is created and populated with instances.

In Fig. 33, we can see a schematic result of import of mini-university DB into a model-based repository.

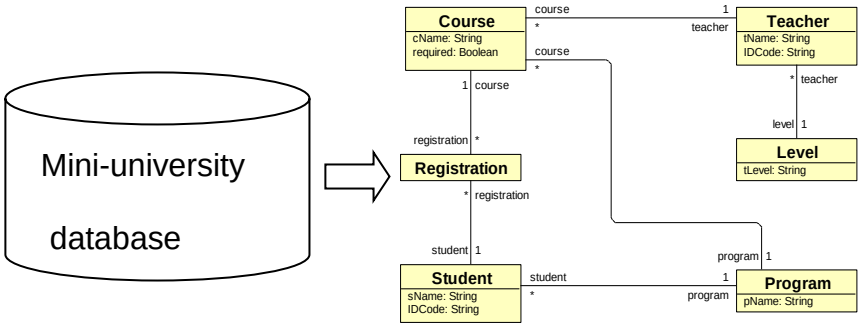

Fig. 33 The result of import of mini-university DB

6.2.1.2 Importing ontology

In general, the import of OWL ontology into a metamodel-based data store can be a rather non-trivial task [51]. However, we confine ourselves to only those ontologies that conform to the UML/OWL subset. The basic idea of this subset is to only use those OWL DL

constructs that can be adequately represented with UML class diagrams [4]. In Fig. 34 we can see the mini-university ontology presented in the UML/OWL subset.

In the case that the ontology conforms to a UML/OWL subset, we can derive a fairly simple set of rules for importing the ontology into a metamodel based data store:

1) For every <<owlClass>> axiom, an appropriate class is created in a metamodel-based data store
2) For every <<rdfsSubClassOf>> axiom, a generalisation relationship between appropriate classes is created
3) For every <<owlProperty>> axiom along with sub-stereotype <<objectProperty>>, an association between appropriate classes is created
4) For every <<owlProperty>> along with sub-stereotype <<datatypeProperty>>, a class attribute of appropriate type is created (at the given point, we only support four elementary types – real, boolean, string, integer).

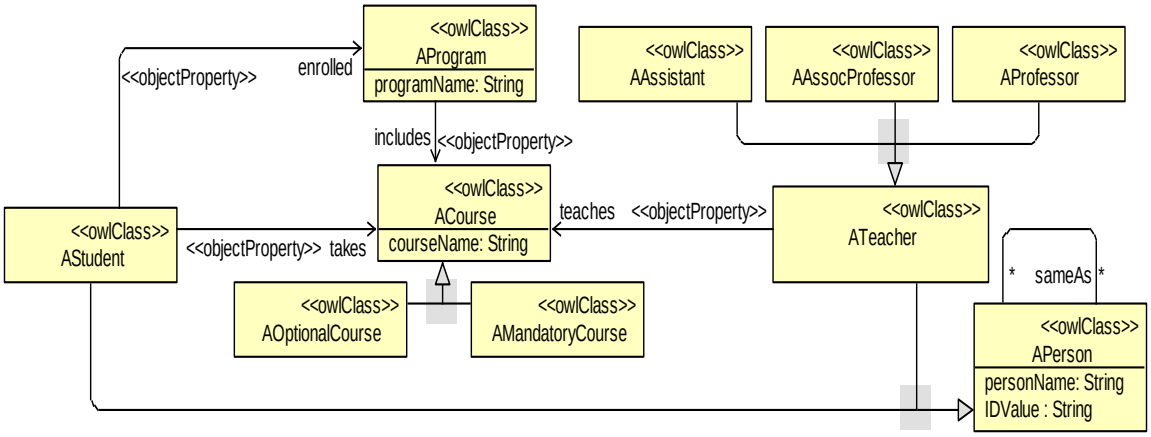

Fig. 34 The imported ontology

6.2.1.3 Exporting RDF Triples

The last step of the migration process is the generation of NTriples from instances contained in a metamodel-based data store:

1. For every object of every class, we generate triples in the form <ObjectURI> <rdf:Type> <ClassURI>.

1.1. For instance, a triple <http://lumii.lv/ex#Student1> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://lumii.lv/ex#Student> is generated for the object of class “Student”.

2. For every link between objects, we generate triples in the form <ObjectURI1> <ObjPropURI> <ObjectURI2>.
   2.1. For instance, a triple <http://lumii.lv/ex#Student1> <http://lumii.lv/ex#takes> <http://lumii.lv/ex#Course2> is generated to represent an instance of association “takes” between objects of classes “Student” and “Course”.
3. For every attribute value present in a data store, we generate triples in the form <ObjectURI> <DataTypeURI> <AttributeValSpec>.
   3.1. For instance, a triple <http://lumii.lv/ex#Student1> <http://lumii.lv/ex#personName> "Dave"^^<http://www.w3.org/2001/XMLSchema#string> is generated to represent the fact that the value of the attribute “personName” of object “Student1” is “Dave”.

6.2.2 Domain – Specific Step

When the relational schema, relational data corresponding to this schema and the ontology have been imported into the data store, we can start the specification of correspondences between elements of ER schema and ontology. For a definition of these correspondences a subset of a graphical high-level model transformation language MOLA, called A-MOLA is used. MOLA program is then compiled to the L0 program. So all transformations are performed by L0 code.

A-MOLA program specifying transformation consists of consecutive *foreach* loops. The body of each loop contains exactly one rule. Logically transformation can be divided into three steps:

1. Class mappings. Each class from the ER model is mapped to an ontology class.
2. Association mappings. Each association from ER model is mapped to an ontology association.
3. Many-to-many association mappings. Finally, each many-to-many association is processed.

Specification of model transformation starts with a metamodel. In Fig. 35, we can see a metamodel for the mini-university example containing imported ontology and ER schema.

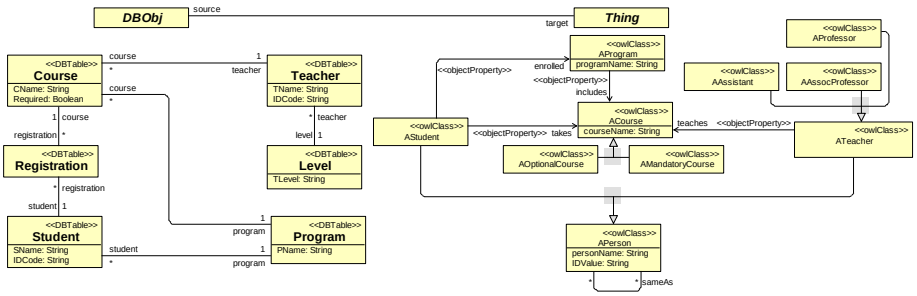

Fig. 35 Complete metamodel for mini-university example

In this example, there are different mapping situations. Besides trivial mappings, quite sophisticated ones are present as well.

Let's start with the obvious things. For objects of the class "Student", we just have to create the according objects of the class "AStudent" and copy the attribute values (for each value of "IDCode" attribute, we will create an independent instance of the class "APersonID"). For objects of the class "Program" the situation is quite similar. In the case of objects of the class "Course", the situation is more interesting because we would like to split the data from the table Course into two different classes: "AOptionalCourse" and "AMandatoryCourse", depending on the value of attribute "Course.Required". The name of the course has to be copied as well. A similar situation can be found in the case of "ATeacher". The only difference is the fact that here we are splitting data depending on the value found in another table (Level.TLevel). Besides all that we should take care of all associations in the source ER model and substitute objects of class "Registration" that was used as a table storing information about the many-to-many relationship with instances of the association "takes". In the case of RDF, there is no need for this kind of table.

In Fig. 36, we can see an A-MOLA program implementing the transformation for the mini-university example. Let us briefly comment on this program. We start our program with mappings for classes.

In the first loop (in MOLA *foreach* loops are denoted with black rectangles) we say that for every object *o* of the class *Course* where *o.required* is true, we create the object of the class

A*MandatoryCourse*. A traceability link between the object of the class *Course* and object of the class *AMandatoryCourse* is created (elements to be created are denoted with a dashed line in MOLA) as well. The assignment *"courseName := dbCourse.cName"* specifies the mapping of attribute values. The second *foreach* loop deals with those *Courses* that are mapped to *AOptionalCourses*.

The third loop maps objects of ER class *Program* to the ontology class *AProgram*. Then there are three *foreach* loops dealing with the migration of teachers – each instance

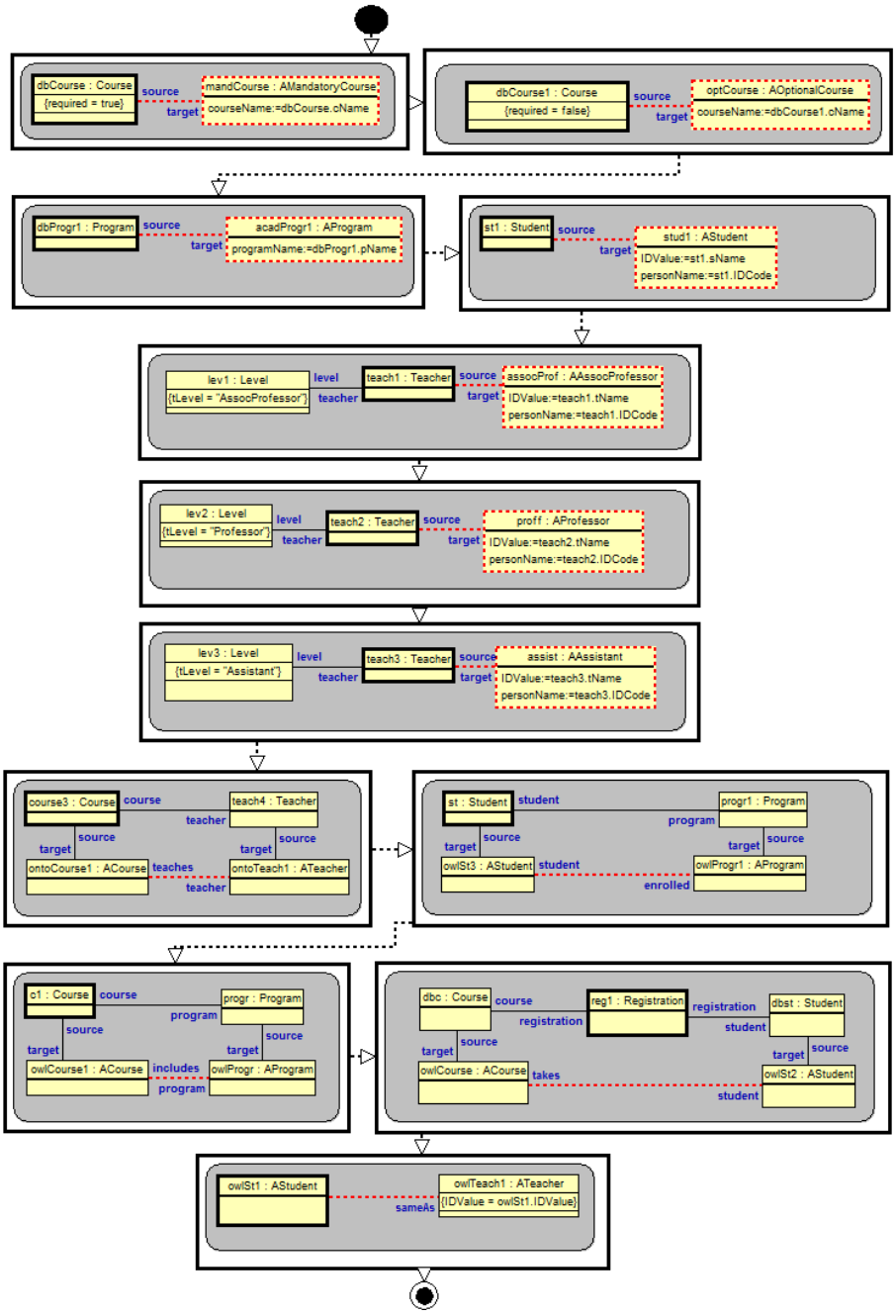

Fig. 36 Transformations for mini-university example

of the class *Teacher* is mapped to the instance of either the class *AProfessor*, class *AAssocProfessor*, or class *AAssistant*, depending on the value of the attribute *tLevel* of the *Level* object connected to this object of the class *ATeacher*. After this, we specify the mapping of DB associations to OWL associations (object properties). Finally we create

sameAs links between objects of the classes *ATeacher* and *AStudent*, having identical *IDValue*'s.

6.2.3 Results of Practical Application

The proposed method has been tested in practice. We have successfully migrated databases of 6 Latvian medical registries into a single shared RDF database. All correspondences between ER schema and OWL ontology have been specified with model transformations. These registries have the following data volumes:

- source relational database
  - 100 tables
  - 1300 columns
  - 3 million rows
- target OWL ontology
  - 170 OWL classes
  - 200 OWL data type properties
  - 800 OWL object properties
  - 41 millions RDF triples

Total time needed for migration was 8 hours, of which 1 hour and 30 minutes for data import from relational database, 5 hours and 30 minutes for data transformation, 1 hour and 10 minutes for data export to NTriples. The time necessary for the specification of correspondences between the database and ontology was about 3 person days for one register.

6.2.4 Analysis of the Proposed Method and Future Work

In this section, a new model transformation-based approach for the migration of relational data to RDF has been presented. Practical experience of the successful application of this method has been reported as well. The main advantages of the proposed method are easily readable, easily understandable and easily writable transformations specifying mappings between RDB and RDF. It becomes possible because of using a specific subset of a graphical model transformation language MOLA, called A-MOLA. One more important thing is the fact that our approach is self-consistent. It means that there is no need for some

additional external mechanisms to be used for specifications of “sophisticated” mappings. In the worst case we can supplement A-MOLA with additional facilities found in full MOLA language.

To summarise, there are two reasons for the introduction of A-MOLA language: methodological – by restricting facilities allowed in the transformations definition we can obtain more readable and comprehensible transformation and technical – it is expected that for this subset of MOLA, efficient implementation by direct translation to SQL can be obtained.

It should be noted that the proposed method has some room for improvement. Two things that in our opinion should be improved are:

- problem of excessive RAM consumption (for instance to migrate a relational database containing 3 Gb of data we needed about 8 Gb of RAM)
- execution performance.

