Baltic J. Modern Computing, Vol. 4 (2016), No. 4, 934-964
http://dx.doi.org/10.22364/bjmc.2016.4.4.21

LINQ as Model Transformation Language for MDD

Audris KALNINS[1], Elina KALNINA[1], Agris SOSTAKS[1],
Edgars CELMS[1,2], Ivans TABERNAKULOVS[2]

[1] Institute of Mathematics and Computer Science, University of Latvia,
Raiņa bulvaris.29, Riga, LV-1459,
[2] Faculty of Computing, University of Latvia, Raina bulvaris 19, Riga, LV-1536, Latvia

audris.kalnins@lumii.lv, elina.kalnina@lumii.lv,
agris.sostaks@lumii.lv, edgars.celms@lumii.lv,
ivan.tabernakulov@gmail.com

Abstract. Model Driven Development (MDD) witnessed a boom both in research and practical applications starting from 2001. Its main innovative idea was the model transformation languages as a real support for MDD. A couple of years ago MDD survived a deep crisis due to disillusionment by practitioners in support for it. However, recently it got a revival in the form of Domain Specific Languages, this again stimulated the research and usage of transformation languages. One of the obstacles for MDD adoption in industry has been the reluctance of practitioners to learn a new language. This paper offers a new approach how the well-known extension LINQ of the popular C# language can be used for defining model transformations. Especially the functional style supported by LINQ is promoted, which has not been widely used in the existing transformation languages.

Keywords. LINQ, Model transformations, MDD, DSL.

Introduction

One of the cornerstones for Model Driven Development (MDD) of software is model transformations, which require specific languages and tools for their development. There are many such model transformation languages, but they are not well adopted by software development practitioners. This paper offers a new approach how the popular LINQ extension of the main Microsoft software development language C# can be used for model transformation development as well. In practice LINQ is mainly used for querying and modifying relational databases in a functional style. Particularly this well supported functional style – lambda expressions for inline function definition and facilities for function composition in LINQ are the main value also for a new style of transformation definition which could be quite familiar for practitioners.

The first section of the paper discusses a brief history of MDD and its current status. The crucial role of model transformations and their support for MDD is emphasized there. Section 2 unveils the place of Microsoft technologies - .NET, C# and LINQ in software development. The existing support for MDD in the C# technical space is covered there as well. A new architecture is offered which permits to use LINQ as a model transformation language, by integrating LINQ technical space to typical technical spaces for model transformation development. Section 3 evaluates the expressiveness of LINQ for some typical model transformation tasks, especially emphasizing the functional style. Two tasks of various kind and complexity are solved in LINQ. One the tasks is the diagrammatic visualization how a business process execution advances. The second task is a model transformation which transforms a Petri net into a functionally equivalent statechart. Section 4 provides a general comparison of LINQ to some popular model transformation languages. Section 5 compares LINQ features to the support of functional style in Java included in the version 8 of Java.

1. State of the Art in Model Driven Development

1.1. Initial Stage of Model Driven Development

Models and modelling are one of the most important and most used new paradigms in the software development in this century. Though some kinds of models for software development were used starting from the nineties, a wide usage of models was enabled by the appearance of Unified Modeling Language (UML). The first usable version of UML – UML 1.1 – was published by OMG in 1998 (OMG, 1997 a). It was a reasonable unification of ideas coming from the previous graphical modelling notations such as OMT by J. Rumbough et al (1991) for capturing the ground principles of object-oriented design, entity-relationship (E-R) models by Chen (1976) for database design and some other notations. UML with its vast set of diagrams kinds – class diagrams, use case diagrams, sequence diagrams, activity diagrams etc. could support all phases of software development. In addition, OMG published also the MOF (Meta-Object Facility) (OMG, 1997 b), introducing formally the metamodel concept and four MOF layers. But all this was oriented towards a more formal and precise definition of UML model syntax via metamodels. However the principles how such a wide spectrum of models should be used for software development were not present in UML. Therefore OMG in 2000 came with another crucial initiative – the Model Driven Architecture (MDA) (Siegel and OMG, 2001). MDA for the first time introduced a sequence of models to be used for software development. These models were CIM (Computation Independent Model, a sort of software requirements specification), PIM (Platform Independent Model, the logical design specification of the software system) and PSM (Platform Specific Model, the software implementation specification in the given target development platform) (OMG, 2003). At that time the main goal for MDA was set to create some automation for the transition from a PIM of a system to several PSMs related to different platforms. This goal required the introduction of the most important new technology for processing models – model transformations. It was expected that an automatic transformation could create a PSM for the chosen platform from the PIM of the software system. Later on a goal of at least partially automatic transition from PSM to the real system code in the given platform was added.

To achieve these goals OMG initiated a request for creating an appropriate language for developing transformations – a model transformation language (OMG, 2002). This language should be tightly related to the MOF standard – models to be transformed were built according to the corresponding metamodels (the metamodels strictly specified the syntactic structure of models). Thus PIM models were assumed to be built according to the standard UML metamodel, but PSMs could use an extension of UML via specific profiles. Due to all this, the expected transformation language was named MOF QVT (Queries, Views, Transformations), since initially also queries and model views were expected to be supported (in fact, later only the model-to-model transformation definition remained as a significant goal).

This request for transformation languages by OMG created a boom of various model transformation languages created by a number of research teams. First, it was recognized that the existing earlier graph transformation languages could be reused as MOF model transformation languages – they have been built for transforming (rewriting) graphs with typed and attributed nodes. Such graphs in fact are close to MOF models. Since graphs are simpler and more formal mathematical objects, even some formal graph transformation theory (based on algebra and category theory) had been developed. Thus a number of model transformation languages based on graph transformations soon appeared – AGG (Taentzer et al., 2005), PROGRES (Schurr et al., 1999), GreaT (Agrawal et al., 2003), Viatra (Taentzer et al., 2005), TGG (Konigs, 2005) etc. Two very important basic concepts in transformation languages were taken over from graph transformations – the concepts of rule and pattern. The pattern is a declarative description of subgraphs to be located in the source graph. A rule contains a source pattern (to be located, i.e. matched) and a target pattern – specifying how the located source subgraph should be modified. Typically there is no execution control structure superimposed over the rules – the rules are just let to be applied to the source graph in an arbitrary order, while there is a rule whose source pattern can be matched. Many model transformation languages share the same basic principles, however, some explicit procedural rule execution control is typically added.

Several new model transformation languages appeared soon as well. The most prominent ones are ATL (Jouault and Kurtev, 2005), Tefkat (Lawley and Steel, 2005), UMLX (Willink, 2003), MTF (IBM, 2004), ATOM[3] (de Lara and Vangheluwe, 2002), BOTL (Braun and Marschall, 2003), Fujaba (Fischer et al., 2000), Epsilon (Kolovos et al.., 2016). It should be noted that the OMG language MOF QVT language was significantly delayed, the first version appeared only in 2008 (OMG, 2008). The System Modelling Lab (SysLab) team at IMCS UL also participated in this “transformation language race” and created the MOLA language (Kalnins et al., 2005) in 2004, together with simpler support languages Lx (Rencis, 2008). A transformation language itself may be textual or graphical, e.g. ATL, Tefkat, MTF are textual, MOLA and Fujaba are graphical languages.

Certainly, the languages should not only be defined, but also implemented. The implementation firstly requires a model repository. In general this repository should be MOF-compliant, since most models are built according to MOF standard. The most popular such repository, no doubt, is the Java-based Eclipse EMF (Steinberg et al., 2008), which was created in due time for the transformation language boom, and it is supported by the largest number of transformation languages. There are also some other usable repositories, such as MDR (WEB, a) and GraLab (Dahm and Widmann, 2003) – both also Java based. Secondly, for a transformation language to be usable, a complete tool support is required (as for any programming language) – a classical style IDE for textual languages or graphical editor for graphical languages, a compiler or interpreter, debugging facilities, service facilities for transformed model management, integration with modelling tools etc. The quality of tool support for different languages significantly differs. Probably the best supported language is ATL, which has also gained the widest industrial usage. The MOLA support is also at a required level. Probably the worst supported language paradoxically is MOF QVT.

Due to such a vast support for model transformations, initially there was a sufficient interest in software industry to use MDA in practice. However, soon it became clear that the basic goal setting of MDA is not completely appropriate for practice. Even the role of CIM was not sufficiently clearly set there. The industry also wanted to get tangible results, namely, if models are used, as much as possible code should be generated automatically from them. Thus other approaches appeared – also based on MOF compliant models and model transformations, but not tied to the fixed set of models in MDA. The most popular alternative is MDD (Model Driven development) or a very similar MDSD (Model Driven Software Development). There a set of created models can be chosen in a most appropriate way for the given software development domain, in order to get maximum benefit from the final code generation step. This approach is tightly related to another paradigm – UML and its profiles frequently is not the best modelling notation for the given domain. There domain specific modelling languages (DSML) frequently are used in MDD. Such languages may be standardized, such as BPMN or SPEM for process modelling, but for narrower domains custom DSMLs typically are created. This is especially typical for embedded or real-time software development. Usage of custom DSMLs requires an additional tool support – for model creation custom graphical editors are needed. This is another area for MDD application, see more in Section 1.3.

With MDD the developed models become true development artefacts the same way as the developed code – the code is directly obtained from models. Therefore the models should be maintained as carefully as the code, for instance, versioned, etc. Thus models become a true part of software engineering process. This approach is named MDE – Model Driven Engineering. This is the most inclusive of all model related development approaches, besides software engineering it applies also data engineering, system engineering. This concept was mainly coined by J. Bezivin (2012), however no very precise definition of it is provided. It is more like a maximal use of models in all software development related activities. Certainly, software industry has expected more benefits from this most comprehensive approach, but, unfortunately, the expectations have not come true.