In the following sections we take a more in-depth look at these problems, describe possible solutions and give directions for future research.

6.3 Problem of Excessive RAM Consumption

Above we described a novel model transformation-based method for the specification of correspondences between the elements of DB schema and OWL ontology. The main advantage of the proposed method is easily readable, easily understandable and easily writable transformations specifying mappings between RDB and RDF.

This method has also been approbated on a real world use case by successfully migrating relational databases of 6 Latvian medical registries [5] into a single shared RDF database [6]. Migration was successfully finished, but during this practical approbation it became clear that the proposed method can be improved with respect to RAM usage and execution performance. For instance, to migrate the aforementioned relational databases of 6 Latvian medical registries containing 3 Gb of data, we needed about 8 Gb of RAM. The reason for

such extensive RAM consumption is related to the fact that the model transformations performing the migration of data are implemented on top of an in-memory data store. One more problem stemming from the usage of an in-memory data store is the necessity to import source relational data to an in-memory data store before mappings that specify correspondences between the source ER schema and the target OWL ontology can be executed. In the case of the aforementioned migration of medical data bases, this import took about 1 hour and 30 minutes. The execution performance on data bases of a substantial size could be improved as well.

Below we try to overcome problems related to the use of an in-memory data store rather radically – we use relational DBMS instead of an in-memory data store for the implementation of the model transformation-based migration method. In this case, DBMS takes care of memory management, and the problem of excessive RAM usage disappears. The problem of data import to an in-memory data store also becomes obsolete – data are transformed in place (in a relational database). The next section gives a more detailed description of the above-mentioned ideas.

6.3.1 Migration Method on the Basis of DBMS

The conceptual schema of the improved method is given in Fig. 37. According to this schema the migration process consists of the following steps:

1. Import target OWL ontology into the database. During this import we create relational tables, representing target OWL ontology and its instances.
2. Specify mappings between the source class diagram (automatically obtained from source relational schema) and the target OWL ontology. These correspondences are specified in a high-level graphical model transformation language MOLA.
3. Compile MOLA program to SQL.
4. Execute the obtained SQL code – this code populates relational tables representing OWL ontology, with data to be exported as prescribed by MOLA mapping specified in the previous step.
5. Export data, contained in relational tables, representing OWL ontology to RDF files.

It should be noted that all of these steps, except for the specification of mappings that requires domain knowledge and is dependent on a specific pair of source ER schema and target OWL ontology, are fully universal (in the sense that they are not dependent on the target ontology or the source ER schema) and do not require any human involvement. One of the corner stones of this method is the transformation of one MOF [13] metamodel instance to another metamodel instance. This problem is also quite well known in a classical MDA approach, where the Platform Independent Model (PIM) is transformed to a Platform Specific Model (PSM). It is a classical use case for model transformation languages with the only difference being that the metamodel and its respective instances are stored in a database. However, model transformation languages are usually implemented on top of some metamodel based data store, and not on top of a relational data base.

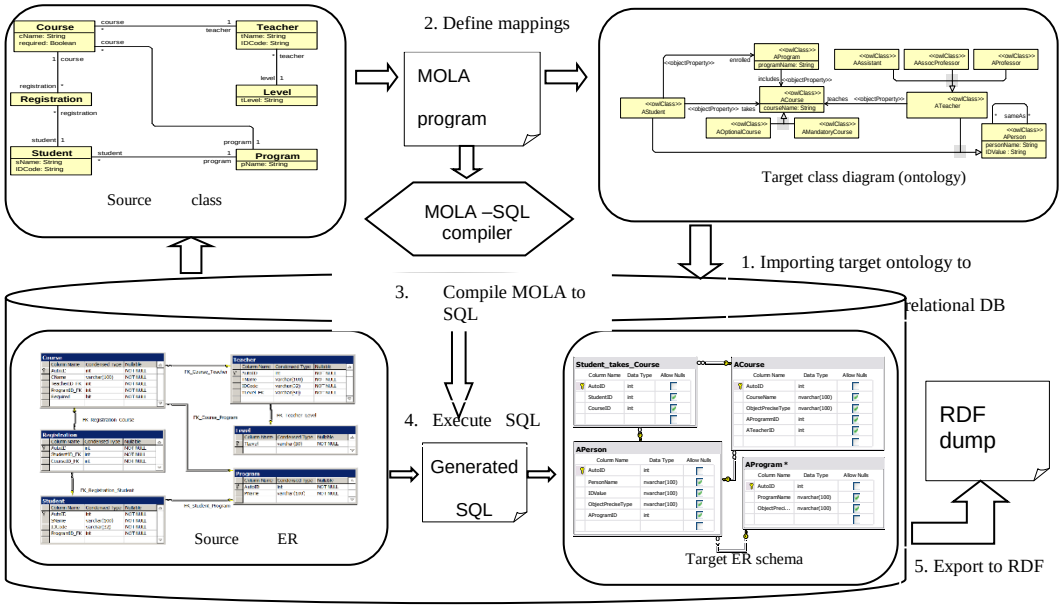

Fig. 37 The conceptual schema of the proposed method

Therefore it is vitally important to implement an efficient compiler of MOLA to SQL. Our first impression was that this kind of efficient compiler would be rather difficult to implement – because SQL supports efficient manipulation on the level of tables (or "big data chunks") but MOLA operates on the level of individual objects. So here we had 2 options:

- Compile MOLA to row-by-row SQL operations (instead of set operations).
- Restrict MOLA expressivity in such a way that only class level processing would be allowed (processing of individual objects would be prohibited).

We chose the first option, and therefore the main goal of this approach is to verify how the proposed model-transformation based migration method will perform from the efficiency point of view.

Traditionally, model transformation languages are implemented by compiling high-level transformation language to lower level base transformation language and finally compiling base transformation language to executable code [1].

In the case of MOLA the base transformation language is L0 [2], and there is already an efficient MOLA to L0 compiler [29] implemented through a bootstrapping method [1]. This is why we assume that the transformation program is specified in L0, and we concentrate on the efficient compilation of the L0 program to SQL.

6.3.2 Accessing Source ER Schema

To implement the above-mentioned ideas we need to transform a database ER schema to a UML class diagram. It is quite simple:

1. Every table T from the source relational schema is represented with class C. Every record R of table T is represented with an object O of class C.
2. Every column *Col* of a table *T*, excluding FK columns and PK columns, is represented by an appropriate class attribute *A*. Values of column *Col*, are represented by values of attribute *A*.
3. Every foreign key relation is represented with an association between appropriate classes.

6.3.3 Importing Target OWL Ontology

To provide a way to process target OWL ontology with corresponding instances, an ER schema, representing target OWL ontology, is created in a data base that contains source relational data.

In general, the representation of OWL ontology with relational tables can be a rather non-trivial task. However, we confine ourselves to only those ontologies that conform to the

UML/OWL subset. The basic idea of this subset is to only use those OWL DL constructs that can be adequately represented with UML class diagrams. If ontology conforms to a UML/OWL subset, we can derive quite a simple set of rules for importing the ontology into a relational database:

- For every <<owlClass>>, a corresponding relational table is created.
- For every owl <<objectProperty>>, a foreign key relation between appropriate tables is created. If the object property represents an m:n relation, a junction table is created.
- For every owl <<datatypeProperty>>, a table column of appropriate type is created (at the given point we support only four elementary types – real, boolean, string, integer).
- Inheritance hierarchies are flattened. It means that we create only one relational table corresponding to the root of this hierarchy and all classes derived from this root will be represented by the same table (special descriptors are used to denote the precise object type).

Now we can freely manipulate elements of this ontology through a model transformation program.

6.3.4 Exporting RDF Triples

The last step of the migration process is the generation of NTriples from instances contained in relational tables representing OWL ontology:

- For every row of every table, we generate triples in the form <ObjURI> <rdf:Type> <ClassURI>.
- For every link between objects (be it a simple foreign key relation or a junction table), we generate triples in the form <ObjURI1> <ObjPropURI> <ObjURI2>.
- For every column value present in the relational tables representing OWL ontology, we generate triples in the form <ObjURI> <DataTypeURI> <AttributeValue>.

6.3.5 Main Step: Compilation of L0 to SQL

L0 language is a textual low-level model transformation language. Its constructs can be divided into the following groups:

- Model (both objects and links) iteration commands
- Model update commands

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

- Control flow commands
  - Low-level control flow instructions and labels

All commands are executed on the level of an individual object. There are no commands that allow set processing.

We start the explanation of compilation principles with model iteration commands. The most straightforward approach for iterating through SQL table rows are SQL cursors. However, there are also some penalties for using them. The main one being performance problems. This is why we choose to compile L0 model iteration commands to row-by-row SQL operations without using cursors. The main idea of this compilation is to represent L0 pointer by several SQL variables representing the state of it. If a pointer is used to iterate through class objects, the following information about the pointer state is required:

- initialisation kind – shows what L0 command was used to initialise the pointer (there are 3 options: *first*, *first from by*, *first from where*)
- current row id – represents id of the current row to be processed
- next row id – represents the next row to be processed

In the case of iteration through association instances-links (*first from by*) and iteration through objects satisfying given conditions, we need more information about the pointer state, but here we will omit these details.

Model update operations are translated to SQL INSERT and UPDATE sentences modifying rows representing corresponding elements of the target ontology. As for control flow commands, compilation is rather straightforward: L0 goto command and labels are compiled to its direct counterparts in SQL. Other control flow instructions are compiled to a related constructs of SQL. In Table 2, an illustration of these principles is given.

Table 2. Principles of L0 command compilation

First <pointerName> : <Class> **else** <Label>;	**SET** @<pointerName >_InitKind = 1; **SET** @next_<pointerName >_RowID = NULL; **SELECT** @next_<pointerName >_RowID = **MIN**(GetTablePK(<Class>)) FROM <Class>; **IF ISNULL**(@next_<pointerName >_RowID,0) = 0 **BEGIN GOTO** <Label>; **END ELSE BEGIN** **SET** @curr_<pointerName>_RowID = @next_<pointerName>_RowID; **END;**
Next <pointerName> **else** <Label>;	**IF** @<pointerName>_InitKind = 1 **BEGIN** **SET** @next_<pointerName>_RowID = **NULL** **SELECT** @next_<pointerName>_RowID = **MIN**(<referencedClassName>.GetTablePK(<referencedClassName>) **FROM** <referencedClassName> **WHERE** <referencedClassName>.GetTablePK(<referencedClassName>) > @curr_<pointerName>_RowID; **IF ISNULL**(@next_<pointerName>_RowID,0) = 0 **BEGIN GOTO** <Label>; **END ELSE BEGIN**; **SET** @curr_<pointerName>_RowID = @next_<pointerName>_RowID; **END**; **END**;
addObj <pointerName>:<ClassName>;	**INSERT INTO** [<className>] default values; **SET** @curr_<pointerName>_RowID = **SCOPE_IDENTITY()**; **SET** @next_<pointerName>_RowID = @curr_<pointerName>_RowID;
addLink <pointer1Name>.<roleName>.<pointer2Name>;	**UPDATE** GetReferencedClassName(<pointer1Name>) **SET** <roleName> = @curr_<pointer2Name>_RowID **WHERE** GetReferencedClassName(<pointer1Name>). GetTablePK(GetReferencedClassName(<pointer1Name>)) = @curr_ <pointer1Name>_RowID;
label <labelName>;	<labelName>:
goto <labelName>;	**GOTO** <labelName>;

6.3.6 Results of Practical Application

We have implemented a prototype of the L0-SQL compiler with respect to the principles specified above and compared the performance of the migration process when it is

implemented on top of an in-memory data store and when it is implemented on top of relational data base (i.e., transformations are executed in place). At the given moment we have compared the migration of only one relational database – System Core – instead of 6 medical databases as it was in the original setting. There were several reasons for this. Firstly, it is easier to test the migration of one DB. Secondly, in our prototype implementation of the L0-SQL compiler, some advanced transformation language features (derived class processing, m:n association processing, connecting multiple types to an object) have not been implemented yet, but these features are used for the migration of the remaining 5 DB. It should be stressed that it is definitely clear how to implement them. According to this comparison, the migration method implemented on top of relational DB demonstrates approximately 2 times better performance. A similar performance improvement is expected for the remaining 5 DB. Problems concerning excessive RAM usage (DBMS takes care of memory management) and the necessity to import relational data have also been overcome.

To summarise, we have shown that by implementing the model transformation-based migration method on top of relational DB we can solve the problem of excessive RAM usage and improve execution performance. However, performance improvement was not sufficiently satisfactory. The not particularly impressive performance improvement in the above proposed method can be explained by the following facts:

1. compilation process starts with L0 code, which is obtained by compiling A-MOLA code
2. target SQL code relies on the use of row by row processing, meaning that data transformation is not being performed on the level of tables (sets), but on the level of individual records.

In the case if the compilation process were to start with A-MOLA, instead of L0, we could translate it to SQL, which uses high-level set operations (instead of processing individual records). It looks like in this case, substantial performance improvement could be achieved. A direct compilation of A-MOLA to high-level SQL set operations constitutes a potential direction of future research.

6.4 Performance Problem

Above, we have described two variations of model transformation-based approaches for the migration of relational DB to RDF. Both of these approaches heavily suffer from performance problems. In this section we analyse what the root cause of these problems is and propose a solution for them – improved approach for the implementation of model transformations. The proposed approach does not rely on the use of any repository – metamodel and its instances are stored directly in RAM without any middleware.

Before offering improvements, we need to understand what the cause of the performance problems is. In general, after compilation, the model transformation program is a C++/Java program (depending on the target environment), that spends most of its execution time executing model processing commands. In our particular case, model transformations that are used to migrate relational databases to RDF are quite specific, since they almost never perform delete operations – only create instances (both class and associations), get/set attribute values and iterate through existing class and association instances. To better understand the current situation (define a baseline) we set up the experiment, where we compare the performance of individual model processing operations in different execution environments. The first environment that is examined is C++ with JR [41] repository for model operation processing (**C++/JR**). The second environment – C++ without repository (**C++/new**), models are stored directly in RAM, using "natural coding" (Model class is represented with a C++ class, model object is represented with a C++ object. Associations between classes are represented by std::vector containers with pointers to partner objects. Attribute values are stored in corresponding class members.) New objects are created with C++ *operator new*. The third environment – Java with model instances stored according to the aforementioned "natural coding" (**Java**). The fourth environment is C++ without repository (**C++/placement_new**), model instances are stored directly in RAM according to "natural coding", but unlike C++/new environment, here we create objects using C++ placement operator new (first we allocate a memory buffer that will hold newly created objects and then every new object is placed in this buffer by C++ *placement operator new*).

For every abovementioned environment we created the following programs and measured their corresponding execution times:

- addObj – program creates 1000*1000 objects of one class
- setAttr – program creates 1000*1000 objects of one class and for every created object, creates and sets an attribute value (all attribute values are different). Two subcases are considered here: one for attributes of the type *string*, one for attributes of the type *int*.
- addLink – program creates 50*1000 objects of class A, then for every object of class A a_j creates 30 objects of class B b_i and creates one link between the corresponding a_j (j=1..50*1000) object and every newly created b_i (i=1..30) object
- first – program creates 1000*1000 objects of one class and then measures how much time it takes to iterate through all the newly created objects
- first/get – program creates 1000*1000 objects of one class and for every created object creates and sets some attribute value of the type *int*. Then it measures how much time is needed to calculate the sum of all attribute values. The sum is calculated by iterating through all objects and obtaining the attribute value stored in each individual object
- firstfromBy – program creates 50*1000 objects of class A, then for every object of class A a_j (j=1..50*1000) creates a batch of 30 objects of class B b_i (i=1..30) and creates one link between a_j and every one of b_i from a batch of 30 B, created for this a_j. Then the program measures how much time is needed to iterate through all aj and all bi connected to the current aj.

Execution times of these programs in different environments can be found in the following table.

Table 3 Execution times of elementary model processing operations

		1000*1000		
	Java	**C++ / new**	**C++ /placement_new**	**C++/JR**
	ms	**ms**	**ms**	**ms**
addObj	76	1828	23	125
setAttr int	111	1854	24	37862
setAttr str	261	4921	5261	36121
first	10	3	2	20
first/get	11	6	4	335
		A = 50*1000 B = 30		
addLink	239	3929	3055	551
firstFromBy	19	7	20	

By analysing execution times several observations can be made:

1) If we compare a C++ environment that does not use repository (one where objects are placed in RAM(**C++/new**) and one where objects are placed in a pre-reserved memory buffer (**C++/placement_new**)) we can see that there is a difference of almost 2 orders of magnitude in the case of object creation operation (addObj). Other operations in these environments have mostly comparable execution times.
2) The best overall performance (with the exception of 2 operations setAttr/str and addLink) can be observed in the **C++/placement_new** environment. The next best environment (**Java**) demonstrates 2-5 times worse performance depending on the operation to be considered. The only exception is the performance of operations setAttr/str and addLink, the execution of which took substantially more time in the **C++/placement_new** case compared to the **Java** case. The difference can probably be explained by the fact that both setAttr str and addLink operations implicitly use a new operator for memory management (for string allocation and vector element allocation), which degrades performance.
3) If we compare C++ environments that use a repository with ones that do not – we can see that most operations in the case of C++ with repository (C++/JR) are 2-5 times slower. A **notable exception** here is an operation that sets attribute values. In

the case of C++/placement_new it takes 24 ms, but in the C++/JR case it takes 37862 ms. Such a big difference can be explained with a way setting of attribute value works in the JR repository. To set the attribute value in the JR repository we need to create a *data value* with a certain type (repository receives *data value* seed in a textual form), then we need to create an attribute link of a corresponding type between a created data value and an object for which the attribute value is set. When a new *data value* is being created, uniqueness control takes place, and it is checked if the repository already contains a data value that we would like to use as an attribute value. In the repository, existing data values are stored on a list – to perform the uniqueness check of a newly created data value, the repository performs a linear search in this list. As a consequence, the attribute setting operation has a linear complexity from the number of already existing data values in the repository.

The experiment showed that there are environments that can provide better performance (at least in cases when transformations only create new model elements and do not delete existing model elements – a typical process of migration of relational DB to RDF complies with this condition) than the currently used C++/JR environment does. To test this hypothesis we decided to rewrite model transformations performing data migration from relational DB to RDF without using a repository. The approach according to which such rewriting has been performed is described below.

6.4.1 An Approach to Data Migration Without Use of a Repository

According to the new approach, the data migration process will consist of the following steps:

1) Generate Java classes from ER schema
2) Generate Java classes from ontology
3) Write a Java program that performs data migration (reads data from classes corresponding to the ER schema and creates data (objects, links, attribute values) in classes corresponding to ontology) by working with the generated classes as a model store

4) Run Java program – it will do 3 things:
    a. Import data from source DB – read relational DB and populate with instances of Java classes corresponding to source ER schema
    b. Transform data – read instances of Java classes corresponding to source ER schema and populate with instances of Java classes corresponding to ontology
    c. Generate .ntriple files from instances of Java classes corresponding to OWL ontology

In the figure below we can see an overall schema of a proposed approach.

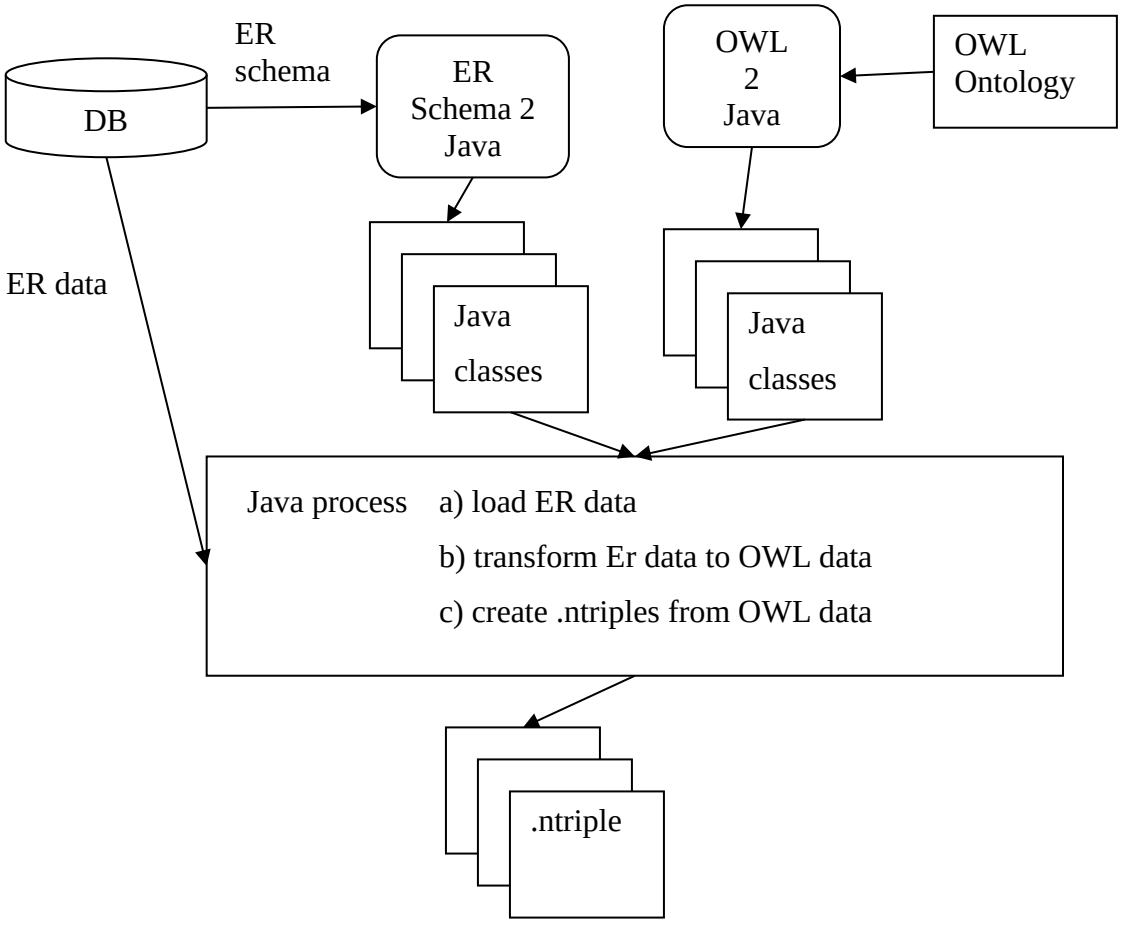

Fig. 38 A migration approach without repository

Below we describe each of these steps in a more detailed way:

ErSchema 2 Java:

Here we describe how Java classes representing ER schema to be migrated are generated:

1) For every table T from relational database we create a corresponding Java class JavaClass(T). A generated Java class will have a static member that holds references to all created objects of this class.

2) For every column Col of database table T, that is not a foreign key, we create a member of appropriate type JavaAttr(Col) in a corresponding Java class JavaClass(T)

3) For every foreign key FK we add attributes to a corresponding source JavaClass(SourceTable(FK)) and target JavaClass(TargetTable(FK)) classes. These attributes will represent foreign key references of relational DB with Java references. These attributes will have the following form:

- target class:

  public ArrayList< JavaClass(SourceTable(FK)) > SourceColumnName(FK) = new ArrayList();

- source class

  public ArrayList< JavaClass(TargetTable(FK)) > TargetColumnName(FK) = new ArrayList();

4) For every generated JavaClass(T) corresponding to relational table T, we generate data loading methods – these methods will populate Java class JavaClass(T) with instances that correspond to rows of a relational table T. A generated method connects to DB through JDBC, reads all the rows from table T and for every row r creates the object o of java class JavaClass(T). After that, object o attributes are populated with the corresponding column (only non FK columns are considered) values of row r belonging to table T. During this process we store mapping between the created Java object o and corresponding relational table row r. Technically this mapping is represented as a mapping form a pair of table name and value of table primary key column to a Java reference that points to the Java object corresponding to this row (<table_name, pk_row_id> -> java_ref).

5) For every foreign key FK links loading method is created. This method connects to DB through JDBC, reads rows that correspond to FK and for every row creates a link of

appropriate type between corresponding Java objects. Links are modeled with ArrayLists; thus to create an individual link, we place references to corresponding source and target Java objects in ArrayLists representing the respective FK. The correspondence between the database table row and respective Java object is determined using the mapping described in the previous step.

OWL 2 Java

1) For every OWL class C we generate a corresponding Java class JavaClass(C). This generated class will contain a static attribute, that will hold references to all created objects of this class.

2) For every OWL datatype property ODP we generate an attribute of a corresponding type in a respective Java class JavaClass(Domain(ODP)).

3) For every OWL object property OOP we generate an association between the corresponding Java classes JavaClass(Domain(OOP)) and JavaClass(Range(OOP)). Association is implemented with ArrayList fields in both corresponding classes.

NTriple generation

For every class generated in the previous step, we generate methods that will be able to generate Ntriples:

1) for every class we generate the method generateClassNtriples() – this method iterates through all objects of its parent class and for every object creates n-triple in the form:
<objID> <rdf:type> <OWLClassID>.

2) For every attribute of a generated class we generate the method generate<AttrName>NTriples – this method iterates through all objects of a parent class and for every object, generates n-triple in the form:
<objID> <attrID> <attrValSpecification>.

3) For every association that starts from this class we generate the method generate<AssocName>NTriples, that iterates through all objects of this class and generates ntriples for instances of <AssocName> (it is for pairs of objects connected by <AssocName>). Ntriple will be in the form: <startObjID> <assocID> <targetObjID>.

6.4.2 An Example of Data Import Step

Let's illustrate the data import from the relational data base to Java classes according to the new method with a concrete example.

We start with a simple ER schema. It has only two tables, Persona and Gender; both tables have primary keys(PersonID and GenderID respectively) and several other columns(Person.PersonName, Gender.GenderName). There is one foreign key relationship between the tables.

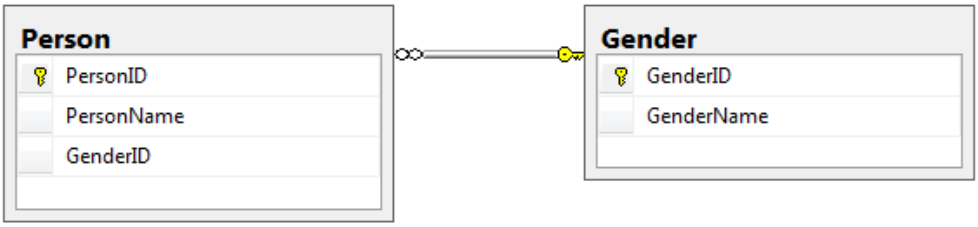

Fig. 39 An example ER schema

Data base tables contain the following data. Table gender has only 2 rows in it:

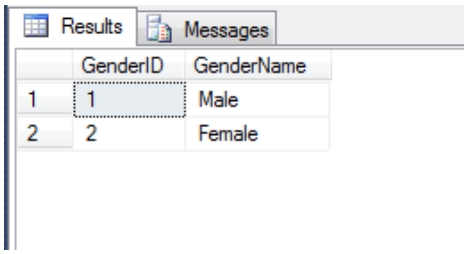

Fig. 40 Table Gender

Table Person has 3 rows in it:

Results | Messages

	PersonID	PersonName	GenderID
1	1	Frank	1
2	2	Eve	2
3	3	Bob	1

Fig. 41 Table Person

One can see how persons are linked with genders, by using the foreign key column GenderID. From the ER schema seen above we generate two Java classes: Person.java and Gender.java. A structure of these classes with some comments of what can be seen in what line can be found below.

Person.java | Gender.java

```
1  package dbexample;
2
3  import java.sql.Connection;
4  import java.sql.ResultSet;
5  import java.sql.SQLException;
6  import java.sql.Statement;
7  import java.util.ArrayList;
8  import java.util.HashMap;
9  import java.util.Map;
10
11 public class Gender {
12
13     public static final ArrayList<Gender> instances = new ArrayList<Gender>(1);
14
15     public int GenderID;
16     public String GenderName;
17
18     public ArrayList owlObj = new ArrayList(1);
19
20     public static Gender createGender() {
25
26     public static long LoadGender(Connection conn, Map<String, Map<Integer, Object > > classNam2classObjectsMap)
27     throws SQLException {
57
58     public static long LoadGender_GenderID_GenderID_Person
59     (
60         Connection conn,
61         Map<String, Map<Integer, Object > > className2classObjectsMap
62     )
63     throws SQLException {
91
92     public ArrayList<Person> GenderID_1 =  new ArrayList();
93 }
94
```

Fig. 42 Gender.java

11 – start of Java class Gender definition

This class corresponds to table Gender.