1.2. MDA Crisis

Starting from 2011 the first signs of industry dissatisfaction with the practical use of models in software development have emerged. This happened 10 years after the start of MDA initiated by OMG. The most significant signal was the keynote “Why did MDE Miss the Boat” by J. Bezivin (2011), the main promoter of MDE, at SPLASH 2011 conference, later on repeated at several smaller conferences. The main message there was that the usage of MDE methodology and relevant tools in software industry has reached a standstill. Several reasons for this are mentioned – there are no convincing success stories of MDE application in industry, there is no clear methodology how MDE should be applied, most users still associate MDE with its original base MDA (the fixed set of models CIM-PIM-PSM, UML as the modelling language etc), metamodels on which MDE is based are too complicated for practitioners, there is no adequate support for model evolution. However, the conclusion is rather optimistic – if these issues would be removed, MDE could regain value in practice. Similar views are expressed by J. Den

Haan, the scientific leader of the Mendix company building software tools for MDD support in his 2011 presentation “Why there is no future for Model Driven Development” (den Haan, 2011). His analysis includes also the use of DSMLs for MDD – and as the main reason the lack of adequate support for evolution of models in a DSML and especially the DSML itself is mentioned. It occurs in practice that maintenance of models is more difficult than the maintenance of code and can outweigh the advantages in development. However his final conclusions are optimistic as well. Similarly, according to Gartner hype cycle reports on technologies for software development (WEB, b), MDA was at the cycle peak till 2010, slid into trough from 2011 to 2013 and then disappeared at all from the Gartner technology list (unfortunately, Gartner reports have not considered MDD or MDE).

This disillusion with MDD in software industry has had also an impact on research in the area. This is especially visible in research on model transformation languages. A certain indicator for this can be the topics of papers in annual ICMT conferences which are dedicated exclusively to transformation language development, implementation, support and usage. Due to lack of significant new applications in industry also the interest in providing new transformation principles and new languages has dropped significantly in years 2012 – 2014. The main topics investigated were “internal problems” – transformation testing, validation, evolution. Only in the last two years the situation has revived again, see next section.

1.3. New Stage in MDD – a Consistent Use of DSLs

As it was already mentioned, an innovative aspect in MDD (when compared to MDA) was the usage of DSMLs instead of UML for creating models adjusted to the chosen domain. But nevertheless these models were mainly used as steps in the software development process. Recently in several software development domains Domain Specific Languages – DSLs have gained popularity as direct software development tools. Software systems are created in these languages instead of general-purpose languages (GPLs), and the source code is directly compiled to executable units or to some GPL. Typical examples of such domains are various embedded or real-time software systems, e.g. in automotive industry, avionics, telecommunication. The usage is based on the fact that the software there is mainly created by domain experts familiar with the hardware to be controlled, and precise engineering design notations (languages) frequently exist there. One such example is DSLs for the Autosar project (Koenemann and Nyssen, 2013) . There several DSLs for building components of automotive software have been created (Graf and Voelter, 2009). A typical example of DSL for telecommunication is the Flick (Sultana, 2015) for programming non-standard routers. DSLs can be used also for some more traditional software development tasks. In the ReDSeeDS Fp 6 ICT project (WEB, c), in which the SysLab team also participated, the original approach was a traditional MDD-style chain of models, where a special RSL language was used for creating an extended formal CIM model containing system requirements. MOLA language was used to transform one model to the next in the chain, but the transformed model should also be completed manually (Kalnins et al., 2009). Thus a certain part of Java code could be generated for simple information systems. After the end of the project the leading team of the project from WUT under M. Śmialek decided to modify the goal and extended the RSL language even more by adding typical user interface elements, more precise business logic description and data persistence aspects. In the result RSL grew into a full-fledged DSL for simple web-based information systems from which a major part of implementation in Java could be directly generated (Śmiałek and Nowakowski, 2015).

Such DSLs can be both graphical and textual. Though DSL users see it as a normal programming environment, in most cases the support for such DSL is essentially based on models. This includes both graphical or textual editors and the language translation. Especially the graphical editors are built using model based graphical tool frameworks such as Eclipse GMF (Gronback, 2009) or MetaEdit (Kelly and Tolvanen, 2008). One such framework – GrTP/TDA (Barzdins et al., 2008) has been developed also at SysLab IMCS. Such frameworks typically use also model transformations internally, in case of TDA an original transformation language lQuery (Liepins, 2011, 2012) based on the functional programming language Lua (Ierusalimschy et al., 2007) is used. It should be noted that tasks to be solved by transformations in graphical tool context frequently are different from those in typical MDD – they are mainly model navigations and local in-place updates. The DSL translation process typically also uses model transformation languages and is performed in several steps each having a different metamodel, therefore they are closer to traditional MDD tasks. However, the final step there is model-to-code transformation requiring another kind of languages – for model-to-text transformation (here we will not discuss this kind of languages closer).

Thus the DSL tool building has again raised a need for practical use of model transformations. However here the tasks to be solved are more variable. This has raised the need for model transformation languages to become domain specific as well. A classic example here is the Epsilon language (Kolovos et al., 2016), which, in fact, consists of 9 sublanguages. Among these is Epsilon Object Language (EOL) for performing low-level model operations and Epsilon Transformation Language (ETL) for performing "standard" transformation tasks. But there is also Epsilon Flock language for model migration – transforming a model in response to its metamodel evolution. It contains typical model migration operations described at a high level and consequently in a more concise way than, e.g., via ETL. This language can be applied to some other simple transformations of the whole model as well – see comments on use of Flock for the initial step of PN2SC task in Section 4.2. The lQuery language is also domain specific – adapted to typical model modification tasks during a diagram building. In some cases a simple domain specific transformation can be better described as model mapping, therefore mapping languages are also used. For example, when domain specific workflow languages are to be transformed to their standard execution form by a workflow engine in TDA platform (Kalnins et al., 2014), a special mapping language occurs to be the best solution (Lace et al., 2014). And in the ReDSeeDS project some traditional model to model transformation steps could be better described by a dedicated mapping language (Kalnina et al., 2012) than by the universal MOLA language actually used.

Most of the transformation languages mentioned so far are somehow related to Java programming language and its support platforms. First, most of the languages are implemented on the basis of EMF model repository in Eclipse, which is a completely Java-based environment. Many of the languages permit closer links to Java by supporting custom extensions based on Java types and constructs. Therefore it is a natural question – what is the situation related to the other most used software platform in practice – Microsoft .NET.

2. LINQ, .NET and MDD

According to TIOBE Index for August 2016, C# is the 4th most popular programming language (WEB, d). C# is a general purpose, object oriented, C like language. C# programs run on .NET Framework. Microsoft introduced the first version of C# in 2000. Since then several versions of C# have been released. Currently the actual version is C# 6.0 released in July 2015. From previous versions we want to highlight C# 3.0, where LINQ and lambda expressions were introduced. LINQ – Language Integrated Query Language – is a .NET Framework component integrated in C# (or VB.NET) supporting native query capabilities. LINQ is inspired by the SQL language used to define database queries. However, LINQ queries may be used to query data in different data sources, e.g. LINQ to Entities (database), LINQ to XML, LINQ to Objects. LINQ in C# supports query syntax and method syntax. Query syntax is used to define SQL like queries. Method syntax is used to define queries as a chain of LINQ methods, with internal function parameters defined via lambda expressions. Though initially the query syntax was faster accepted due to its similarity to the classic SQL, now most of C# developers prefer the method syntax over query syntax as this approach is more universal and more naturally integrates with other C# features (see, e.g. Albahari and Albahari, 2015).

Typically, C# developers use some version of Visual Studio for software development.

2.1. MDD Technologies in .Net World

As C# and .NET is one of the widely used programming techniques there have been attempts to bring MDD technologies to .NET technical space. Visual Studio 2015 enterprise edition supports a special kind of projects – modelling project. In the modelling projects it is possible to create various UML diagrams – Class diagrams, Sequence diagrams, Use case diagrams, Activity diagrams and Component diagrams. However, there is no close integration between models and code. Static structure, for instance, class diagrams, of course, corresponds to classes and properties in the code, but apart from descriptive mappings there is no behaviour related integration between models and code.

Visual Studio has a support for DSL tool definition – Visual Studio extension "Modeling SDK" (WEB, e). It is possible to generate C# code (or any other text, including textual code in any programming language) from models created using DSL, for code generation the text template language T4 (WEB, f) is used. This language consists of text blocks – fragments of a text to be generated with placeholder variables inserted, and control blocks – sections of a code in C# which navigate the source model, control the order of generation and generate values of placeholder variables. In general, the Modeling SDK supports the creation of a Domain model (simplified class diagram) of the DSL to be created, defining a diagrammatic visualization of such models (graphical programs in this DSL) and executable code generation from these models using T4 (Cook et al., 2007).

Of course, apart from Microsoft proposed MDD related technologies there are also third party solutions for MDD adaption in .NET world, but these approaches cover only some aspects of MDD.

One of the approaches positioned as MDD technology for .NET is Xomega project (WEB, g). It is possible to use a database or an object model to generate an application implementing CRUD operations for model elements. Various .NET related target environments are supported, e.g. WPF, ASP.NET.

Another MDD framework is open source framework NReco (WEB, h). It is usable only for ASP.NET application development. Actually this is only XML based DSL for ASP.NET application development. There only model to text transformation (actually XML to text transformation) is used for code generation.

Another MDD related approach is Novulo (WEB, i). In this approach multiple models (user interface, process models and Data models) are used. Models are used to generate code.

One of the MDD aspects is model-to-model transformations. There are few model transformation languages using .NET technical space. The most popular and mature one is GrGen.NET (Jakumeit et al., 2010). GrGen.NET is a graph rewrite system therefore it supports transformation of graph models. It consists of Graph model language, Pattern language, Rewrite Language and Rule Application Language. Actually, it is a full transformation language working in .NET technical space.

Another, a later model transformation language using .NET technical space is NMF or .NET Modelling Framework (WEB, j). The best document describing this approach is Master thesis by Georg Hinkel (2013). The supported transformation language, NTL, has a structure similar to QVT-R (OMG, 2011a) and consists of rules, patterns and declarative rule dependency descriptions, however, fragments in plain C# can be easily incorporated. NMF now has its own model repository, which to a certain degree supports model interchange with Eclipse EMF (Steinberg et al, 2008), see more in Section 4.3.

2.2. LINQ

As it was already stated in Section 1, model transformation tasks also are domain specific. We think that for many domains true model transformation languages for application development in C# are actually not necessary. We think that LINQ features are sufficient to define model transformations for these domains.

LINQ mainly is a query language. Originally it was designed for convenient data selection from relational databases. But data selection is an important part of transformation tasks as well. LINQ typically works on object model representing a database or some other data source. Actually, the object model used in LINQ is very similar to MOF metamodels.

Besides data selection, transformation tasks include also data modification (including create and delete). LINQ has some means for data modification (create and delete) as well.

Thus LINQ provides all necessary means for model transformation definition. In addition, all these features can be defined in a true functional style. The question is whether these means are easy to use and adequate for transformation definition.

The only serious gap between LINQ and model transformation languages is technical spaces. LINQ supports various technical spaces including databases, object collections, XML files, etc, but currently does not support model repositories (including the most popular one – Eclipse EMF (Steinberg et al, 2008)) most often used in the model transformation development. In the next section we propose a LINQ adapter for model repositories thus enabling model transformation development in LINQ.

2.3. LINQ as Model Transformation Language

The most popular model repository to store source and target models for model transformations is Eclipse EMF (Steinberg et al, 2008). Since LINQ is usable for various data sources we propose to add one more data source – model repositories (see Fig. 1).

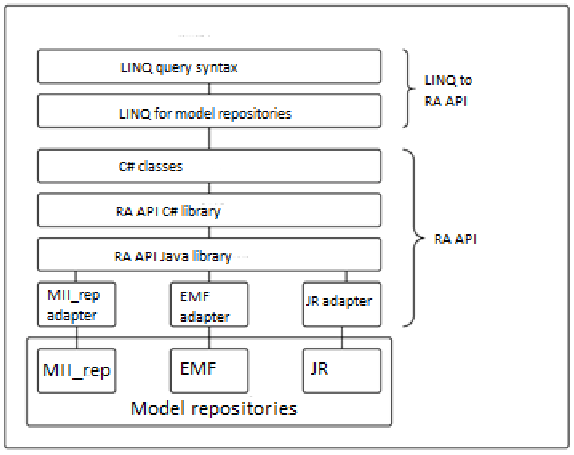

Fig. 1. Model repositories for LINQ

Instead of basing LINQ adapter directly on Eclipse EMF API (Steinberg et al, 2008) we decided to use RA API (Kozlovics and Barzdins, 2012). RA API is an interface supporting various model repositories including Eclipse EMF (Steinberg et al, 2008), besides, it already has a C# wrapper.

The most important part of the developed LINQ adapter for model repositories is the class TDAQueryable implementing IQueryable<T> interface. The class implements methods required to query data in model repositories.
LINQ to entities generate C# classes for database entities. Similarly, in our approach for all metamodel classes in a model repository C# classes are generated. Besides the classes generated from the metamodel, one context class managing repository objects is also generated. The Context class, like in LINQ to Entities (WEB, k), provides instance sets for each metamodel superclass. The context class also ensures change propagation and manages communication between the model repository (managed through RA API) and LINQ methods.

3. Case Studies

In this section we discuss two transformation case studies with LINQ used as a model transformation language. We start with the simpler case – visualization of a process execution progress. We assume that we have the process definition as a diagram and we want to change the line colour of the executed process instances.

A more complicated transformation case study Petri Nets to Statecharts (van Gorp and Rose, 2013) is discussed in Section 3.2. This case study is interesting because of the need for a complicated data selection and management operations. It permits to evaluate the expressiveness of a model transformation language. In addition, this case study was discussed in TTC 2013 (van Gorp et al., 2013), therefore there are publicly available solutions of this case study in other transformation languages.

3.1. Model Transformation Example: Process Visualization

In this section we describe a model transformation use case within the business process management domain. It emerged from the ESF project *Process management program systems' construction technology and its support tools*[1] carried out by the Institute of Mathematics and Computer Science (IMCS), University of Latvia. The main task is to visualize the given process execution.

Usually the processes are described by some sort of business process modelling language, e.g. a graphical one like BPMN, but it could be also another visual language. Every step of the process is represented by some element of the modelling language, e.g. a BPMN activity is depicted as a rounded rectangle, but BPMN gateway is a diamond. An execution of a particular process goes through the elements of the process according to the semantics of the modelling language. Monitoring of the execution can be done through visualisation. The simplest way is to mark those process elements which have been executed. It can be achieved by changing the colour of the process elements, e.g. they can be outlined red. Marking of executed elements, may be used also for other visual representation, e.g. change of font colour if process elements are depicted textually.

Let us define abstract syntax of a very simple process modelling language (see Fig. 2).

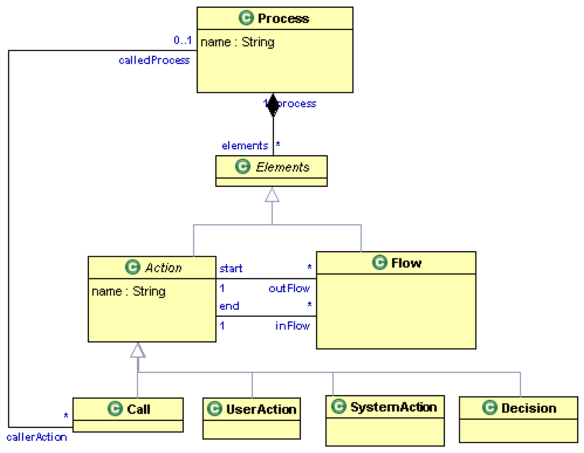

Fig. 2. Process definition language metamodel

Processes are defined by the *Process* class. A process has a name and it consists of elements. The abstract *Elements* class has two subclasses, thus there are two types of process elements, actions and flows, defined by the *Action* and *Flow* classes. Actions describe steps of the process. They may be user actions (*UserAction*) when the performer of the action is a human, system actions (*System Action*) when the performer is a

[1] Agreement no. 2010/0325/2DP/2.1.1.1.0/10/APIA/VIAA/109 (2010-2013).

computer, call actions (*Call*) which trigger an execution of another process (referenced by the *calledProcess* association) and decision actions (*Decision*) which select next actions depending on some constraints. It is possible to obtain following actions using flows. Every flow has a start action (referenced by *start* association) and end action (referenced by *end* association). Every action may serve as a start or end for any number of flows. The execution of the process starts from any action which has no incoming flows. When the execution of an action is finished any actions reachable by outgoing flows may be started. The execution of the process ends when there are no more started actions. The execution traces are stored according to the metamodel depicted in Fig. 3.

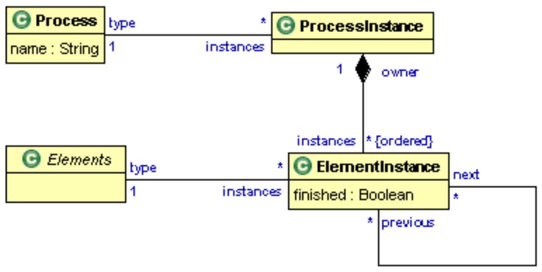

Fig. 3. Process execution metamodel

When a process has started, the *ProcessInstance* object (we will call it simply a process instance) is created. The *type* link is created to the appropriate process definition (*Process* object). When an action has started, the *ElementInstance* object is created for the action (referenced by *type* link) and for the flow (also referenced by the *type* link) which is used to start the action. *ElementInstance* objects are put into the process instance (referenced by the *owner* link). An *ElementInstance* object is connected to the *ElementInstance* object which has triggered the start of the object by the *previous* link. Thus the next executed instances are denoted by *next* links (there may be more than one next instance). The finished instances have the value of the attribute *finished* equal to *TRUE*.

So far we have defined the abstract syntax of process definition language. We assume that there may be more than one visual representation (concrete syntax) for a process. Therefore we define abstract classes that represent a generic visualization (see Fig. 4).

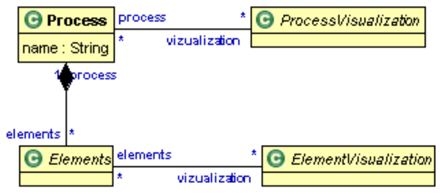

Fig. 4. Generic visualization metamodel

Every process may be visualized by any number of process visualizations. Every process element may be visualized by any number of *ElementVisualization* objects. It should be noted that in a single process visualization there may be shown more than one process and also multiple process elements may be depicted as a single visualization element. Let us define particular means for process visualization – simple graphical diagrams (see Fig. 5).

The *GraphDiagram* class represents graphical diagrams which are used to represent processes. We may think that every process definition is represented as a single *GraphDiagram* object. A graph diagram contains nodes (class *Node*) and edges (class *Edge*). An edge connects exactly two nodes. A node is shown in a diagram as a box. The box has a shape (e.g. rectangle, circle), line colour, background colour and text within the box. An edge has a shape (e.g. arrow), colour and text. Actions are shown as nodes, flows as edges. Thus every instance of the *Action* class has a *visualization* link to a *Node* class instance, but every instance of the Flow class has a visualization link to an Edge class instance. Depending on the type of the action the appropriate box has a particular shape, line and background colours. However this is not important for the transformation task described below, so we omit these details.

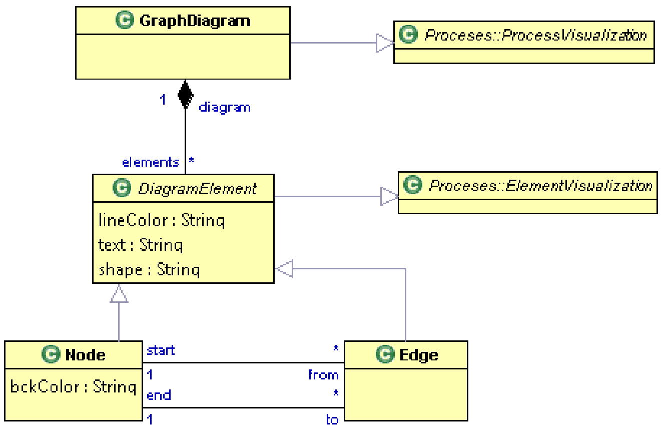

Fig. 5. Graph diagram metamodel

Let us get back to the task. Assume that there are defined several processes and several visualizations for every process. Process instances are executing within the system. We need to mark the visual elements of a particular process instance if they have already finished their execution. Given a single *ProcessInstance* object (process instance), we Find all *ProcessVisualization* elements that visualize process elements (*ElementInstance* objects) within the given process instance (having an *owner* link to the given instance), which have been already executed (*finished* equals *TRUE*). We mark *ProcessVisualization* elements of executed process instances according to their types.

- *Node* – make line color red (set the attribute *lineColor* value to “red”).
- *Edge* – make line color red (set the attribute *lineColor* value to “red”).