13 – a static member *instances* of the class Gender is defined; during the program execution time, this list will contain references to all created instances of the class Gender

15 – attribute holding values of field GenderID

This attribute corresponds to column GenderID of table Gender

16 – attribute holding values of field GenderName

This attribute corresponds to column GenderName of table Gender

20 – helper method to create objects of the class Gender

26 – helper method that reads data from the database table Gender and creates corresponding objects of the Java class Gender

58 – helper method that loads data of the foreign key relationship (Person<>Gender) from the database and creates links between corresponding Java objects

92 – a list that contains references to objects of the class Person, that are linked to *this* object of the class Gender by association (Person<>Gender)

This list corresponds to the foreign key relationship between the tables Person and Gender in the direction from Gender to Person.

```
1  package dbexample;
2
3  import java.sql.Connection;
4  import java.sql.ResultSet;
5  import java.sql.SQLException;
6  import java.sql.Statement;
7  import java.util.ArrayList;
8  import java.util.HashMap;
9  import java.util.Map;
10
11 public class Person {
12
13     public static final ArrayList<Person> instances = new ArrayList<Person>(1);
14
15     public int PersonID;
16     public String PersonName;
17
18     public ArrayList owlObj = new ArrayList(1);
19
20     public static Person createPerson() {
25
26
27     public static long LoadPerson(
28         Connection conn, Map<String, Map<Integer, Object > > classNameClassObjectsMap
29     )
30     throws SQLException {
60
61
62     public ArrayList <Gender> GenderID_0 = new ArrayList();
63
64
65 }
66
```

Fig. 43 Person.java

11 – start of definition class Person

This class corresponds to the table Person.

13 – a static member *instances* of the class Person is defined; during the program execution time, this list will contain references to all created instances of class Person

15 – attribute holding values of field PersonID

This attribute corresponds to the column PersonID of the table Person

16 – attribute holding values of field PersonName
This attribute corresponds to the column PersonName of the table Person
20 – helper method for the creation of class Person objects
27 – helper method that reads data from the database table Person and creates corresponding objects of the Java class Person
62 – a list that contains references to objects of class Gender, that are linked to *this* object of class Person by association (Person<>Gender)
This list corresponds to the foreign key relationship between the tables Person and Gender in a direction from Person to Gender.

We finish our example by showing how the data imported from the relational data base is represented with Java class instances. In the figure below we can see two lists: one list containing instances of Java class Person and one list containing instances of the class Gender. Class Gender has two instances. Class Person has three instances. These instances represent the data imported from the data base shown above. Every object of the class Gender has 2 attributes of elementary types: GenderID and GenderName. Every object of class Person has two attributes of elementary types as well: PersonID and PersonName. Links between objects are implemented with attributes of type ArrayList. In the figure, links are depicted in two ways: with graphical arrows and as attributes of respective objects. We can see, that for example the object of the class Gender with a value of attribute GenderID equal to “1” is connected to two objects of the class Person (p1 and p3) through a link of type GenderID_1 (an inverse link GenderID_0 is also present, this links starts from objects of the class Person and ends at objects of the class Gender. This inverse link is represented with the attribute Person.GenderID_0 of type ArrayList.)

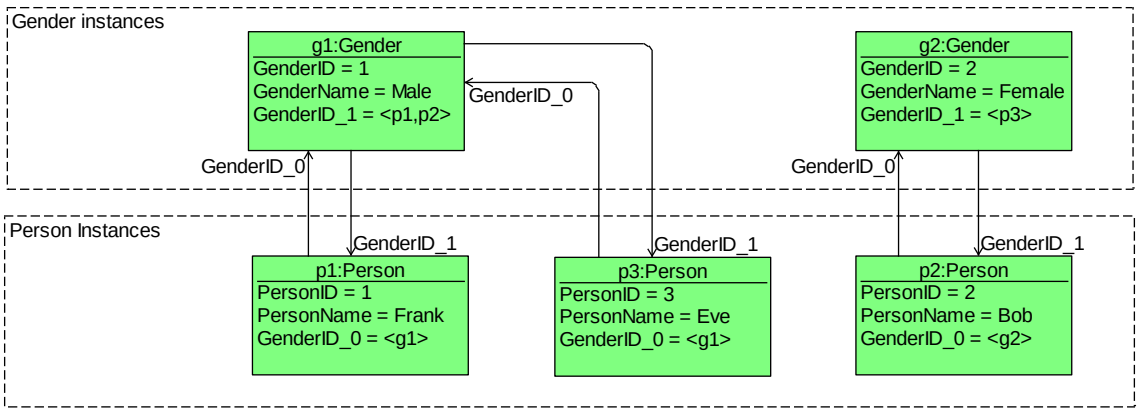

Fig. 44 Java instances corresponding to data base data

6.4.3 Results of Method Approbation

There are several ways of how to verify the method proposed above. An ideal way would be to create a compiler from transformation language L0 to Java (where metamodel and its instances are stored in RAM according to the principles described in section 6.4.1). Then compile the transformation for 6 medical registries used in the validation of the previous method with this new compiler and compare the execution times of the compiled transformations obtained with the new and old method. A simpler way is not to rewrite the L0 compiler, but manually rewrite register migration transformations in Java (that does not use a repository), writing Java code in such a way that the obtained code would closely resemble Java code that could be generated by a prospective compiler. We chose the simpler way – transformation for one the medical registries (System core) has been manually rewritten in Java and execution times have been compared.

Below we can see a relatively typical fragment of manually written Java code for data migration and the corresponding L0 code. We can notice that manually written Java code closely resembles a code that could be generated by a compiler. Let's start with L0 code – in Fig. 45, L0 transformation performing the migration of the data base table Preda.dbo.XAP is shown. With the help of the first (120) and next (136) instructions, we iterate through all objects of class PREDA_DB_dbo_XAP. For every object of this class: we create one object

of the class *Arsts* (124), and we create one mapping link of type owlObj (124) (this link is needed to be able to trace the correspondence between the created OWL object and source database object), the value of attribute *PREDA_DB_dbo_XAP_VOAVAAPID* is assigned to attribute *arVOAVAKods* (126), attribute *arAktivs* is filled with values from attribute *PREDA_DB_dbo_XAP_APStatuss* (if the value of *PREDA_DB_dbo_XAP_APStatuss* is "A", *arAktivs* gets the value *true*, if the value of *PREDA_DB_dbo_XAP_APStatuss* is "N", *arAktivs* gets the value *false* (128-134)).

```
116  procedure Arsts();
117    pointer a : Arsts;
118    pointer xap : PREDA_DB_dbo_XAP;
119  begin;
120    first xap : PREDA_DB_dbo_XAP  else done;
121    label nextObj;
122      addObj a : Arsts;
123
124      addLink xap.owlObj.a;
125
126      setAttr a.arVOAVAKods = xap.PREDA_DB_dbo_XAP_VOAVAAPID;
127
128      attr xap.PREDA_DB_dbo_XAP_APStatuss == "A" else L1;
129        setAttr  a.arAktivs = true;
130        goto L2;
131      label L1;
132      attr xap.PREDA_DB_dbo_XAP_APStatuss == "N" else L1;
133        setAttr  a.arAktivs = false;
134      label L2;
135
136    next xap else done;
137    goto nextObj;
138    label done;
139  end;
```

Fig. 45 L0 transformation for migration of table Preda.db.dbo.XAP

In Fig. 46, L0 transformation described above is manually rewritten in Java. With for cycle (organised with **first** and **next** commands) we iterate through all instances of class PREDA_DB_dbo_XAP. Then for every object of this class: we create the object of class *Arsts* (89), create a mapping link of type *owlObj* (90) – it is done by adding reference to partner objects to attribute owlObj (attribute owlObj is an ArrayList representing association owlObj – this association is the mapping association); after that we copy the

values of attribute *VOAVAAPID* to *arVOAVAKods* (92), and finally fill the values of attribute *arAktivs* with the values of attribute *APStatuss* (if the value of *APStatuss* is "A" we set arAktivs to *true*, if the value of *APStatuss* is "N", we set arAktivs to *false* (92-98)).

```
87      public static void Arsts() {
            for (XAP xap : XAP.instances) {
89              Arsts a = Arsts.createArsts();
90              xap.owlObj.add(a);
91
92              a.arVOAVAKods = xap.VOAVAAPID;
93
                if (xap.APStatuss.equals("A")) {
95                  a.arAktivs = true;
96              } else if (xap.APStatuss.equals("N")) {
97                  a.arAktivs = false;
98              }
99          }
100     }
```

Fig. 46 Java transformation for migration of table Preda.db.dbo.XAP

In Table 4 we give a comparison of the execution times of transformation, performing System core migration. System core is one of the medical registries we migrated before [6, 9].

We compare the execution times of the transformation in three different environments: C++ debug mode, C++ release mode, and Java. C++ programs (that are in fact L0 transformations compiled according to the principles described in section 6.2) use a repository for model operation processing. Java environment does not use a repository, the model is stored directly in RAM according to the principles presented in section 6.4.1. Migration transformation has three parts: data import from the relational data base to the model, transformation from the model corresponding to the ER schema to the model corresponding to OWL ontology, data export from the model corresponding to ontology to .ntriple files. We present both the total execution time and time for the execution of each separate part.

Table 4 Execution times of System core migration transformations

		Execution time					
		Import	**Transformation**	**Ntriples**	**Total**	**Total, s**	
C++	debug / VS2008	0:10:03	0:12:45	0:06:12	0:29:00	1680	
C++	release / VS2008	0:02:29	0:02:13	0:01:14	0:06:23	383	
Java		0:00:08	0:00:05	0:00:21	0:00:38	38	

By analysing execution times we can see that the smallest total time in the case of C++ environments using JR repository (C++/release) is 6 minutes and 23 seconds. In the case of the Java environment (model is stored according to the principles from section 6.4.1) transformation execution took 38 seconds (almost 10x difference). If we compare the execution times of separate steps, we can see that in transformation the step difference between execution times is even more expressed: 5 seconds (Java) versus 2 minutes and 13 seconds. Such a notable performance improvement allows us to assert that the hypothesis presented in section 6.4 is plausible – by using the Java environment, where model is stored in RAM (and no repository is used), a significant performance improvement can be achieved.

By extrapolating the execution times of System core migration transformation in 3 different environments we can approximately estimate what the execution time would be for the migration of all 6 registries in the case of migration performed in the Java (no repository) environment. Total time needed for the migration of 6 registries in the case of the C++/JR environment (in debug mode) was close to 8 hours. Out of these 8 hours, the data import step took 1 hour and 30 minutes, the model transformation step took about 5 hours and 30 minutes, and model export to Ntriples took 1 hour and 10 minutes. From the analysis of the execution times of System core migration transformation we know that the difference between executions times in debug/release mode is about 5 times. Thus we can expect that in release mode the execution times for 6 registries will be approximately 5 times smaller as well: 18 minutes for import, 66 minutes for transformation, 14 minutes for export to Ntriples. By comparing the execution times of System core migration in the

C++/JR environment (release mode) and Java environment, we see a difference of almost 10 times. This allows us to expect that the execution time of migration of all 6 registries in the case of the Java (no repository) environment could take about 10 minutes. It should be stressed that this is only an approximate estimate and the real execution time for the migration of all 6 registries in Java (no repository) could be somewhat different. At the same time, it is reasonable to expect that the migration of 6 registries in the Java (no repository) environment will be substantially faster.

6.5 Conclusions

This chapter describes a series of experiments that were devoted to the use of model transformations in a fundamentally new application area – the migration of databases to RDF.

When designing this series of experiments, we had two main concerns that could have become points of failure for the entire experiment:

1. it could turn out that the specification of mappings between ER schema and OWL ontology would be too complicated (difficult to write/difficult to read)
2. it could be the case that when processing such atypically large amounts of data, the execution environment of the transformation language would not be able to execute the created mappings (too much memory is needed/execution is too slow)

.

Experiments performed have shown that these two concerns were not an obstacle to the successful completion of the migration process. The proposed method was tested in practice. We have successfully migrated 6 relational databases of Latvian medical registers to one common RDF database [5, 6, 9] using the migration method proposed in this section.

All correspondences between the ER schema and the OWL ontology were specified by model transformations.

The created RDF data warehouse with medical data was later practically used for ad hoc data analysis with the graphical query construction tool ViziQuer [4, 5], which allowed one to provide a new quality end-user interface with real medical data at the ontology level [85].

The first experiments showed that data migration using model transformations can be relatively slow in terms of execution time. However, the results of later experiments show that a speedup of about 10 times can be expected using the repository-less method.

Therefore, the main conclusion is that the use of model transformations for migration of relational data to RDF is definitely possible.

There are several directions of future research. The first one is improvement of compilation to SQL. The approach proposed in section 6.3 used translation from L0 to low level SQL (generated SQL code processed data on the level of individual records). This approach can be improved by starting the compilation process from A-MOLA. This would allow compiling of the A-MOLA construction to high-level SQL set operations instead of low-level operations processing data on the level of individual records. We suppose that this kind of compilation would allow one to achieve a substantial performance improvement.

One more quite natural direction of future research is to create L0 (or some other higher level model transformation language) compiler, that would generate target code, that does not use any kind of repository and processes the metamodel and its instances that are stored directly in RAM, according to “natural coding” (presented in section 6.4). The main benefit of this method would be improved transformation performance. A limitation of this method is the fact that we need to be able to place an entire model to be transformed in RAM. As a consequence, the method will not be able to migrate data bases that could not be stored in RAM (at least partially). At the same time it is known that the amount of available RAM is continually increasing. So one can expect that the application range of the proposed method and model transformation in general will also expand.

CHAPTER 7

Applications of L0 in Higher Level Transformation Language Implementation and Tool Building Platform Development

7.1 Introduction

In this chapter we briefly describe the main applications of L0 language that were mainly performed (in the context of applied projects of IMCS UL) by the colleagues of the author of this thesis. This chapter is included in the thesis to confirm the practical usability of the results of the thesis in real world projects.