In the Listing 1 C# method implementing the transformation is given. The method has one parameter - Id of the *ProcessInstance* whose execution progress should be displayed. All LINQ queries are executed in some LINQ context. In this case study the context class is named *ProcVisContext*. (User provides the desired name of context class when generating C# classes for model repository.) The first task is to find the required process instance using its *Id* (Line 5-6). Afterwards the executed (those where *pe.finished* is true) process element instances are found.

For the selected transformation task various modifications are possible. To better demonstrate LINQ query possibilities we have selected a modification where the elements of executed sub processes are coloured as well. Subprocesses are called using call elements, therefore process element instances whose type is *Call* are selected. For each call element instance a set of next elements is found. Among next elements there may be elements executed in other process instances which follow elements in the current process. Afterwards we find Id of the process instance containing the selected element. We are interested to find other process instances, therefore only Ids of other process instances are left. For each called process instance this method is recursively called.

Afterwards the executed elements in the processed process instance are coloured. At first a type of the executed process element is found. Afterwards visualisation of the element type is found. For the visualization its type is clarified. If the type is *Node* or *Edge*, the line colour of the element is modified. Although it is not considered in the discussed case study, other means of visualization could be used. For the type clarification the type cast operation is used. If the type cast is successful, an element instance of the type is created. Otherwise, value of a variable is null. Null conditional operator is used to evaluate whether the type cast was successful.

Transformation completes with *SaveChanges* operation persisting changes to the model repository.

```
static void ProcessVisualization(Int32 processInstanceId)
{
using (ProcVisContext dc = new ProcVisContext())
{
  ProcessInstance pi = dc.ProcessInstances.First(pii => pii.Id ==
processInstanceId);
  List<ElementInstance> eleminst = pi.instances.Where(pe =>
pe.finished).ToList();
  var idList = eleminst.Where(piac => piac.type is Call).SelectMany(iac
=> iac.next).Select(oo => oo.owner.Id).Where(ids => ids !=
processInstanceId).Distinct().ToList();
  foreach (var pids in idList)
  { ProcessVisualization(processInstanceId);}
  var visualizations = eleminst.Select(eii => eii.type).SelectMany(v =>
v.visualization).ToList();
  foreach (var v in visualizations)
  {
    DiagramElement n = v as Node;
    n = n ?? v as Edge;
    if (n != null)
    {
      n.lineColor = "red";
    }
  }
  dc.SaveChanges();
}
}
```

Listing 1. Process visualization transformation in LINQ

While working on the case study it appeared that expressiveness of the LINQ allows to define the same transformation in various ways. For example, it is possible to write a long LINQ constraint and select chains, or to split them in smaller chains. Instead of *Select* and *SelectMany* it is possible to use a *foreach* loop processing each collection element separately. The authors think that the approach with *foreach* loops is more understandable for inexperienced LINQ users. However, the approach with condition and select chains allows to express the transformation algorithm in a shorter and more declarative way. We believe that the selection of style is a matter of taste and both approaches are usable.

We think that the possibility to define the discussed transformation in various programming styles demonstrates expressiveness of the LINQ language. This case study confirmed that LINQ could have been used as a transformation language for similar transformation tasks.

3.2. Petri Nets to Statecharts Transformation

In this section we describe the LINQ implementation of the Transformation Tool Contest (TTC) 2013 case – a transformation of Petri nets to Statecharts (van Gorp and Rose, 2013). Both Petri nets and statecharts are well known extensions of finite automata. They are modelling notations used for describing various kinds of processes.

Petri nets in fact are a simplified subset of UML activity diagrams (OMG, 2011b). In activity terminology they contain only simple action nodes, start, fork and join nodes as control nodes and control flows. No behaviour branching mechanism is included. In Petri Net terminology the actions are named places, control nodes are named transitions and control flows are simply edges (see an example in Fig. 6, the upper part). The execution semantics of both notations is defined by a token mechanism, certainly, Petri nets is a simple subcase in this aspect. The basic concept there is the possibility for a place to contain one or more (control) tokens, a place with a token is said to be active. Initially the start place contains just one token. A transition is enabled when all its source (input) places contain at least one token. An enabled transition can fire, then a token is removed from each source place and a token is added to each target (output) place. A typical situation in modelling a system is that no place contains more than one token, thus an "execution" of a Petri net means a permitted sequence of sets of active places.

The statecharts discussed in this case are a simple subset of UML state diagrams. They contain simple (basic) states and compound states (AND and OR states) which contain other states (child states), there is also an initial state. An AND state node contains several regions, which are child OR states. All these regions are active in parallel. On the contrary, an OR state contains a set of nodes linked by transitions, only one of these nodes may be active. The state transitions are represented by hyperedges (possibly having several source or target state nodes), these hyperedges are graphically shown as simple edges (when there is one source and one target node) or as fork/join nodes plus edges from source nodes and to target nodes (see an example in Fig. 6, lower part). Transitions have no events or guard conditions. We will use only transitions where all source and target nodes are simple states. The behaviour of a statechart is defined as a consistent sequence of sets of active nodes – configurations. Informally, a configuration is consistent, if together with a state all its nesting states are active as well. In addition, only one OR state child may be active, but all children of an AND state must be active. Thus the behaviour semantics is a permitted transition from a configuration (active state set) to configuration. A transition can occur according to a hyperedge if all its source states are active (in our case it is sufficient that basic source states are active). In the result all basic source states become inactive and all basic target states become active (and relevant compound states according to configuration definition), we assume that basic source and target states do not overlap.

Thus the behaviour both for Petri nets and statecharts is a set of permitted sequences of sets of active elements (sets of active places in one case and sets of active states in the other). The execution is completely non-deterministic in both cases – the simple syntaxes do not include any data-based conditions to make it deterministic. The basic principles of determining a possible execution path are also similar – source elements of a transition must be active. The main difference is that a set of places in a Petri net is flat, but states in a statechart have a hierarchic structure. The possible concurrent activity of places thus is implicit (defined by transition configuration), concurrent activity of states is determined also by the state hierarchy. This is the main obstacle in building a simple mapping from Petri nets to statecharts.

The given task is to define a structure-preserving transformation from a Petri net to an equivalent statechart. Structure preserving here means that we map each place to a basic state node and each transition to a hyperdege. The relevant hierarchy of AND/OR states has to be added to make it all consistent. The equivalence means that the possible execution sequences are equal with respect to this mapping. Eshuis (2005) proved that such a transformation algorithm exists, which creates a correct result in most cases, but can also fail in some cases when a solution does exist. It should be noted that such a solution does not always exist, e.g. the Petri net must be safe – no place is allowed to hold more than one token during any execution sequence. The algorithm is quite complicated – an initial "draft" statechart is built, which then is gradually optimized by improving its state hierarchy structure, the reached progress is marked by removing relevant elements from the net. The success is signalled by the fact that no excess elements of certain kind have been retained in the source net.

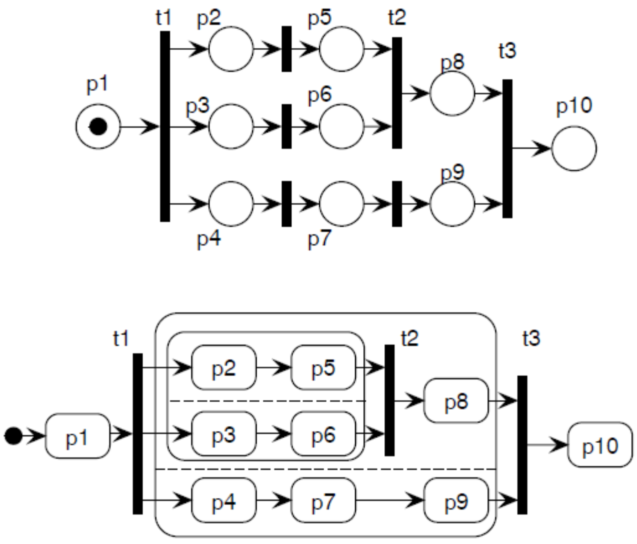

Fig. 6. Petri net and equivalent statechart, hyperedges are shown as forks/joins – taken from (van Gorp and Rose, 2013)

This algorithm has been chosen for TTC 2013 – and also for implementation in LINQ, because it puts interesting requirements on the chosen transformation language, both for implementing a recursive optimization of the target model and having a necessity to select elements from the source model using a quite complicated pattern. In addition it should be noted that the task has a practical value as well. As already pointed out, in fact it is a transformation from a subset of UML activity notation to a subset of state diagram notation. Such task is required in some software development methodologies, where requirements to a system are specified by activity diagrams (corresponding to use cases), but implementation is related to state diagrams. In particular, many implementation DSLs, especially for embedded systems, are based on statecharts. It is true that the given algorithm solves only “the bare skeleton” transformation, but it could be extended by including appropriate data-related elements both for Petri nets (in fact, activity diagrams) and statecharts.

Now the formal description of the algorithm and the solution in LINQ is given. First, an example of such transformation in a diagram form is given in Fig. 6.

The mapping of net elements to basic states and hyperedges (visualized as forks or joins) is shown by reusing place and transition element names for statechart element names. It can be seen that two nested *AND* states four *OR* states must be used in the solution.

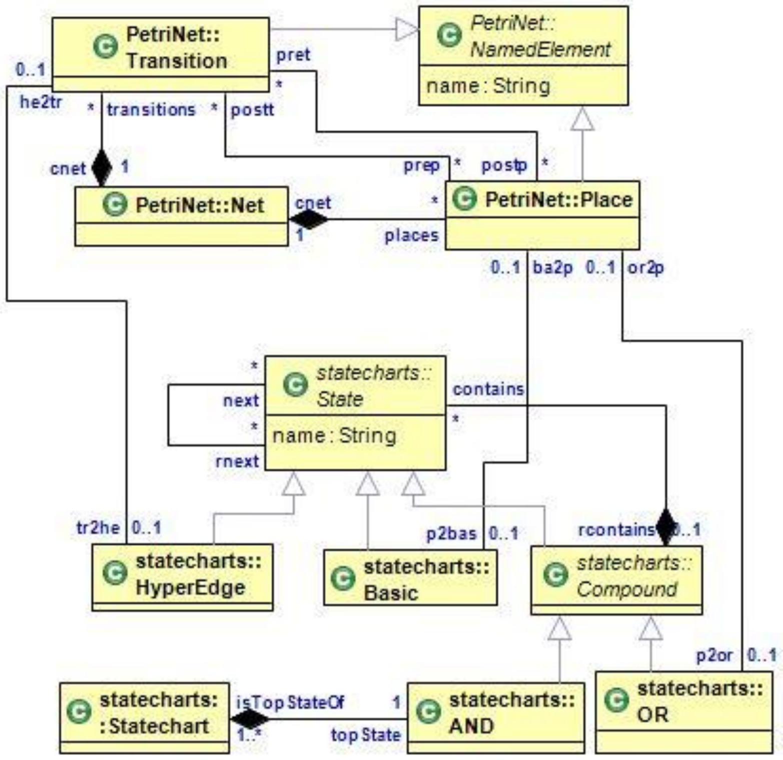

Fig. 7. Source (top) and target (bottom) metamodels of the transformation

The source and target metamodels are shown in Fig. 7. The metamodels in our approach are merged into one common metamodel. In addition, three traceability associations linking elements of both models are added:

- (*Place*): (0..1)ba2p <-> (o..1)p2bas :(*Basic*) - *Place* to *Basic*
- (*Place*): (0..1)or2p <-> (o..1)p2or :(*OR*) - *Place* to *OR*
- (*Transition*): (0..1)he2tr <-> (o..1)tr2he :(*HyperEdge*) - *Transition* to *Hyperedge*

Note that in the proposed target metamodel *HyperEdge* is made a subclass of *State* (to optimize the used associations).

The algorithm contains the following steps (according to (van Gorp and Rose, 2013))

- Create the initial version of the statechart, to be optimized after:
  - For every *Place p* in source model create an instance *b* of *Basic* with *b.name = p:name* and an instance *o* of *OR* with *o.contains = {b}*
  - For every *Transition* t create an instance of *HyperEdge* e (with *e.name = t.name*)
  - All *pret/postp* and *postt/prep* arcs should be mapped to *next/rnext* links between the *States* equivalent with the input *NamedElements* that are connected by these arcs
  - Add links between the source and target model instances according to the introduced traceability associations
- Apply the AND/OR state reduction rules as long as possible:
  - Try to apply the pre-AND rule – to construct an *AND* state for a set of *Places* that are connected to the same incoming and outgoing *Transitions*. The pre-AND rule can be applied to a *Transition t, iff* |*t.prep*| > *1* and every *Place* in *t.prep* is connected to the same set of outgoing transitions and the same set of incoming transitions. In our implementation the rule is split into two rules – tryPreAnd which tries to find a *Transition t* satisfying the precondition and in case of success invokes the second rule applyPreAnd, which performs the AND reduction for this *t*. More precisely, a new *AND* state *a* is created which contains the *OR* states *t.prep.p2or*; and a new *OR* state *p* containing a is created. In addition, in the source model all but one of the *Places* in the set *t.prep* are removed, the remaining *Place* is linked to *p* .
  - Try to apply the similar post-AND rule – to construct a similar *AND* state, but with a different precondition - |*t.postp*| > *1* and every *Place in t.postp* is connected to the same set of outgoing transitions and the same set of incoming transitions. Similarly, the implementation uses two subrules – tryPostAnd and applyPostAnd (similar to the pre-case, with only the *prep* link replaced by *postp*).
  - Try to apply the OR rule – to construct an *OR* state for a *Transition t* that has a single preceding *Place* and single succeeding *Place*. The rule precondition requires also that there is no *Transition t1*, such that *(q ∈ t1.prep) ^ (r ∈ t1.prep) or (q ∈ t1.postp) ^ (r ∈ t1.post p)* where *q* is the single place contained in *t.prep* and *r* is the single place contained in *t.postp*. However, it is also permitted to be *q = r*. If such *t* is found, then applying the OR rule to it means the creation of a new *OR* state *p* such that *p.contains* is the set of *OR* states *q.p2or.contains ∪ r.p2or.contains*. In addition, in the source net model the *Transition t* and *Places q* and *r* are removed and a new *Place p* is added such that *p.pret = (q.pret ∪ r.pret)* and *p.postt = (q.postt ∪ r.postt)*. The *Place q* can also be used in the role of the new *p*.

The reduction rules are applied (in any order) as long as possible. For the subclass of Petri nets where this algorithm really produces a result, there should be exactly one *Place* left in the source net. Then the resulting statechart is the transformation result, otherwise, only some partial result is obtained.

```
public class PN2STtransf
{
  static Model1Container context = new Model1Container();
  bool rulePerformed;

  public void transform()
  {
    initialize();
    do
   {
      rulePerformed = false;
      rulePerformed = tryPreAnd();
      if (rulePerformed) continue;
      rulePerformed = tryPostAnd();
      if (rulePerformed) continue;
      rulePerformed = tryOr();
    }
    while (rulePerformed);
  }

  public void initialize()
  {
    var topStat = new AND {name = "TOP"};
    context.Statecharts.Add(new Statechart() {topState = topStat});
    foreach (var plac in context.NamedElements.OfType<Places>())
    {
      var basicStat = new Basic {name = plac.name};
      context.States.Add(basicStat);
      plac.p2bas = basicStat;
      var orStat = new OR() {name = "o_" + plac.name};
      orStat.contains.Add(basicStat);
      context.States.Add(orStat);
      topStat.contains.Add(orStat);
      plac.p2or = orStat;
    }
    foreach (var trans in context.NamedElements.OfType<Transition>())
    {
      var hyperE = new HyperEdge() { name = trans.name };
      foreach (var pl in trans.prep) {hyperE.rnext.Add(pl.p2bas);}
      foreach (var pl in trans.postp) {hyperE.next.Add(pl.p2bas);}
      context.States.Add(hyperE);
      trans.tr2he = hyperE;
    }
    context.States.Add(topStat);
    context.SaveChanges();
  }
  public bool tryPreAnd()
  {
    var transList = context.NamedElements.OfType<Transition>().Where(t =>
t.prep.Count() > 1).ToList().Where(t=>t.prep.All(p => p.pret.OrderBy(t1
=> t1.name).SequenceEqual(t.prep.FirstOrDefault().pret.OrderBy(t2 =>
t2.name))) && t.prep.All(p => p.postt.OrderBy(t3 => t3.name)
.SequenceEqual(t.prep.FirstOrDefault().postt.OrderBy(t4 => t4.name))))
.ToList();
```

```
    //pattern-based selection - only these transitions t where pre-places
p of t are more than 1 and for all such p the set (list) of pre-
transitions is the same as this set for the first //such pre-place, and
also for all such p the same is true for the set of post-transitions
    if (transList.Count() >= 1)
    {
      var trans = transList.FirstOrDefault();
      applyPreAnd(trans);
      return true;
    }
    else return false;
  }

  public bool tryPostAnd() //symmetric to tryPreAnd with post instead pre
  {
    var transList = context.NamedElements.OfType<Transition>().Where(t =>
t.postp.Count() > 1).ToList().Where(t => t.postp.All(p => p.pret
.OrderBy(t1 => t1.name).SequenceEqual(t.postp.FirstOrDefault().pret
.OrderBy(t2 => t2.name))) && t.postp.All(p => p.postt.OrderBy(t3 =>
t3.name).SequenceEqual(t.postp.FirstOrDefault().postt.OrderBy(t4 =>
t4.name)))).ToList();
    if (transList.Count() >= 1)
    {
      var trans = transList.FirstOrDefault();
      applyPostAnd(trans);
      return true;
    }
    else return false;
  }

  public void applyPreAnd(Transition trans)
  {
    var newAnd = new AND(){ name = "and" };
    var newOr = new OR(){ name = "or" };
    newOr.contains.Add(newAnd);
    foreach (var orStat in trans.prep.Select(pl => pl.p2or))
      { newAnd.contains.Add(orStat); }
    int n = 0;
    foreach (var plac in trans.prep.ToList())
    {
      n++;
      if (n == 1) {plac.p2or = newOr;}
      else
      {
        context.NamedElements.Remove(plac);
      }
    }
    context.States.Add(newAnd);
    context.States.Add(newOr);
    context.SaveChanges();
  }

  public void applyPostAnd(Transition trans)
  { //not shown - symmetric to applyPreAnd
  }

  bool sharesTrans(Places prepl, Places postpl)
  {
    if (prepl.pret.Any(t1 => t1.postp.Contains(postpl))) return true;
    if (prepl.postt.Any(t2 => t2.prep.Contains(postpl))) return true;
    return false;
  }
```

```
// the semi-procedural version for tryOr and applyOr combined
  public bool tryOr()
  {
    Transition transToUse = null;
    Places prepl = null, postpl = null;
    foreach (var trans in
context.NamedElements.OfType<Transition>().ToList())
    {
      if (trans.prep.Count() != 1 || trans.postp.Count() != 1) continue;
      prepl = trans.prep.Single();
      postpl = trans.postp.Single();
      if (prepl == postpl)
      {
        transToUse = trans;
        break;
      }
      if (sharesTrans(prepl, postpl)) continue;
      else
      {
        transToUse = trans;
        break;
      }
    }
// apply OR rule - reset the relevant links to pre-place , remove the
used transition and post-place
    if (transToUse == null) return false;
    OR preor = prepl.p2or;
    OR postor = postpl.p2or;
    preor.contains.Add(transToUse.tr2he);
    if (prepl != postpl)
    {
      foreach (State st in postor.contains) {preor.contains.Add(st);}
      foreach (Transition tr in postpl.pret) {prepl.pret.Add(tr);}
      foreach (Transition tr in postpl.postt) {prepl.postt.Add(tr);}
      context.States.Remove(postor);
      context.NamedElements.Remove(postpl);
    }
    context.NamedElements.Remove(transToUse);
    context.SaveChanges();
    return true;
  }
}
```

Listing 2. Petri nets to statecharts transformation in LINQ

The provided C# + LINQ code for the Petri nets to Statecharts case is sufficiently readable. The code length is also similar to other available solutions in textual transformation languages from TTC 2013 (van Gorp et al., 2013), including the solution in Epsilon Object Language (EOL) (van Gorp and Rose., 2013) provided by the case submitters. Especially, the possibility to describe in a declarative way complicated search patterns in the source model should be emphasized. An example is the pattern for finding the list of Transitions to which the pre-AND reduction can be applied (in the method tryPreAnd). There all the applicability preconditions are specified as one LINQ expression. Certainly, the expression would become more readable if set equality function would be available in LINQ – currently only the sequence equality is available. However, set equality function is absent in some other transformation languages as well, including EOL (Kolovos et al., 2006). In a contrast, the precondition for OR reduction could be specified in LINQ in a declarative way as well, however here this style occurs to be less readable than the semi–procedural style used in the code. The features of C# permit also to describe the general recursive structure of the statechart reduction algorithm in a natural way. See more on LINQ comparison with other languages on the basis of this case in Section 4.