7.2 Implementations of High-level Model Transformation Languages

As was already mentioned, one of the MDA cornerstones are high-level model transformation languages. On the one hand a rather typical construct found in a high-level model transformation language is pattern. On the other hand, one can notice that model transformation languages are implemented on top of metamodel-based data stores. These data stores provide a low level API, for manipulating metamodels and their instances EMF [37], JGralab [43], MDR [38], MII_REP [44]. One of the most important problems in the implementation of high-level model transformation languages is to find a way of how to represent patterns found in a high-level model transformation language with low-level operations found in an interface of a typical metamodel based data store. Practice shows that it is a difficult challenge. One of the reasons for this is the big semantic gap between high-level pattern and low-level operations, found in API of a metamodel-based data store.

The author of this work has participated in the development of the bootstrapping idea as a co-author in [1, 2]. The main contribution of the author was the design and implementation of the base transformation language L0. Further development of bootstrapping schema that is based on L0 language was carried out by A. Sostaks [52] and E. Rencis [62].

Let’s describe how these ideas were applied to the implementation of a high-level graphical model transformation language MOLA. To simplify MOLA implementation, a

sequence of model transformation languages (Lx language family) was proposed [1, 29, 30]. Specifically for MOLA implementation, the Lx language family consisted of languages L0, L1, L2 and L3. This language family has the following properties:

- L0 language; constructs found in this language are rather close to constructs found in the API of a typical metamodel based data store.
- Every language L_i is obtained by supplementing language L_{i-1} with new constructs.
- Language L3 already contains rather advanced facilities and its expressivity is rather close to the expressivity of a typical model transformation language.
- For every language from this family, except for language L0, which is the base language, a compiler written in L0 is built: from L3 to L2, from L2 to L1, from L1 to L0.

Transformation language L1 is obtained by adding pattern definition facilities to L0 language. The pattern in language L1 can contain constraints, restricts allowed attribute values, the presence of links of specific types between objects and such like. To be able to use pattern definition constructs in L1 we extend the syntax of the **first** and **next** commands:

first <pointerName1> **:** <className> [**from**
<pointerName2> **by** <roleName>] [**suchthat**
begin
<L1 Commands>
end]**;**
next <pointerName> [**suchthat**
begin
<L1Commands>
end]**;**

Suchthat block is a new kind of Boolean expression – *begin-end* expression. This expression value is true, if program execution reaches the **end** command, otherwise its value is false.

Transformation language L2 is obtained by adding *foreach* loop facilities to L1 language:

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

L3 language is obtained by adding the "*if-then-else*" construct to L2:

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

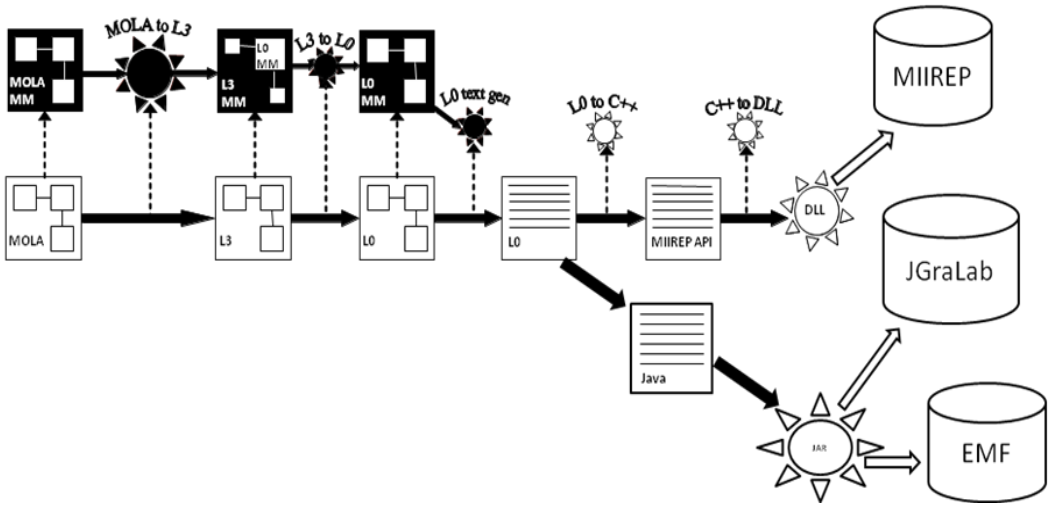

Fig. 47 MOLA- L0 compilation schema [52]

Finally, when we have language L2, we can build the MOLA compiler to L3. Thanks to the fact that the expressivity of MOLA and L3 are rather close, it is much easier to build the MOLA compiler to L3, than the MOLA compiler to L0. The full compilation schema is shown in Fig. 47. This idea (MOLA compiler to L3) has been practically approbated in the context of A. Šostak's thesis [52]. The obtained results confirmed the advantages of the bootstrapping approach for the implementation of model transformation languages – compiler development was possible with moderate allocation of resources and the developed compiler proved to generate quite efficient (from a performance point of view) code.

With this new approach to MOLA implementation based on the bootstrapping process, it was possible to achieve a significant speed improvement (in some examples up to 65 times [52]). As it is noted in [52] MOLA implementation through the Lx language family demonstrated impressive performance improvements. For example, for test models of size N <=10000, which is a typical model size in the MDSD (model driven software development) context, the transformation execution time was less than one second. Considering the fact that the examples used in the performance tests were selected as typical for MDSD transformations, these tests confirmed that the implementation of MOLA through languages of the Lx family has been sufficiently efficient for use in typical MDSD tasks.

This, in turn, allows us to reasonably state that one of the goals of this thesis – to create a low-level model transformation language that could be directly used in the implementation of higher-level transformation languages using the bootstrapping method – has been successfully achieved.

7.3 Tool Building Platform – GrTP

Another important L0 use case is a graphical tool building platform GrTP [18], operating in conformance with MDA principles Fig. 48 The structure of GrTP [18] It is interesting to note that L0 was used as a main transformation language (and not some higher level model transformation language such as MOLA – as it turned out pattern recognition facilities were not that necessary) in the development of this platform. It confirms that language is usable in real world applications and thus satisfies the requirements defined in section 3.1. This platform was developed at LU IMCS by J. Bārzdiņš, A. Zariņš, K. Čerāns, A. Kalniņš, E. Rencis, L. Lāce, R. Liepiņš, A. Sproģis. The main idea of this platform is a strict separation of domain model processing and user interface component processing. The only allowed way to define correspondences between domain elements and user interface elements is through model transformations.

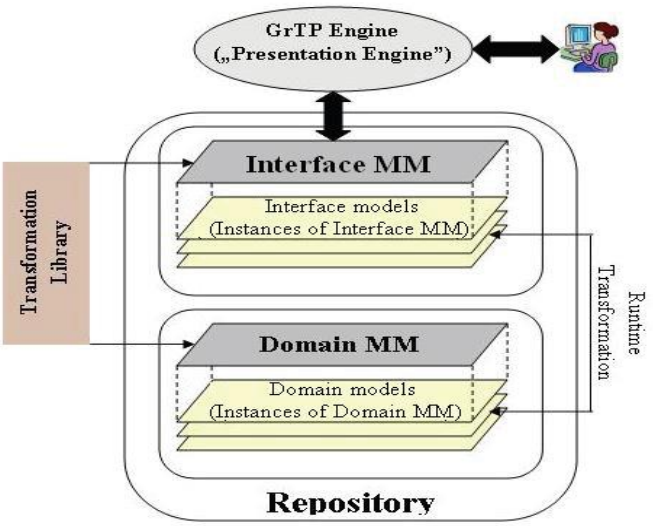

Fig. 48 The structure of GrTP [18]

Graphical tool building platform GrTP is based on the following ideas:

- There are two special kinds of metamodels:
  - User interface metamodel. Every instance of interface metamodel consists of a graphical view that represents the respective instance of the domain metamodel. The interface model also contains instances of classes that represent the symbol palette, its elements, and toolbar and user actions, performed on a specific user interface element.
  - Domain metamodel. In the case if we are building a class diagram editor, instances of the domain metamodel will represent a specific UML class diagram that conforms to the UML class diagram metamodel.
- A library of presentation engines is created. A presentation engine is a software component that can visualise instances of a specific metamodel and handle user actions that are specified in a metamodel.
- Transformations, defining an internal logic (how a specific tool handles user actions) of a specific tool, are created. These transformations define correspondences between domain model instances and user interface model instances. In the process of handling almost every user action, a transformation analysing and processing this action is called.
- GrTP platform is based on a highly efficient (from a performance point of view) metamodel based in-memory data store. This data store contains an interface metamodel (instances of this metamodel are interface models) and domain metamodel (instances of this metamodel are domain models).

By using the GrTP platform and writing the model transformations (that define the internal logic of both the platform in general and of the specific tools in particular) “directly” in L0, several practical tools were developed:

- **GradeTwo** [84] is a modelling tool containing a metamodel-based model transformation-driven graphical tool building platform used for developing graphical domain specific tools. GradeTwo incorporates two main features of good software – it is very easy to use (new domain specific tools can be generated in

hours not days) while it still provides a very large expressiveness (almost every feature one can imagine can also be integrated in the tool being created). This tool is used in several courses of the master's program at the Faculty of Computer Science of the University of Latvia.

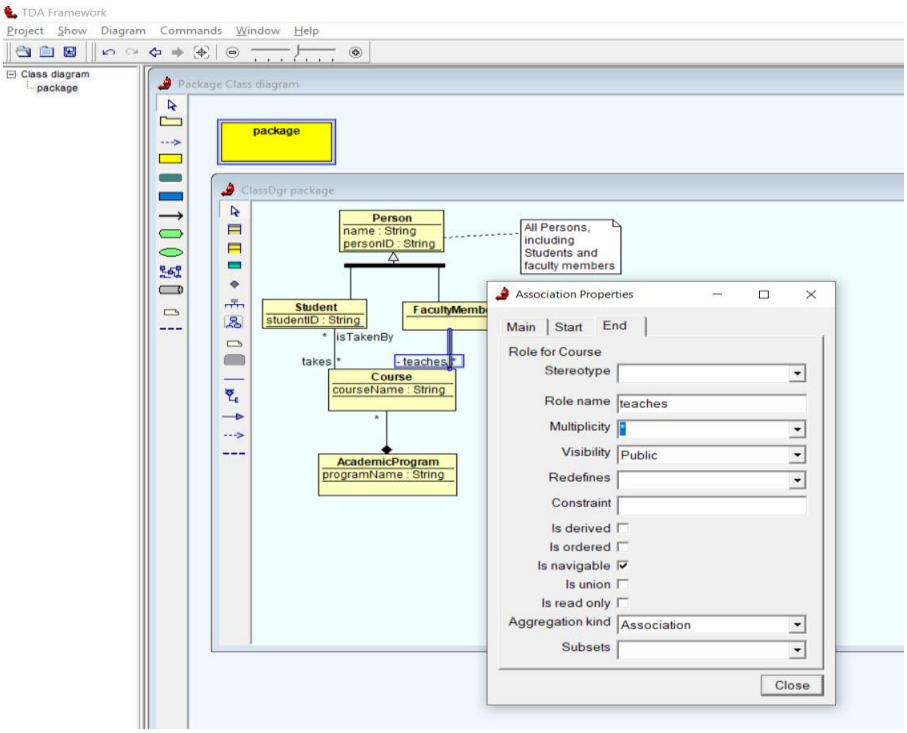

Fig. 49 GradeTwo tool

- **Viziquer** [4, 82] is a graphical tool that allows one to connect to a SPARQL endpoint and construct visual queries (which are then translated to SPARQL) to get data from this endpoint. Viziquer allows one to perform the visualisation and analysis of the data schema of the Sparql endpoint. This tool is designed to make working with RDF data easier. It should be noted that Viziquer (which was built using the L0 language) was one of the first tools for the visual construction of SPAQL queries.

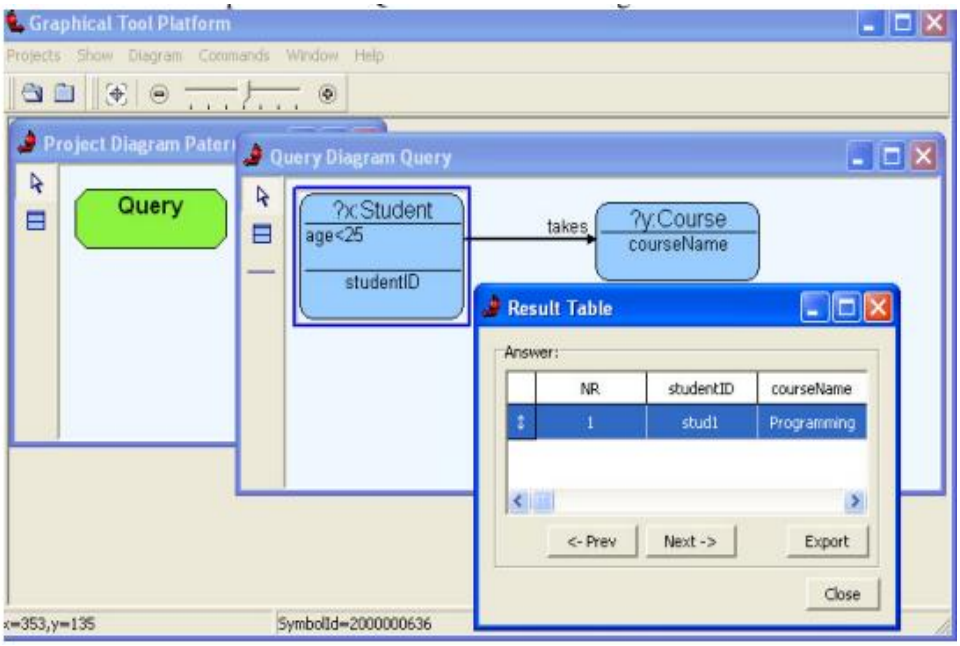

Fig. 50. Viziquer tool [4]

- **Graphical editor for Project Assessment Diagrams** [83] is an editor for visualising business processes regarding the review and assessment of submitted projects. This editor is based on UML activity diagrams and thus contains means for modeling business processes. Yet, some new attributes and some new elements have been added in order to handle the specific needs. This editor is a part of a bigger information system for document flow management (a simplified BPM suite), so services providing interconnection between the system and the editor have been provided for it. For example, the import of the model of the project evaluation diagram to the database is provided, where the external information systems are able to interact with the models created in the editor.
- **Business process editor for the State Social Insurance Agency** [83] – another DSL tool that was created with the GrTP platform. This tool is quite similar to a BPMN editor. However, the tool comes with 3 relatively specific services:
    1. Online collaboration with a relational database –searching for information in a database was to be combined with the graphical tool. The use case of this was a possibility to browse for normative acts during the diagram

design phase – the normative acts are stored in a database and need to be accessed from the tool.