4. LINQ versus Model Transformation Languages

In this section we will discuss expressiveness of LINQ compared to other transformation languages. Though transformation languages is not a very popular domain there are dozens of transformation languages. Even in different editions of Transformation Tool Contest (TTC) more than 20 languages have appeared, but there are many that have not participated in TTC. Therefore we will discuss only some languages most relevant for this paper.

Languages GrGen.NET and NMF are sharing .NET and C# technical space. Language built as internal DSLs in other language is lQuerry. Widely used languages are Epsilon and ATL. In addition, MOLA language is discussed. It is a language of completely different type – a graphical language.

4.1. lQuery

The lQuery language (library) (Liepiņš, 2012, 2015) is a set of functions for querying and modifying models stored in a model repository. To achieve the expressivity of the existing model transformation languages in a general-purpose scripting language authors of lQuery use the ideas from the functional programming, specifically from combinator parsing (Hutton, 1992). Functions are built in progressive layers, where every next layer is based on the previous one. The first layer is built directly on the repository API. Selector functions and function combinators are defined so that reference objects and object collections can be referenced easily by passing them as arguments to those functions. For common cases, where string expressions would suffice, an XPath (WEB, l)-like selector is defined. It is a shorthand notation that can be easily mixed with selector functions and combinators.

lQuery is implemented in the Lua (Ierusalimschy et al., 2007) scripting language and is used for the development of domain specific graphical modelling tools in GRAF platform (Sproģis et al., 2010). lQuery is a domain specific transformation language designed for the task of development of in-place transformations (in contrast to many transformation languages which are mainly tailored for batch model conversions from one metamodel to another). It provides options for integration with other parts of the system (databases, compilers, simulators, etc) in which the DSL tool is only a component. The majority of lQuery transformations are context based. Namely, the transformation starts with a single instance element, afterwards it must find the context of the instance, and, finally, it must make some adjustments in the instance graph (create or delete an instance; add or remove a property link; modify an instance attribute). Thus, the necessary steps are navigation, filtration and modification of the instance graph.

For typical tasks appearing in building of diagram editor, the model instance navigation could be specified in LINQ nearly as easy as it is in lQuery. However, lQuery permits to build more easily a combination of several model modification tasks.

Obviously, LINQ could be used also for similar purposes in tool building as lQuery, if tool definition platforms had to be implemented in .NET.

4.2. Epsilon Languages: EOL, ETL

As already mentioned in Section 1.3, the Epsilon transformation language family consists of 9 sublanguages some of which are very domain specific. The most interesting for our comparison is the low-level Epsilon Object Language (EOL) (Kolovos et al., 2006) whose functionality is closest to that of LINQ. In addition, as stated in 3.2, the Petri nets to Statecharts case was solved in EOL by the case submitters (van Gorp and Rose, 2013). EOL is a procedural language containing assignment, if, for and while statements (and some auxiliary ones). However, its main value is the rich set of types and operations. Types include primitive types and collections similar to those in OCL (bag, sequence, set, ordered set, map). In addition, classes of source and target metamodels are also used as types. If EOL is implemented over Java (the standard situation) Java classes can also be interpreted as types. A rich set of operations is already defined for collections – comparable to such a set in OCL, but still less, e.g., there is no set equality operation, but there is a subset checking (includesAll), union (addAll), sub-collection select and quantifiers (exists and forAll). For each of the types new operations can be defined. Especially this feature is used to define operations for model element types (metamodel classes) – typically the main transformation functionality is based on such operations. Expressions are built from operations using the standard dot notation, the same notation is used to navigate a model (as in OCL, but dot notation can be used instead of '->'). There is no explicit lambda notation for function definition, but OCL-like constructs such as places.forAll(p | p.hasSameTransitionsAs(places.first)) in fact play the same role. Thus the functional style in fact is sufficiently supported in EOL. When compared to the LINQ approach presented here, the expressiveness of both languages is quite close, the main difference is in the sets of built-in operations. This fact shows up also when solutions of the Petri nets to Statecharts case are compared – the code length is similar, though not always the most concise expressions are used in the EOL solution. The other Epsilon sublanguages, e.g. Epsilon Transformation language (based on a QVT-R like rule design pattern), are not very relevant for such cases. The exception is the very domain specific Epsilon Flock language (Rose et al., 2010) for model migration to a new metamodel. The specific constructs such as retype, migrate, original, migrated etc. enable to describe simple model mapping tasks to a new metamodel in a very concise way. In the case solution by submitters Epsilon Flock was used for the statechart initialization step – in this step a simple mapping from Petri nets model to statechachart model in fact is done, with *Net* retyped to *Statechart*, *Place* retyped to *Basic* and *Transition* to *HyperEdge*. The migrate construct then permits to define the relevant attribute and link transformations in a very declarative way. Thus the most concise version of this step has been obtained – only 17 lines of code.

4.3. NMF

The .NET Modeling Framework (NMF) is the latest of approaches for support of MDE on the .NET platform. It contains its own model repository based on the metametamodel NMeta. This metametamodel is quite similar to the EMOF standard. The model serialization in XMI format is compatible to the serialization of Ecore models in Eclipse (if some advanced Ecore features are not used), therefore direct model interchange between these two environments is possible in most cases. NMF now is being developed as an open source project (WEB, j.), started by G. Hinkel on the basis of his Master thesis (2013). NMF is based strictly on .NET standards, but especially adapted to C# language.

The most interesting component is the NMF Transformations Language (NTL). The language constructs are deeply rooted in various C# features. However, the general structure of NTL tries to be similar to the standard OMG language QVT-R, it consists of rules and patterns. The basic control structure is rule dependency, according to which one rule invokes other dependent rules. However, the desire to comply completely to C# features and usage patterns makes transformations not so easy readable. The TTC 2013 PN2SC case has been solved also in NTL (Hinkel et al., 2013). The transformation initialization part (using strict NTL rule style) requires 58 lines in NTL, as opposed to 25 lines in LINQ here. The code readability also seems to be worse. In the provided NTL solution the statechart reduction rules are coded in plain C#, without explicit use of patterns, but relying strongly on advanced C# facilities for collections. Only the reduction frame is specified as a true NTL transformation rule. Since all C# features relevant for model transformations can be freely used in the LINQ approach proposed in this paper, no clear advantages of NTL can be seen.

4.4. GrGen

This transformation language is interesting for comparison since it also operates in the technical space of C# and .NET. The support for the most popular model repository EMF (Steinberg et al., 2008) is only partial. GrGen (Jakumeit et al., 2010) uses its own repository, therefore models have to be migrated from EMF to GrGen repository. For data import a partially standardised solution is available, but data export has to be created for each case from scratch.

Unfortunately, the authors of this paper have no access to a solution in GrGen for any of the analysed cases. GrGen is a very typical transformation language from a structure point of view, though its terminology is based more on graphs than models. A transformation is defined by specifying a pattern to be matched in the source model (graph). The pattern part is followed by rewrite part, in the form of a replace or modify block. The replace block is used to replace the matched subgraph with the pattern defined in the block. The modify block specifies which modifications have to be performed in the matched subgraph. One more important block is *eval* which is used to modify the attribute values.

GrGen is a completely different kind of language than LINQ, since the matching (query) part is strictly separated from the modification part. In LINQ there is no such strict separation. The rule execution control structure in GrGen again is a completely separate part. The expressiveness of the languages is similar, though typical the code size in GrGen could be larger due to the separate rule control part. However, for specific tasks having a built-in support constructs GrGen could be superior, e.g., when a retype operation having a specific notation in GrGen is to be applied, or when all matched elements in a pattern must be deleted.

4.5. ATL

ATL (Jouault and Kurtev, 2005) is the most popular model transformation language so far, also the one most used in real industry projects. It is a textual language based on the standard rule – pattern schema. Each rule contains a source pattern (using the keyword from) and a target pattern (keyword to), both patterns are based on a metamodel class. The source pattern contains matching constraints possibly involving other classes as well, but the target pattern contains expressions for setting the attribute and link values for the instance to be created. Both the constraints and value expressions are based on OCL (in the target pattern these in fact are OCL based assignments). The preferred style in ATL is to apply the rules in a declarative way to the source model, without any control structure, while a source pattern match can be found. However, the imperative style is also supported – each rule can have a do-section, where other rules can be explicitly invoked, for and if statements can be used in this section. The main component contributing to the expressivity in ATL is the large set of collection types and operations on them taken from OCL (the original OCL syntax – '->' for applying an operation to a collection is retained). However, direct collection comparison is not included. Iterators for collections are supported as well, thus in fact lambda expressions in OCL syntax are supported, and so is the functional style in general. Helper functions over the source model can also be defined, e.g. to support a reuse of subexpressions. A special syntax is provided to reference the targets of source model elements in the target model, but explicit traceability links between the source and target metamodels cannot be used. Thus ATL seems to be more expressive than LINQ in general.

However, there is an issue with ATL when transformation tasks similar to our cases are considered. The standard syntax and semantics of ATL is oriented to a situation when source and target models are completely different, but both considered cases are typical in-place transformations – the source model is being gradually modified. Only relatively recently (Tisi et al., 2011) the in-place transformation option, with source and target metamodel also being the same, has been added to ATL. There is no ATL solution for the Petri net to statechart case, therefore it is not so easy to evaluate how readable would be the statechart reduction part of the case in ATL, but at least it could be specified in the style close to the LINQ solution.