2. Users wanted to start using the tool as soon as possible – even before the language definition had been fully completed. That means it was necessary to ensure the preservation of user-made models while the language could still change slightly. So DSL evolution over time is an issue to be considered.
3. The tool had to provide the ability to create textual reports from graphical information.

It is quite obvious that in the context of the aforementioned platform, transformation performance is very important. In fact, transformations should be executed in a very limited amount of time, before the user has noticed a delay in tool operation. Another requirement that must be met by the language used to write transformations in the context of the above-mentioned platforms is that the language in which transformations are written must be easy to use – such that it is "relatively convenient" for the programmer (toolmaker) to write transformations in this language.

L0 language that is used as a main language for the specification of transformations in GrTP fully satisfies these requirements. This, in turn, allows us to reasonably claim that one of the main goals of the thesis – to offer a low-level model transformation language that has effective implementation and that can be directly used for writing transformations – has been achieved.

CHAPTER 8

Recent Trends in Model Transformation Language Development

8.1 Development of New Imperative Model Transformation Languages

Above (see CHAPTER 4) we had provided a comparison of L0 with other low-level imperative textual transformation languages and had given an overview of related languages. Comparison with ATL byte code has been performed by compiling L0 commands to ATL byte code. The languages that we reviewed had emerged and been published around the same time (or slightly later) as when the L0 language appeared. Since that time, a series of new transformational languages have emerged.

Several studies are dedicated to the analysis and systematization of existing model transformation languages, starting with classical but somewhat dated [57] and more up-to-date but somewhat limited in scope [89, 90]. The research that gives the most comprehensive survey of model transformation languages and related tools and provides a complete picture of the current state-of-the-art and practice in model transformation is given in [91], where authors review 60 model transformation tools and classify and compare them using 45 facets distributed over six categories.

In [91] the authors demonstrate that the boom in model transformations development occurred in the early 2000s, with the emergence of the most transformational languages. Following that, there was a noticeable decline, with the number of newly proposed languages decreasing, and updates for already existing languages becoming less frequent.

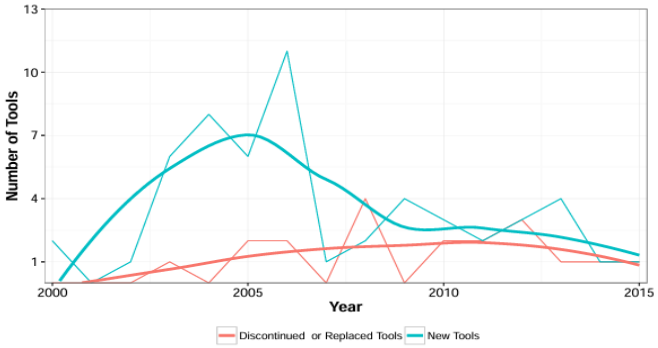

Fig. 51 Number of new and discontinued tools [91]

Starting from 2010, the number of newly proposed model transformation languages was no longer as impressive, but new languages still continued to emerge (see **Fig. 51**). In [91] authors provide a list of transformation languages of various types (relational, imperative, graph-based, hybrid, and template) that appeared after 2010. Concerning specifically imperative languages, from the table below (see **Fig. 52**) we can observe that after 2010, only three new imperative model transformation languages emerged: Xtend [92], Melange [93], and JQVT [94].

As to low-level languages for which effective implementation can be realistically obtained with some moderate resource investment (as it was in case of L0), information about the emergence of such languages is not available.

Table 1: Classification of model-to-model (M2M) transformation tools.

App.	Tool	Description	Lang.	FR	LR
Imperative	ModelAnt [52]	an extension of Apache ANT to support model transformations	Java	2004	2014
	Xtend[53]	statically-typed high-level programming tool for JVM, successor to Xpand	Java	2013	2017
	MetaEdit+ [54]	creates and develops DSM languages	MERL	1993	2017
	QVTo-Eclipse[55]	an Eclipse implementation of Borland Together based on QVTo	Java	2008	2017
	Kermeta2[56]	a meta-programming environment based on a model-oriented language	Java	2005	2012
	Modelio[57]	successor to Objecteering based on UML and BPMN	Java	2009	2017
	Umple[58]	a programming language family for model-oriented programming	Java	2008	2017
	Melange[59]	modular language workbench with re-usability, successor to Kermeta2	Java	2015	2017
	MagicDraw[60]	a visual UML, SysML, BPMN, and UPDM modeling tool	Java	1998	2017
	JAMDA[61]	supports Java code generation from a model of the business domain	Java	2002	2003
	SmartQVT[62]	a partial implementation of the QVTo language	Java	2006	2008
	SiTra[63]	a Java library supporting a Java-based approach to M2M	Java	2006	2012
	Mitra2[64]	successor to Mitra, optimized for semi-automated transformations	Java	2010	2012
	JQVT[65]	based on a compiled QVT engine for Java	Java	2012	2013
	Merlin[66]	based on EMF and Java Emitter Templates (JETs)	Java	2004	2005
	Together[67]	a set of Eclipse plugins to partially implement the QVTo language	Java	2003	2016
	MOFScript[68]	successor to UMT, implements the OMG MOFM2T specification	Java	2006	2011

Fig. 52 List of imperative model transformation languages with release dates (first (F), last(L)) [91]

Further, we give a brief look into fundamental characteristics of these 3 languages.

8.1.1 Xtend

Xtend is an object-oriented extension of the Java general-purpose programming language that is specifically designed for model-to-model (M2M) transformation. It is part of the Eclipse Modeling Framework (EMF) and provides a powerful and expressive syntax for defining transformations between different modeling languages [92].

Xtend's syntax is based on Java. Xtend provides built-in support for EMF metamodels. Xtend integrates seamlessly with the EMF, enabling the use of EMF tools for model management and transformation execution.

In summary, Xtend is a versatile and powerful language for defining model transformations in a Java-like way, particularly in the context of EMF-based projects.

8.1.2 Melange

Melange is a meta-language for modular and reusable development of domain-specific languages (DSLs). It provides a set of language operators that allow designers to define, assemble, and customize DSLs in a structured and composable manner. Melange is tightly

integrated with the Eclipse Modeling Framework (EMF) ecosystem, leveraging the Ecore meta-language for defining DSL abstract syntax and the K3 meta-language for specifying operational semantics [93].

Melange facilitates the creation of modular DSLs by enabling the reuse of language components across multiple DSLs. Melange supports customization of DSLs through language operators that allow for inheritance, merging, slicing, and weaving of language elements.

8.1.3 JQVT

JQVT (Java Query for Versioned Transformations) is a transformation language based on QVT (Query/View/Transformation) that targets the Java type system instead of EMF/Ecore. It provides a declarative approach to model transformation and is primarily used for code generation and manipulation tasks. Unlike Xtend, which is an extension of Java, JQVT is a standalone language with its own syntax and semantics [91].

Here we just cite JQVT description given in [91], because the page that have been containing JQVT docs [94] is now unavailable.

8.1.4 Conclusions

Overall, despite the fact that the number of new languages that emerged after 2010 was significantly smaller than the number of languages that appeared before 2010, it can be noted that new transformation languages continued to emerge.
In particular in the area of imperative languages the emergence of new languages such as JQVT, Xtend, and Melange shows that new languages introduce higher-level abstractions and novel features that enhance expressiveness and readability of transformations.
Unlike traditional low-level model transformation languages that often require intricate and detailed specifications, these newer languages offer a more declarative, concise, and user-friendly syntax. Xtend, for instance, leverages its concise and expressive syntax, inspired by Java, to facilitate model transformations in a manner that is both familiar to developers and conducive to code readability. Melange, on the other hand, stands out for its domain-

specific language (DSL) approach, streamlining the definition and generation of DSLs for model-driven engineering within the Eclipse Modeling Framework (EMF) ecosystem. By adopting an aspect-oriented approach and integrating with Eclipse Xtext, Melange not only simplifies DSL development but also promotes language modularity and extensibility.

When comparing these newer languages with traditional low-level model transformation languages (such as L0), it becomes evident that they operate at a substantially higher level of abstraction and therefore cannot be directly compared.

8.2 Use of Low-level Model Transformation Languages

Above, we provided an overview of the situation with the emergence of new high-level model transformation languages after 2010. The next interesting question is what happened in the field of low-level model transformation languages during this time. It should be noted that we do not have information about the emergence of new low-level model transformation languages in this period. Therefore, we will further review the low-level languages L0 and ATL bytecode, which already existed before 2010. As for the ATC language, we do not have information about the use and development of this language after 2010.

8.2.1 ATL byte code and ATL

When it comes to ATL bytecode itself, its definition has remained unchanged. The direction where changes were noticeable is in the development of ATL virtual machines. Apart from the classical VM, which has been developed in 2005, an EMF VM has been built, demonstrating significantly better performance [95]. The most modern and advanced VM built for ATL is EMFTVM [96]. Its first version was presenteded in 2011.

The EMF Transformation Virtual Machine (EMFTVM) is derived from the current ATL VMs and bytecode format. Instead of using a proprietary XML format, it stores its bytecode as EMF models, such that they may be manipulated by model transformations [96].
Compared to the original VM EMFTVM contains some advanced features such as :

- Compiler defined as higher order ATL transformation
- Advanced tracing facilities
- In-place transformations
- JIT compiler from EMFTVM bytecode to Java bytecode.

Similarly, we observe the emergence of ATL VM, which enhances ATL with new features (e.g., parallel execution of ATL rules) [97].

However, despite all that has been mentioned above and impressive advancement in the area of ATL VM development, we can see that the ATL byte code per se have not seen any changes and the main use case for it has stayed the same – it is used as the pillar of ATL implementation.

8.2.2 L0 and MOLA

Speaking about the advancements of the L0 language, it should be noted that there have been no significant updates since 2010. However, the language itself was actively used. More information about the applications of the L0 language can be found in the relevant sections of this thesis (see CHAPTER 6, CHAPTER 7) and related publications [1, 2, 4, 6, 10, 29, 30, 52, 62, 82, 83, 84], including our publication from 2023 [10], which specifically focuses on an overview of practical applications of L0 language. Here, we will specifically highlight those applications that happened after 2010:

- The work on the GradeTwo tool [84] was continued. L0 was used as the main language for transformation development. These transformations handle user-generated (and other) events and keep domain and presentation models in a synchronized state.
- The L0 language was used in student teaching process at the Faculty of Computer Science of the University of Latvia. It was used to demonstrate model transformation concepts in the Systems Modeling course of the Master's program. Within the Systems Modeling course, students were taught UML modeling languages – class diagrams, activity diagrams and other types of diagrams. While talking about class diagrams, model transformations were also discussed and written down using L0 syntax [10].
- However, the **primary** use case for the L0 language after 2010, is considered to be the fact that L0 has been and continues to be used as the base language for the high-level language MOLA implementation through the bootstrapping process. More information about this MOLA implementation can be found in the MOLA

implementation chapter (see 7.2) and relevant papers [27, 29, 52]. Here, we will emphasize that this type of MOLA implementation has been successful. Throughout these years, MOLA language users have been utilizing MOLA language directly through this implementation, and there has been no need for an alternative implementation. The fact that the MOLA language itself remains relevant is confirmed, for example, by its use in research whose results are presented in scientific papers published in 2022 [98].

8.3 Towards Web-based Tool Building Platforms

One of the applications of L0 was its use in the development of the GrTP platform and the specification of tools, which were built on the basis of the GrTP platform. In this section, we will look at two successors to the GrtP platform that emerged after 2010: ajoo [102] and "Sirius on the web" [99, 100].

Ajoo framework has been developed at the IMCS in 2016. Ajoo is a framework for implementing DSML tools as web applications. The framework consists of DSML tool building platform and the DSML configuration tool. The framework contains a configuration tool that provides means to specify DSML tools using graphical user interface to simplify and accelerate the DSML implementation process. Thus the framework allows implementing a wide area of DSML tools without programming [102].

The first notable difference of ajoo compared to GrTP is that ajoo is a web-based tool, whereas GrTP is a desktop-based tool. This immediately brings several significant advantages over GrTP [102]:

- First, the tool can be accessed from anywhere in the world.
- Second, the online tool solves multi user problem since the collaboration is performed in realtime. In case of DSML tools, it means that a user may edit a diagram and the rest of the collaborators would receive the changes immediately, thus eliminating users messing with complex multi-user mechanisms.
- Third, the system can be used on any device including mobile devices and tablets.

The second notable difference is that the ajoo platform does not rely on the use of model transformation languages for tool specification. In ajoo platform, the new tool is created by graphically specifying mappings from presentation elements to types. If there is any functionality that cannot be expressed through graphical representations, additional features can be implemented using the platform's extension mechanisms, which allow adding missing functionality by writing it in JavaScript.

Another modern web DSL tool building platform that emerged after 2010 is Sirius on the web [99]. Sirius [100] is an Eclipse project which allows you to easily create your own graphical modeling workbench by leveraging the Eclipse Modeling technologies, including EMF and GMF. Sirius, just like ajoo, is a web-based tool. Mappings between represantation and semantic model element are being defined by means of a special configuration file. Similarly to ajoo, Sirius does not rely on the use of model transformations for tool specification. If we are talking about differences between Ajoo and Sirius – the main difference is that Sirius has expilicitly defined domain model, ajoo does not. Tools specification in Sirius start with the specification of domain (semantic element) model. Then representation elements are specified and finally we specify mappings between representation elements and domain model elements.

From the perspective of this thesis, the main difference between ajoo and Sirius compared to Grtp is, of course, that these platforms do not use special transformation languages to define tools. How can this be explained?