4.6. MOLA

We conclude this language comparison with a transformation language of a different kind – the graphical language MOLA (Kalnins et al., 2005). Both cases considered in Section 3 have been implemented in MOLA as well (but unfortunately, not published). Transformations in MOLA consist of procedures, which in turn consist mainly of rules and loops, but can contain also text statements (assignments/decisions) and procedure calls. Each procedure is a diagram similar to UML activity diagram. Rules and loops are "large" nodes of the diagram containing fragments of the metamodel – classes and associations renamed to class elements and links. These fragments define the source patterns, where class elements can contain also textual constraints on attributes (in a subset on OCL). Patterns are matched to the source model – a model fragment containing class instances for all class elements of the pattern, linked accordingly and satisfying all constraints must be found. Pattern in a rule is matched once – if a match is found, the normal exit is used, otherwise the else exit is used. A loop node is a container containing one or more rules linked by flows, the first rule is a special one – the loop head. The loop head contains one specially marked class element – the loop variable. A loop execution means that all possible instances of the loop variable are checked whether there exists a relevant match for other pattern elements as well, and, if so, the loop body (other rules of the loop) is executed for this instance. A pattern can contain also a reference to a class element of other rule already matched, then namely the matched instance is used, thus a dependency between the rules can be specified. Rules can contain also specially marked creation elements – then an instance is created and attributes are set as specified in the assignments. Assignments can be used also in other class elements – to modify an attribute of the matched instance. Similarly links can be created. Another marking is used to specify elements and links to be deleted. The main strength of such graphical pattern notation is in the fact that it very clearly shows what a set of instances and links must be found in the model being transformed. At the same time the expression language used in constraints is limited to a degree – all typical operations are supported for primitive types, but sets can be defined only via model navigation (in OCL style) and checked for being non-empty. In the result, for example, the constraint for application of the pre-And rule to a Transition in Petri net case requires nested loops at depth 3 (it was a single expression in LINQ). At the same time finding the relevant node instances to be modified in the visualization case can be specified using a single pattern in a loop head. Thus, there is no clear evidence which of the styles – graphical or textual – is better in transformation definition.

4.7. Summary

From the direct comparisons in this section and case analysis in the previous section it can be concluded that LINQ can be successfully used as a transformation language. As already mentioned in Section 1.3, the recent research shows that model transformations are domain specific as well. For different kinds of transformation tasks different transformation languages must be used. Taking this into account we can say that LINQ is not and never will be the best transformation language for all tasks. However the case studies in Section 3 show convincingly that LINQ can be successfully used in typical transformation tasks. The code length in LINQ solutions is comparable to code length in other languages, and the code readability is sufficiently high. In addition, the expressiveness is high enough to enable the building of various solutions for the same task in completely different styles. It is possible either to apply a completely functional style by using chained sequences of operations (which typically yields a shorter code), or use explicit nested loops, which result in a longer code, but may help in understanding which data element currently is being processed.

5. Functional Language Constructs in Java 8 for Model Transformations

As already mentioned before, in the TIOBE programming language popularity index (WEB, d) C# has the fourth place, but Java has the first place. Java 8 now fully supports functional style and lambda expressions. Therefore it is quite natural and interesting to compare Java 8 capabilities with LINQ. Especially it is so because, as mentioned in Section 1, Java based Eclipse EMF is the most popular platform for implementing model transformation languages. Though Eclipse now fully supports Java 8 as a development environment, it seems that no serious attempts have been made to integrate Java 8 functional style with EMF as a base for transformation languages.

On the other hand, since the functional style support in Java 8 is equivalent to that in C#, there would be no serious problems to provide an analogue of LINQ in Java 8.

Java's Streams API in Java 8 is the closest concept to .NET's LINQ, in the sense that it allows you to query/filter/manipulate collections in a functional style. In a sense, Java 8 Streams API can be considered to be "functional transformations of in-memory collections". Streams API is a view on Java collections which offers these new filtering and other. operations to be applied to a collection. In order to use these operations, a collection typically is interpreted as a stream using the stream() function, thus stream can be treated as a new collection type in Java 8. However, this type is virtual in the sense that it does not really store its elements, they have to be stored in a collection. The Streams interface offers several operations: filter, map, flatMap, sorted, limit, skip, distinct, concat etc., which create a new stream (so called intermediate operations). These operations can be applied one after another to the source stream ("pipelined") until the desired result is obtained. The filter operation contains as argument a lambda expression defining the predicate according to which the elements are retained in the result – a direct equivalent of where operator in LINQ. Similarly, the map operation contains a lambda expression defining how the current element of the stream must be transformed (an equivalent of select in LINQ). The flatMap (also containing a transformation definition) is an equivalent of selectMany. Finally, the "terminal operation" collect should be applied, which stores the result as the chosen collection kind, e.g. List. Alternatively, the quantifier operations anyMatch and allMatch, both containing a predicate defined via lambda expression (equivalents of All and Any in LINQ), can be applied. Thus the main constructs of LINQ relevant for model transformations are present also in the Streams API.

A comparison of LINQ to Streams API on some most typical elementary use cases is given in (Web, m), here are some examples from this comparison:

```
//------------------------------------------------------------
//Any - exists quantifier over a stream:
//LINQ
string[] persons =
            {"Sam", "Danny", "Jeff", "Erik", "Anders", "Derik"};
bool x = persons.Any(c => c.Length == 5);

//Java Streams
String[] persons =
            {"Sam", "Danny", "Jeff", "Erik", "Anders", "Derik"};
boolean x = Arrays.stream(persons).anyMatch(c -> c.length()==5);
//------------------------------------------------------------
//------------------------------------------------------------
//Filtering streams:
//LINQ
string[] names = { "Sam", "Pamela", "Dave", "Pascal", "Erik" };
List<string> filteredNames = names.Where(c => c.Contains("am")).ToList();

//Java Streams
String[] names = {"Sam","Pamela", "Dave", "Pascal", "Erik"};
List<String> filteredNames = Arrays.stream(names).
      filter(c -> c.contains("am")).collect(Collectors.toList());
//------------------------------------------------------------
```

Listing 3. Java 8 Stream API versus LINQ

These examples (and other ones from (WEB, n)) show that the solution quality in LINQ and Java 8 is comparable, though in some cases the LINQ solutions are more concise. Thus the Java 8 new features could be used for defining model transformations in a way similar to how we propose to use LINQ in C#. The only serious obstacle here is the fact that in Java world models typically are stored in Ecore format, which cannot so easily be accessed via collection interface without appropriate support from Ecore implementation.

However, in practice LINQ is mainly used to access relational databases in a functional style (e.g. via LINQ to Entities). There have been also several attempts to support this task using functional style in Java 8, e.g. Lambdaj (WEB, o) jLinqer (WEB, p). But the latest and most complete solution for this task is the Jinq framework (WEB, q). Jinq is based completely on standard Java 8 elements and JPA, but its set of provided query operations is close in syntax to the original LINQ. Certainly, it would be natural to include LINQ-style database access in the next versions of standard Java Enterprise Edition (Java EE) (WEB, r), but most probably it will not be present in Java EE 8 – evidently due to some implementation difficulties. However, research in this direction will have little impact on the use of Java 8 for model transformations.

Conclusions and Future Work

Though model driven software development has recently survived a serious crisis, some more specific cases of it have regained their value and are really used in practice. Therefore a need for MDD support technologies, including model transformation languages, is still present. Many different transformation languages have been developed. A recent trend is to find domain specific transformation task subclasses and to create domain specific transformation languages for these subclasses. This approach is most profoundly supported by the Epsilon project, where the Epsilon language family consists of 9 sublanguages. Certainly, in an appropriate domain specific language for a subclass the transformation can be defined in a more concise and readable way. However, it is not reasonable to create a new language for each task, therefore general purpose transformation languages for a wide variety of tasks still have a value.

On the other hand, software practitioners frequently are reluctant to learn new languages. Therefore the main topic of this paper is how familiar programming languages could be used also for transformation development. Mainly the C# language has been analysed, which is in the 4-th place according to the Tiobe popularity index. C# now includes the LINQ extension for data selection and modification in a functional style. The paper proposes a solution how LINQ could be used as a transformation language. A LINQ adapter has been developed which enables full LINQ-based access to model repositories, including Eclipse EMF. Transformations in LINQ have been developed for very different transformation tasks. The provided solutions and comparison to other popular transformation languages permit to conclude that LINQ is really usable as a transformation language. The expressiveness of LINQ is high enough to support various development styles for a task according to the developer preferences. The classical declarative rule-pattern style is supported on the basis of full functional programming support in LINQ. At the same time the procedural style by nested loops is supported as well, and both styles can be freely mixed, as required by the task.

The most popular programming language Java has been briefly analysed as well. Java 8 now contains the Streams API whose expressiveness is similar to LINQ. However, in order to use Streams API for model transformations it is necessary to enable Streams API access to model repositories – not only to standard Java collections as it is now. It could be one of future research tasks to develop an API for Java model repositories supporting the access via Java 8 Streams.

Acknowledgments

This work is partially supported by the Latvian National research program SOPHIS under grant agreement Nr.10-4/VPP-4/11.

In addition, authors would like to thank Sergejs Kozlovics for valuable consultations on the details of RA API.

References

Agrawal A., Karsai G., Shi F. (2003). Graph Transformations on Domain-Specific Models. *Technical report*, ISIS Vanderbilt University.

Albahari J., Albahari B.(2015). *C# 6.0 in a Nutshell*, 6th Edition. O'Reilly Media.

Barzdins, J., Rencis, E., Kozlovics, S. (2008). The Transformation-Driven Architecture. *Proceedings of DSM'08 Workshop of OOPSLA 2008*, Nashville, Tennessee, USA, 60 – 63.

Bezivin, J. (2011). *Why did MDE miss the boat?* In: Keynote at SPLASH 2011, https://modelseverywhere.wordpress.com/2011/10/17/why-did-mde-miss-the-boat/.

Bezivin, J. (2012). *History and Context of MDE.* http://www.nii.ac.jp/userimg/lectures/20120117/Lecture1.pdf.

Braun P., Marschall F. (2003) Transforming Object Oriented Models with BOTL. In: *Electronic Notes in Theoretical Computer Science* 72 (3), Elsevier, 103-117.

Chen, P. (1976). The Entity-Relationship Model - Toward a Unified View of Data. In: *ACM Transactions on Database Systems* 1(1).

Cook S., Jones G., Kent S., Wills A.C. (2007). Domain-Specific Development with Visual Studio DSL Tools. Addison-Wesley.

Dahm P., Widmann F (2003) GraLab - Das Graphenlabor. (German) In: *Projektbericht 4.3.0.*, University of Koblenz-Landau, Institute for Software Technology.