In the case of ajoo, this can be explained by the fact that ajoo is tightly integrated with the Meteor framework, which uses MongoDB (a document-oriented database) for data storage. The content of this database cannot be directly processed using model transformation languages because MongoDB does not provide API for model processing.

In the case of Sirius, data is stored as EMF models. In principle, it would be possible to use model transformation languages here. However, this is not done. We failed to find

comments from the developers of Sirius on why they recommend using the Java language instead of model transformation languages for adding additional functionality. However, we assume that this can be explained by several factors:

1. Modern general-purpose programming languages have progressed significantly in their development compared to the situation when the GrTP platform emerged. Currently, many general-purpose programming languages offer powerful features such as declarative constructs (e.g., Java streams), support for functional programming constructs, and more. These constructs can significantly increase developer productivity and make the use of specialized model transformation languages somewhat less appealing.
2. General-purpose programming languages have significantly better tool support: integrated development environments (IDEs), debugging tools, version control, and more.
3. Often, there is a human factor at play, and users simply may not have the desire to learn a new language when they can use a familiar one, even if the new language may be better suited for certain applications.

In summary, it can be said that the development of DSML tools remains relevant. It is evident that the evolution of this kind of tools is still ongoing. Not all new DSML tool development platforms utilize model transformation languages. It would be interesting to see whether specialized universal transformation language designed specifically for DSML tool development field will emerge one day.

8.4 Conclusions

The common trend is that a majority of new languages being developed are high-level languages. Most of these languages are implemented using virtual machines. This includes building virtual machines designed for the execution of the specific language to be implemented, as well as implementing new languages on virtual machines that were originally created for implementation of other high-level model transformation languages

(for example, building implementations of other languages on the basis of ATL VM [96, 103]).

It should be noted that, with rare exceptions (where highly popular languages such as ATL or QVT are concerned), it is somewhat problematic to find research papers exposing technical details of the implementation of the respective model transformation languages. Typically, papers focus on user-oriented features of the languages. When it comes to transformation languages for which implementation information is available, QVT language is certainly notable. Most well known QVT language implementations are summarized in [101].

Table 2-4. QVT Implementers

		SCOPE	FRAMWRK	ENGINE	EMF-Based
Declarative	mediniQVT	(C)	Eclipse	JVM	EMF
	ModelMorf	(C)	---	JVM	NON-EMF
	MOMENT-QVT	(O)	Eclipse	MAUDE	EMF
	Declarative QVT	(O)	Eclipse	ATL-VM	EMF
Imperative	SmartQVT	(A)	Eclipse	JVM	EMF
	Borland QVTO	(C)	Together	Borland QVT engine	EMF
	Procedural QVT	(O)	Eclipse	Borland QVT engine	EMF
	OpenCanarias QVTo	(C)	Eclipse	ATC-VM	EMF

Fig. 53 Variety of QVT implementations [101]

It can be observed that the majority of these implementations are based on virtual machines (VMs). These virtual machines are specifically built for the implementation of particular languages. At the same time, there are studies where new languages are implemented on virtual machines that were originally constructed for other high-level model transformation languages (for example, implementing other model transformations languages on variants of ATL VM).

This approach (where a specific VM is built for the implementation of a specific high-level language) differs from our proposal for model transformation language implementation. We propose building a series of languages, where the first language is a low-level language L0 and each subsequent language gets progressively closer to the high-level language for which the implementation is being built. It is difficult to definitively say which of these approaches is better. Each has its own drawbacks and advantages.

The main advantage of L0 is that this language contains complete set of means for model processing while being very simple (for example, see the comparison of L0 with ATL bytecode in CHAPTER 4). Consequently, the investment of resources required to obtain language implementation in the new target environment will also be lower compared to more complex languages. As an illustration of this claim, it can be mentioned that we are not aware of the aforementioned VM implementation being transferred to other target environments. This differs from the situation with L0, where due to the simplicity of the language, language implementations were created for several target environments. The L0 language was compiled both to Java and to C++. Moreover, in both the case of C++ and Java, there was an option to choose which repository is used to provide model operation processing: in the case of C++, it was either mii_rep or JAPIER, and in the case of Java, it was either EMF or JGRALAB.

Similarly, the simplicity with which L0 implementations can be transferred to new environments and the fact that L0 contains a set of complete means for model processing (thus making it immediately usable as a practical transformation language, not just a target language) makes L0 a good choice in the context of ***model-driven compiling***. In this approach compilation process consists of three basic steps:

- parse an input program and obtain the model of it
- compile the model of the input program to a model of an output program
- generate the code of the output program from the model

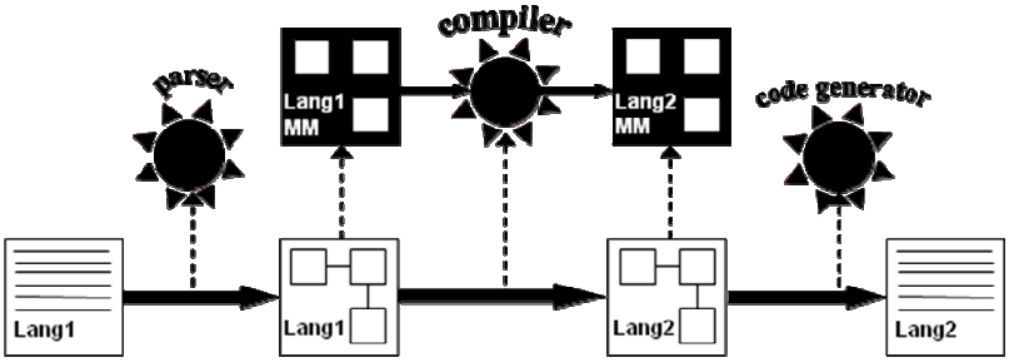

Fig. 54 Model-driven compiling [52]

Models are used as core elements of the compilation process (see figure above). These steps are similar to phases of a compilation in the traditional compilation technique

[31]. The lexical and syntax analysis 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. A model of a source program is stored according to the language metamodel. All three steps of the model-driven compiling require appropriate metamodels already built for both input and output languages and a transformation written using a model transformation language suitable for the compilation tasks [52].

Taking into consideration the statements above, it can be affirmed that the language L0 and the principles we offer for high-level model transformation implementation still retain their relevance.

CHAPTER 9

Conclusions

This thesis is devoted to the problems of the efficient implementation of model transformation languages. Practice shows that the efficient implementation of high-level model transformation language is a non-trivial problem. In contradistinction to traditional programming languages there are no standard solutions for the effective implementation of high-level model transformation languages. One of the main reasons for this is heavy reliance on the usage of patterns in advanced transformation languages. In this thesis we propose a solution to this problem. A hypothesis according to which a possible way to solve the aforementioned problem is to use the bootstrapping method for the implementation of high-level model transformation language was put forward [1]. To practically approbate this idea an efficient **base** language was needed.

One of the most important results of this thesis is the development of a new low-level textual model transformation language L0 that can be used as a base language [1, 2]. This language has several important characteristics: it is very simple (easy to learn) and it contains minimal, but sufficient constructs for the definition of transformations. One more characteristic that is critically important for the base language is efficient implementation. In the context of this work an efficient L0 compiler has been built. This compiler can produce target code in C++ [32] or Java [33] languages. Generated code can be executed on top of the following data stores: mii_rep [44] and JAPIER [41] in the case of C++ or EMF [37] and JGRALAB [43] in the case of Java. When we have an efficient base transformation language, we can practically approbate an idea of implementing higher level model transformation languages through bootstrapping. This idea has been applied for the implementation of high-level model transformation language MOLA in the context of A. Šostak's dissertation and has confirmed its effectiveness in practice [52].

A number of experimental additions for L0 language have also been provided: the L0` language and L0+ language. Language L0` is L0 language supplemented with undo/redo options. L0+ language, in turn, is an extension of the L0 language, which allows operations not at the model level, but at the metamodel level (for example, adding a class or

association, etc.). On the example of these language extensions, it is shown how L0 language can be supplemented with new features.

Outside the domain of model-based compiling there are use cases for L0 language as well. The proposed L0 language is used in the context of a graphical tool building platform GrTP [18] developed at LU IMCS. In GrTP, specification of an internal logic of a specific tool is performed solely by model transformations written in L0. The total size of the source code of transformations developed in this project exceeds 20,000 lines of L0. The efficiency of L0 implementation can be confirmed by the fact that, for example, in the context of the aforementioned tool building platform, pasting of a collection consisting of 1000 elements into the diagram takes less than 5 seconds. Likewise, by using L0, several DSL tools were developed, which are used in practice [4, 83, 84, 85]. This proves that the language is suitable to be used not only as a target language in the bootstrapping process, but also as an independent transformation language for direct transformation specification.

Another important result of this thesis is a series of experiments, where the model transformation approach has been used in a new domain – migration of relational data bases to RDF [6, 9]. The proposed approach is based on L0 language. Application of model transformation in this domain is quite challenging because the amount of data to be processed is substantially larger than in the case of tool building platforms (e.g., GRTP). In the first experiment, migration was based on L0 implementation that uses JAPIER repository for model processing operations. In the second experiment, migration was based on L0 implementation that uses a relational data base for model processing operations. Finally, in the third experiment, migration was performed without using a repository – all models were stored in RAM according to specific model coding principles proposed in section 6.4.1. The results of the experiments show that the best migration process performance is achieved when the migration process is based on implementation that stores the model in RAM (repository is not used). The performed migration experiments have confirmed the practical applicability of the proposed migration methods.

In general it can be considered that all the defined research tasks have been completed:

- a new low-level textual model transformation language L0 has been developed and effectively implemented,
- proposed principles of implementation of high-level model transformation languages through the bootstrapping process (based on L0 language) have been successfully approbated in practice,
- 2 experimental additions to the base language have been proposed: L0`and L0+
- a proposed base transformation language is well suited for real practical applications and successfully approbated in practice,
- a new model transformation based method for migration of relational DB to RDF has been developed and practically approbated.

Therefore, it can be concluded that the main goal of the work – to offer a solution to the problem of effective implementation of model transformation languages and to test this solution in practice – has been achieved. Both proposed hypotheses were confirmed in practice:

- we created a low-level language L0 and its effective implementation. L0 language was both used as an independent language and as a base language in the process of higher-level language implementation through the bootstrapping method
- we demonstrated that it is possible to use model transformation languages for migration of relational data bases to RDF.

BIBLIOGRAPHY

1. J. Barzdins, A. Kalnins, E. Rencis, S. Rikacovs, ***Model Transformation Languages and their Implementation by Bootstrapping Method***. Lecture Notes in Computer Science, Springer Verlag, 2008, pp. 130-145.

2. S. Rikacovs, ***The base transformation language L0+ and its implementation***, Scientific Chapters, University of Latvia, "Computer Science and Information Technologies", 2008, pp. 75-102.

3. J. Barzdins, S. Rikacovs, ***Towards a Seed Transformation Language and Its Implementation.*** Models 2008, Doctoral Symposium, ETH Zurich, 2008, pp. 27-32.

4. G. Barzdins, S. Rikacovs and M. Zviedris, ***Graphical query language as SPARQL frontend***, ADBIS 2009, Local Proceedings, 2009, pp. 93-107.

5. G. Barzdins, S. Rikacovs, M. Veilande, M. Zviedris, ***Ontological Re-engineering of Medical Databases***, Proceedings of the Latvian Academy of Sciences. Section B. Natural, Exact, and Applied Sciences., 2009, Versita, Warsaw, pp. 156-158.

6. S. Rikacovs, J. Barzdins, (2010) ***Export of Relational Databases to RDF Databases: a Case Study***, P. Forbrig and H. Günther (eds.), in proc. of Business Informatics Research (BIR 2010), Springer LNBIP 64, pp. 203-211.

7. S. Kozlovics, E. Rencis, S. Rikacovs, K. Cerans, ***Universal UNDO Mechanism for the Transformation-Driven Architecture***, DBIS 2010. In Proceeding of the Ninth International Baltic Conference, DB&IS 2010, pp. 325-340, 2010.

8. S. Kozlovics, E. Rencis, S. Rikacovs, and K. Cerans ***A kernel-level UNDO/REDO mechanism for the Transformation-Driven Architecture***. In Databases and Information Systems VI – Selected Papers from the Ninth International Baltic Conference, DB&IS 2010, pp. 80-93, 2010.

9. S. Rikacovs, ***Export of Relational Databases to RDF Databases by Model Transformations***, in proc. of Business Informatics Research (BIR 2011), Springer LNBIP 90, pp. 158-166.

10. S. Rikačovs, L. Lāce, E. Rencis, A. Šostaks, K. Čerāns. ***An Overview of Practical Applications of Model Transformation Language L0***. Baltic Journal of Modern Computing, Vol. 12 (2024), No. 1, pp. 1-14,

11. A. Kleppe, J. Warmer, W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*, Addison Wesley, 2003.

12. J. Warmer, A. Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA, Second Edition* , Addison Wesley, 2003.

13. OMG. Meta Object Facility (MOF) Specification Version 2.0 URL: http://www.omg.org/docs/formal/06-01-01.pdf
14. OMG. Meta Object Facility (MOF) Specification Version 1.4.1 URL: http://www.omg.org/docs/formal/05-05-05.pdf

15. OMG. Request for Proposal: MOF 2.0 Query / Views / Transformations RFP, URL: http://www.omg.org/docs/ad/02-04-10.pdf

16. OMG. MOF 2.0 Query/View/Transformation Specification. URL: http://www.omg.org/docs/ptc/07-07-07.pdf

17. OMG. UML 2.0 OCL Specification, URL: http://www.omg.org/docs/ptc/03-10-14.pdf

18. Bārzdiņš, J., Zariņš, A., Čerāns, K., Kalniņš, A., Rencis, E., Lāce, L., Liepiņš, R., Sproģis, A. GrTP: Transformation Based Graphical Tool Building Platform. In: MoDELS'07,Workshop: Model Driven Development of Advanced User Interfaces (MDDAUI-2007), available at http://ceur-ws.org, Vol. 297.

19. Willink E., A concrete UML-based graphical transformation syntax – The UML to RDBMS example in UMLX. Workshop on Metamodelling for MDA, University of York, England, 24-25 November 2003.

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, November 2003.

21. ATL. URL: http://www.sciences.univ-nantes.fr/lina/atl/

22. Tefkat. URL: http://tefkat.sourceforge.net/

23. VIATRA2 URL: http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/VIATRA2/index.html

24. AGG – The Attributed Graph Grammar System. URL: http://tfs.cs.tu-berlin.de/agg/

25. A. Kalnins, J. Barzdins, E. Celms. Model Transformation Language MOLA. – *Proc. MDAFA 2004 (Model-Driven Architecture: Foundations and Applications 2004)*, Linkoeping, Sweden, pp. 14-28.

26. 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, 14-18 June 2004.

27. 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)*, 2006, pp. 159-167.

28. A. Kalnins, 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, p. 14.

29. A. Sostaks., A. Kalnins. The implementation of MOLA to L3 compiler, Scientific Papers University of Latvia, "Computer Science and Information Technologies", 2008.

30. E. Rencis. Model Transformation Languages L1, L2, L3 and their Implementation, Scientific Papers. University of Latvia, “Computer Science and Information Technologies”, 2008.

31. A.V. Aho, R. Seti, J. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison Wesley, 1986.

32. B. Stroustrup, *The C++ Programming language*, 3rd edition, Addison Wesley, 1997.

33. Ken Arnold, James Gosling, David Holmes, Java(TM) Programming Language, The (4th Edition), Addison-Wesley Professional, 2005.

34. S. Bechhofer, F. Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, L. A. Stein. OWL Web Ontology Language Reference. URL: http://www.w3.org/TR/owl-ref/

35. Fischer, T., Niere, J., Torunski, L., Zündorf, A. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Proceedings of TAGT. Volume 1764 of LNCS., Springer (1998) 296-309 Rīks: University of Paderborn. Fujaba Tool Suite. http://www.fujaba.de

36. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.: Eclipse Modeling Framework. Addison Wesley, 2003.

37. Eclipse Modelling Framework URL: http://www.eclipse.org/emf/

38. Metadata Repository (MDR). URL: http://mdr.netbeans.org/

39. Sesame URL: http://www.openrdf.org/

40. J. Broekstra, A. Kampman, F.V. Harmelan. *Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema*. Proc. International Semantic Web Conference, Sardinia, Italy, 2002.

41. M. Opmanis, K. Cerans. JR: A Multilevel Data Repository. Proceedings of the 9th International Baltic Conference on Databases and Information Systems (Baltic DB&IS 2010), Riga, Latvia, 5-7 July 2010, pp. 375-390.

42. 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.

43. S. Kahle. JGraLab: Konzeption, Entwurf und Implementierung einer Java-Klassenbibliothek für TGraphen. Diplomarbeit, University of Koblenz-Landau, Institute for Software Technology, 2006.

44. Modelēšanas rīku informācijas glabāšanas līdzekļu (repozitorija) programmu saskarnes documents (MII_REP). URL: http://www.gradetools.com/new/deliver/mii_rep_API_apraksts.doc

45. G. Hillairet, F. Bertrand, and J.-Y. Lafaye, MDE for Publishing Data on the Semantic Web, in Proceedings of the First International Workshop on Transforming and Weaving Ontologies in Model Driven Engineering (TWOMDE 2008), 2008, pp. 32-46.

46. Sahoo, S.S., Halb, W., Hellmann, S., Idehen, K., Thibodeau Jr, T., Auer, S., Ezzat, A.: A survey of current approaches for mapping of relational databases to RDF. Technical report, W3C RDB2RDF Incubator Group (January 2008)

47. D2RQ Platform. http://www4.wiwiss.fu-berlin.de/bizer/D2RQ/spec/

48. C. Blakeley: RDF Views of SQL Data (Declarative SQL Schema to RDF Mapping), OpenLink Software, 2007.

49. W. Hu, Y. Qu: Discovering Simple Mappings Between Relational Database Schemas and Ontologies", In Proceedings of 6th International Semantic Web Conference (ISWC 2007), 2nd Asian Semantic Web Conference (ASWC 2007), LNCS 4825, pp. 225-238.

50. K. Čerāns and G. Būmans. RDB2OWL: a RDB-to-RDF/OWL Mapping Specification Language. In Proceeding of the 2011 conference on Databases and Information Systems VI: Selected Papers from the Ninth International Baltic Conference, DB&IS 2010, Janis Barzdins and Marite Kirikova (Eds.). IOS Press, Amsterdam, The Netherlands, pp. 139-152.

51. E. Vysniauskas, L. Nemuraite and A. Sukys. A Hybrid Approach for Relating OWL 2 Ontologies and Relational Databases (2010), P. Forbrig and H. Günther (eds.), in proc. of Business Informatics Research (BIR 2010), Springer LNBIP 64, Germany, Rostock, pp. 86-102.

52. Šostaks, A.: IMPLEMENTATION OF MODEL TRANSFORMATION LANGUAGES. Ph.D. thesis, University of Latvia (2010)

53. D.J. Armstrong, The quarks of object-oriented development, Communications of the ACM, 2006.

54. M. Zviedris, G. Barzdins. ViziQuer: A Tool to Explore and Query SPARQL Endpoints, Lecture Notes in Computer Science, 2011, Volume 6644/2011, pp. 441-445.

55. G. Klyne, J. J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. URL: http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/

56. SPARQL Query Language for RDF, http://www.w3.org/TR/rdf-sparql-query

57. K. Czarnecki, S. Helsen. Classification of model transformation approaches, Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture, 2003, pp. 1-17.

58. F. Jouault, I. Kurtev. *Transforming Models with ATL*. Proc. of the Satellite Events at the MoDELS2005 Conference. LNCS, Vol. 3844, Springer-Verlag, 2006, pp. 128-138.

59. Jouault, F., Allilaire, F. An introduction to the ATL Virtual MachineV1.0 draft (online, 20.06.2010) http://www.eclipse.org/m2m/atl/doc/ATL_VM_Presentation_%5B1.0%5D.pdf

http://www.eclipse.org/atl/documentation/old/ATL_VM_Presentation_[1.0].pdf

60. Specification of the ATL Virtual Machine. URL: http://www.eclipse.org/atl/documentation/old/ATL_VMSpecification[v00.01].pdf

61. ACG – ATL Code Generation Language. URL: http://wiki.eclipse.org/ACG

62. Rencis, E.: UZ MODEĻU TRANSFORMĀCIJĀM BALSTĪTU RĪKU BŪVES METOŽU IZSTRĀDE UN REALIZĀCIJA. Promocijas darbs, Latvijas Universitāte (2012)

63. D. Kolovos, R. Paige, F. Polack: The epsilon object language (eol)., Model Driven Architecture–Foundations and Applications LNCS 4066(2006), pp. 128-142.

64. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Model Comparison: A Foundation for Model Composition and Model Transformation Testing. In: Proc. 1st International Workshop on Global Integrated Model Management (GaMMa), ACM/IEEE ICSE 2006, Shanghai, China, pp. 13-20. ACM Press (2006).

65. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Merging Models with the Epsilon Merging Language (EML). In: Proc. ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (Models/UML 2006), Genova, Italy. LNCS (October 2006).

66. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages. In: Proc. Dagstuhl Workshop on Rigorous Methods for Software Construction and Analysis (2007).

67. Kolovos, D.S., Paige, R.F., Rose, L.M., Polack, F.A.C.: Update Transformations in the Small with the Epsilon Wizard Language. Journal of Object Technology (JOT), Special Issue for TOOLS Europe 2007 (2007).

68. Rose, L.M., Paige, R.F. , Kolovos, D. S., Polack, F.A.C.: The Epsilon Generation Language. Model Driven Architecture – Foundations and Applications LNCS Volume 5095, 2008, pp 1-16.

69. Kolovos, D.S., Paige, R.F., Polack, F.A.C. The epsilon transformation language. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, Vol. 5063, pp. 46-60. Springer, Heidelberg (2008)

70. RDF 1.1 JSON Alternate Serialization (RDF/JSON). URL: https://www.w3.org/TR/rdf-json/

71. Barzdins, G., Barzdins, J., Cerans, K.: From Databases to Ontologies, Semantic Web Engineering in the Knowledge Society. In: Cardoso, J., Lytras, M. (eds.) IGI Global, pp. 242-266 (2008) ISBN: 978-1-60566-112-4.

72. A. Estévez, J. Padrón, E.V. Sánchez , J.L. Roda.: ATC: A Low-Level Model Transformation Language. In: MDEIS 2006: Proceedings of the 2nd International Workshop on Model Driven Enterprise Information Systems, May 2006, pp. 64-74. ISBN 978-972-8865-56-6.

73. A. Sánchez-Barbudo, E.V. Sánchez, V. Roldán, A. Estévez, J.L. Roda. Providing an Open Virtual-Machine-based QVT Implementation. In Proceedings of the V Workshop on Model-Driven Software Development. MDA and Applications (DSDM'08 – XIII JISBD), 2008.

74. GATC. URL: http://www.modelset.es/atc/atcdownload.html

75. Kolovos. D.: An Extensible Platform for Specification of Integrated Languages for Model Management. Ph.D. thesis, The University of York, 2008.

76. The Epsilon Book. URL: http://www.eclipse.org/epsilon/doc/book/

77. RDF/XML Syntax Specification. W3C. URL: https://www.w3.org/TR/rdf-syntax-grammar/

78. RDF 1.1 Turtle: Terse RDF Triple Language. W3C. URL: https://www.w3.org/TR/turtle/

79. N3:Notation3 (N3): A readable RDF syntax. W3C. URL: https://www.w3.org/TeamSubmission/n3/

80. RDF 1.1 N-Quads. URL: https://www.w3.org/TR/n-quads/

81. RDF 1.1 N-Triples: A line-based syntax for an RDF graph. W3C. URL: https://www.w3.org/TR/n-triples/

82. G. Barzdins, E. Liepins, M. Veilande, M. Zviedris, "Ontology Enabled Graphical Database Query Tool for End-Users", Selected papers from DBIS'2008, Hele-Mai Haav (Eds.), Frontiers in Artificial Intelligence and Applications series, IOS Press, 2009. 187:105-116.

83. Barzdins, J., Cerans, K., Grasmanis, M., Kalnins, A., Kozlovics, S., Lace, L., Liepins, R., Rencis, E., Sprogis, A. and Zarins, A., 2009, October. Domain specific languages for business process management: a case study. In Proceedings of DSM (Vol. 9, pp. 34-40).

84. GradeTwo. URL: http://gradetwo.lumii.lv/

85. Zviedris, M.: Dati kā ontoloģija - glabāšana, vaicāšana, vizualizācija. Promocijas darbs, Latvijas Universitāte (2014). URL: http://dspace.lu.lv/dspace/handle/7/4970

86. S. Das, S. Sundara, and R. Cyganiak. R2RML: RDB to RDF Mapping Language. W3C Recommendation, W3C, Sep. 2012. URL: https://www.w3.org/TR/r2rml/

87. Calvanese, D., Cogrel, B., Kalayci, E.G., Komla-Ebri, S., Kontchakov, R., Lanti, D., Rezk, M., Rodriguez-Muro, M. and Xiao, G., 2015. OBDA with the Ontop Framework. In SEBD (pp. 296-303).

88. Kharlamov, E., Jim´enez-Ruiz, E., Zheleznyakov, D., Bilidas, D., Giese, M., Haase, P., Horrocks, I., Kllapi, H., Koubarakis, M., Oz¸cep, ¨ O.L., Rodriguez-Muro, M., ¨ Rosati, R., Schmidt, M., Schlatte, R., Soylu, A., Waaler, A.: Optique: Towards OBDA systems for industry. In: Revised Selected Papers of ESWC 2013 Satellite Events. Lecture Notes in Computer Science, Vol. 7955, pp. 125-140. Springer (2013).

89. S. Hidaka, M. Tisi, J. Cabot, Z. Hu, Feature-based classification of bidirectional transformation approaches, Software and Systems Modeling 15 (3) (2015) 1–22

90. C.Gomes, B.Barroca,V.Amaral, Classification of model transformation tools: Pattern matching techniques, 2014, pp.619–635.

91. Kahani, N., Bagherzadeh, M., Cordy, J. R., Dingle, J., & Varrō, D. (2019). Survey and classification of model transformation tools. Software and Systems Modeling, 18(37), 575-598.

92. Xtend, URL: https://eclipse.org/xtend/index.html.

93. T. Degueule, B. Combemale, A. Blouin, O. Barais, J. Jezequel, Melange: A meta-language for modular and reusable development of DSLs, in: 8th International Conference on Software Language Engineering (SLE), 2015, pp. 65–75.

94. JQVT, URL: https://sourceforge.net/projects/jqvt/.

95. ATL VM performance, URL: https://wiki.eclipse.org/ATL_VM_Testing

96. D. Wagelaar, M. Tisi, J. Cabot, F. Jouault. Towards a General Composition Semantics for Rule-Based Model Transformation. In: MODELS '11: Proceedings of the 14th international conference on Model driven engineering languages and systems, pages 623--637, 2011. Springer Berlin Heidelberg.

97. F. Jouault, D. Wagelaar, M. Tisi. Parallel Execution of ATL Transformation Rules. In: Proceedings of the 16th international conference on Model-driven engineering languages and systems, volume 8107, pages 40--54, 2014. Springer International Publishing.

98. Kamil Rybiński, Michał Śmiałek. Beyond Low-Code Development: Marrying Requirements Models and Knowledge Representations. In 17th Conference on Computer Science and Intelligence Systems (pp. 919-928)

99. Viyović, V., Maksimović, M., & Perisić, B. (2014, July). Sirius: A rapid development of DSM graphical editor. In IEEE 18th International Conference on Intelligent Engineering Systems INES 2014 (pp. 233-238). IEEE.

100. Sirius documentation. URL: https://eclipse.dev/sirius/doc/

101. J.M.V.Mesa, M2DAT: A technical solution for model-driven development of web information systems, in: PhD Thesis, University of Rey Juan Carlos, 2009.

102. Sprogis, A. ajoo: WEB Based Framework for Domain Specific Modeling Tools. In DB&IS (Selected Papers) (pp. 115-126). 2016

103. Kling, Wolfgang, et al. "MoScript: A DSL for querying and manipulating model repositories." Software Language Engineering: 4th International Conference, SLE 2011, Braga, Portugal, July 3-4, 2011, Revised Selected Papers 4. Springer Berlin Heidelberg, 2012.