Eshuis R. (2005). Statecharting Petri Nets. *Technical Report Beta WP* 153, Eindhoven University of Technology.

Fischer, T., Niere, J., Torunski, L., Zündorf, A. (2000) Story diagrams: A new graph rewrite language based on the unified modeling language and Java. In: *Theory and Application of Graph Transformations*, LNCS 1764, Springer-Verlag, 296-309.

van Gorp P., Rose L.M., Krause C. (2013) *Proceedings of Sixth Transformation Tool Contest*, EPTCS 135. http://eptcs.web.cse.unsw.edu.au/content.cgi?TTC2013.

van Gorp P., Rose L.M. (2013). The Petri-Nets to Statecharts Transformation Case. In: *Proceedings Sixth Transformation Tool Contest (TTC 2013)*, EPTCS 135, 16–31.

Graf A., Voelter M. (2009) A textual domain specific language for AUTOSAR. In: *Model-based Development of Embedded Systems 2009*.

Gronback R. (2009). *ECLIPSE MODELING PROJECT A Domain-Specific Language Toolkit*. Addison-Wesley.

Hinkel G.(2013). *An approach to maintainable model transformations using an internal DSL*. Master's thesis, Karlsruhe Institute of Technology

Hinkel G., Goldschmidt T., Happe L. (2013). An NMF solution for the Petri Nets to State Charts case study at the TTC 2013, In: *Proceedings Sixth Transformation Tool Contest (TTC 2013),* EPTCS 135, 95 – 100.

Hutton G. (1992). Higher-order functions for parsing. *Journal of Functional Programming* 3(2), 323–343.

IBM (2004) *IBM Model Transformation Framework 1.0.0*, Programmer's Guide.

Ierusalimschy R., de Figueiredo L.H., Filho W.C. (2007). The evolution of Lua. *HOPL 2007*, 1-26.

Jakumeit E., Buchwald S., Kroll M. (2010) GrGen.NET. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(3), 263-271.

Jouault F., Kurtev I. (2005) Transforming Models with ATL. In: *Satellite Events at the MoDELS 2005 Conference*, LNCS 3844, Springer, 128-138.

Kalnina E., Kalnins A., Sostaks A., Celms E., Iraids J. (2012). Tree Based Domain-Specific Mapping Languages. In: Friedrich G., Gottlob G., Katzenbeisser S., Turan G., Bielikova M. (ed.) *SOFSEM 2012: Theory and Practice of Computer Science*, LNCS 7147, Springer Berlin / Heidelberg, 492-504.

Kalnins A., Barzdins J., Celms E. (2005) Model Transformation Language MOLA. In: *MDAFA'03 Proceedings of the 2003 European conference on Model Driven Architecture: foundations and Applications*, Springer-Verlag, 62-76.

Kalnins A., Kalnina E., Celms E., Sostaks A. (2009). From requirements to code in a model driven way. In: M. Kirikova, Y. Manolopoulus, L. Novickis J. Grundspenkis (Ed.) *Advances in Databases and Information Systems*, LNCS 5968, 161-168.

Kalnins A., Lace L., Kalnina E., Sostaks A. (2014). DSL Based Platform for Business Process Management. In: B. Preneel, B. Rovan, J. Stuller, A Min Tjoa V. Geffert (Ed.) *SOFSEM 2014: Theory and Practice of Computer Science*. LNCS 8327, Springer International Publishing, 351-362.

Kelly S., Tolvanen J-P. (2008) *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley, Hoboken.

Koenemann P., Nyssen A. (2013). *Model-based Automotive Software Development using Autosar, UML, and Domain-Specific Languages*. In: Embedded World Conference 2013.

Kolovos D., Paige R., Polack F. (2006). The epsilon object language (EOL). In: *Model Driven Architecture – Foundations and Applications (ECMDA-FA)*, LNCS 4066, Springer, 128–142.

Kolovos D., Rose L., García-Domínguez A., Paige R. (2016) The Epsilon Book, http://www.eclipse.org/epsilon/doc/book/.

Konigs, A. (2005) Model Transformation with Triple Graph Grammars. In: *Proceedings of MTIP05*, Springer.

Kozlovics S., Barzdins J. (2013). The Transformation-Driven Architecture for interactive systems. *Automatic Control and Computer Sciences* 47 (1/2013), Allerton Press, Inc., 28-37.

Lace L., Kalnins A., Sostaks A. (2014). Mappings for Process DSL Using Virtual Functional Views. In: Kalja A., Robal T., Haav H. M.(Ed.), *Databases and Information Systems : Proceedings of the 11th International Baltic Conference, Baltic DB&IS 2014*, 371-378.

de Lara J., Vangheluwe.H. (2002). AToM3: A Tool for Multi-Formalism Modelling and Meta-Modelling. In: *Fundamental Approaches to Software Engineering,* LNCS 2306, Springer, 174-188.

Lawley M., Steel J. (2005). Practical Declarative Model Transformation With Tefkat. In: *Satellite Events at the MoDELS 2005 Conference,* LNCS 3844, Springer, 139-150.

Liepins R. (2011). lQuery: A Model Query and Transformation Library. In: *Computer Science and Information Technologies, Scientific Papers, University of Latvia* 770, 27-46.

Liepins R. (2012). Library for model querying – lQuery. In: *Proceedings of 12th Workshop on OCL and Textual Modeling, OCL 2012*.

Liepiņš R. (2015). *Definition Methods and Implementation of Domain-Specific Modeling Language Tools*. PhD thesis, University of Latvia.

OMG (1997 a). UML Notation Guide, version 1.1, OMG document ad/97-08-05.

OMG (1997 b). Meta Object Facility (MOF) Specification, version 1.0, OMG document ad/97-08-14.

OMG (2002). Request for Proposal: MOF 2.0 Query / Views / Transformations. OMG document ad/2002-04-10.

OMG (2003). MDA Guide Version 1.0.1, OMG document omg/2003-06-01.

OMG (2008). Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, version 1.0, OMG document formal/2008-04-03.

OMG (2011 a). *Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) Specification*. Version 1.1, OMG document formal/11-01-01.

OMG (2011 b). *Unified Modeling Language (OMG UML)*, Superstructure, Version 2.4.1, OMG document formal/11-08-06.

Rencis E. (2008). Model Transformation Languages L1, L2, L3 and their Implementation. In: *"Computer Science and Information Technologies",* Scientific Papers 733, University of Latvia, 103-139.

Rose L., Kolovos D., Paige R., Polack F. (2010). Model migration with Epsilon Flock. In: Gogolla M., Tratt. L. (Ed.) *Theory and Practice of Model Transformations,Third International Conference (ICMT)*. LNCS 6142, Springer, 184–198.

Rumbaugh J.R., Blaha M.R., Lorensen W., Eddy F., Premerlani W. (1991). Object-Oriented Modeling and Design. Prentice Hall, 1991.

Schurr A., Winter A., Zundorf A. (1999). The Progress Approach: Language and Environment. In: *Handbook of Graph Grammars*, World Scientific, 487 -550.

Siegel J. and OMG (2001). Developing in OMG's Model-Driven Architecture. OMG document omg/01-12-01.

Śmiałek M., Nowakowski W. (2015). *From Requirements to Java in a Snap*. Springer.

Sproģis A., Liepiņš R., Bārzdiņš J., Čerāns K., Kozlovičs S., Lāce L., Rencis E., Zariņš A. (2010). GRAF: a Graphical Tool Building Framework. In: *ECMFA 2010 Tools and Consultancy track*.

Steinberg D., Budinsky F., Paternostro M., Merks E. (2008). EMF: Eclipse Modeling Framework, Second Edition. Addison-Wesley.

Sultana N. (20015). *Flick: A DSL for middleboxes*. In: DSLDI'15.

Taentzer G., Ehrig K., Guerra E., de Lara J., Lengyel L., Levendovszky T., Prange1 U., Varro D., Varro-Gyapay S. (2005). Model Transformation by Graph Transformation: A Comparative Study. *Proceedings of Model Transformations in Practice workshop at MODLES 2005*, Montego, Springer.

Tisi M., Martínez S., Jouault F., Cabot J. (2011). *Refining Models with Rule-based Model Transformations*. INRIA technical report.

Willink, E. (2003). UMLX : A Graphical Transformation Language for MDA. In: *Proceedings of MDAFA 2003*.

WEB (a). *MDR*. https://xml.netbeans.org/plans/model/mdr.html .

WEB (b). *Hype Cycle for Application Architecture*, Gartner, 2013. https://www.gartner.com/doc/ 2569522?ref=unauthreader.

WEB (c). *ReDSeeDS*. http://sourceforge.net/projects/redseeds/

WEB (d). *TIOBE programming language popularity index*. http://www.tiobe.com/tiobe_index.

WEB (e). *Modeling SDK for Microsoft Visual Studio 2015*. https://www.microsoft.com/ download/details.aspx?id=48148

WEB (f). *Code Generation and T4 Text Templates*. https://msdn.microsoft.com/en-us/library/bb126445.aspx

WEB (g). *Xomega*. http://www.xomega.net/

WEB (h). *NReco*. http://www.nrecosite.com/

WEB (i). *Model-Driven Development (MDD) and Novulo*. http://www-old.novulo.com/ ModelDriven.aspx

WEB (j). *.NET Modeling Framework (NMF).* https://nmf.codeplex.com/

WEB (k). *LINQ to Entities*. https://msdn.microsoft.com/en-us/library/bb386964(v=vs.110).aspx

WEB (l). XML Path Language (XPath). https://www.w3.org/TR/xpath/

WEB (m). *Java Streams vs C# LINQ vs Java6*.
http://blog.lahteenmaki.net/2013/04/java-streams-vs-c-linq-vs-java6.html .
WEB (n). *Java Streams Preview vs .Net High-Order Programming with LINQ*.
https://blog.informatech.cr/2013/03/24/java-streams-preview-vs-net-linq/.
WEB (o). *lambdaj*. https://github.com/mariofusco/lambdaj.
WEB (p). *jLinqer - Java implementation of LINQ*. https://github.com/k--kato/jLinqer.
WEB (q). *Jinq - database queries for Java 8*. http://www.jinq.org/.
WEB (r). Java Platform, Enterprise Edition (Java EE).
http://www.oracle.com/technetwork/java/javaee/overview/index.html.

Received September 21, 2016, accepted November 11, 2016