Transformation of UML Class Diagram to Internal Java Domain-Specific Language

Dmitry Buzdin[1], Oksana Nikiforova[2], [1-2]*Riga Technical University*

***Abstract* – the article addresses the existing problems found in the area of Domain-Specific Languages (DSLs). Being a widely adopted programming technique, DSL grammar creation process still lacks desirable traceability and automation. The paper proposes a sequential model transformation process based on Model-Driven Architecture concepts as one of the potential solutions to stated problem. One of the main results of the research work is the implementation of prototype of the full-cycle model transformation chain starting from the UML domain-model and ending with internal Java-based DSL grammar implementation.**

***Keywords* – code generation, domain specific, formal grammar, system architecture**

I. Introduction

Software complexity continues to grow with each year, and it is crucial to optimize the development process to follow the increasing demand. One of the ways for the development process optimization is avoiding accidental technical complexity. Accidental complexity is the complexity, which results from computer systems and is not essential to the problem domain. Accidental complexity is caused by the approaches or technologies used to solve the existing real-world problem [1]. Software development process should concentrate on gathering and specifying requirements rather than overcoming technical challenges. One of the sources of complexity is the use of traditional textual programming languages. As promised by Model-Driven Architecture (MDA) [2] the focus of the development should shift from code to models. There are different initiatives, which propose ways to make this happen. One of the promising directions in this field is the use of Domain-Specific Languages (DSLs).

DSLs [3] [4] are formal modelling or textual languages specializing in solving problems in dedicated solution space. In contrast to DSLs, General Purpose Languages (GPLs), such as C++ and Java, are suitable for solving any computational problem. Compared to GPLs, DSLs benefit from expressiveness and specialization in their problem area, reduce the verbosity and result in more comprehensive programs. The approach of creating specialized languages for certain problem domains was described back in 1986 [5].

DSLs are a recognizable way of raising the abstraction level and achieving productivity benefits as proven by existing research [6]. Several well-known DSLs are SQL (structured query language) and LATEX (document preparation language). As of today there are hundreds of different DSLs available, both small and large. One of the potential research directions related to DSLs is Language-Oriented Programming [7]. It proposes to create and reuse DSLs for existing problem domains systematically. Creation of a new programming language becomes an integral part of software development process. This is only possible when there are mature techniques and tools, making creation and evolution of the language as easy and straightforward as writing code in GPL.

Given paper proposes an approach to structure and ease DSL grammar creation and generation process. The main target of the research is to advance DSL approach to the level of generally acceptable development practice.

One of widely used DSL categorization techniques is splitting into two major types: external and internal languages [6]. External DSLs are completely independent of any existing programming language, thus free of syntax constraints imposed by those. Internal language approach, which is also known as language piggybacking [8], is based on the idea when the new language is being created on top of the existing GPL grammar. Internal language typically uses only a subset of syntax and features of the host language. Programs written in internal DSL are considered valid programs in the original GPL. Programs written in internal language are structured in a way to foster its readability and to hide unnecessary programming language details.

The article concentrates on usage of internal DSLs to raise the level of abstraction of computer programs. Currently, internal DSL grammar creation is a manual or semi-manual process also referred to as “grammarware hacking” [9]. The quality and structure of the resulting language are strongly dependent on the language designer’s personal skills and preferences. This imposes high technical risks, increases required software engineer’s qualifications and turns domain-specific language creation process into “black art”.

Internal domain-specific languages can be created using both statically typed languages like Java [10] and dynamic languages like Ruby [11]. One of the core programming activities during DSL design and implementation is its grammar definition preparation. Internal DSL grammar creation involves usage of text editors and integrated development environments (IDEs). None of these tools are specialized in DSL creation. There is a new breed of tools, which are intended to provide grammar definition and generation automation, such as MPS or openArchitectureWare [12] [13]. However, these tools cannot be considered generic ones, since they are only limited to one type of DSLs or a single technology platform. Usually these tools are built for definition of external languages and do not offer support for generation of internal DSLs.

Current situation in internal language development is the reliance on common design patterns and idioms [14]. The same approaches are used for building multiple internal languages. This is a manual process, but there is an opportunity for automation and reuse. Researchers try to increase the level of automation during creation and maintenance of DSLs by proposing both tools and techniques, which make DSL creation a reproducible and predictable process.

The goal of the paper is to offer an approach to ensure traceability and automated transformation from the problem domain model to internal DSL grammar implementation. The approach offered in the paper is specialized for using UML class diagrams [15] at the level of problem domain modelling. For demonstration purposes Java programming language is used for internal DSL grammar creation. Meta-modelling principles defined by Object Management Group for Model Driven Architecture [2] are used for definition of formal basis for model-to-text transformations.

The rest of the paper is structured as follows. Section II explains the research area, which is internal DSLs and transformation of UML class diagrams to working programs. It stresses the problematic aspects in the definition of DSL grammar. Section III proposes the solution offered by the authors. It defines the hypothesis that MDA principles for model abstraction and meta-modelling theory could be applied for the definition of internal DSL grammar. Section IV shows experiment results validating the stated hypothesis. In the last section of the paper, the authors stress important statements on the existing results and outline future research directions.

II. PROBLEM DEFINITION

Internal DSL creation in Java is a common practice used in many modern frameworks. Internal DSLs are also sometimes referred to as Fluent Interfaces. One of the widely used frameworks using such an approach is Apache Camel [16]. Apache Camel is a message routing and transformation engine, which provides reusable enterprise integration pattern implementations and hides underlying middleware specifics. Message routing definition in Camel is a typical example of internal DSL based on Java. DSL main purpose is to allow declarative definition of the message processing flow. The following example is important as it demonstrates the typical usage of internal DSLs in a variety of Java frameworks.

```
RouteBuilder builder = new RouteBuilder() {
  Public void configure() {
    errorHandler(deadLetterChannel(″mock:error″));
    from(″seda:a″).choice()
      .when(header(″foo″).isEqualTo(″bar″))
        .to(″seda:b″)
      .when(header(″foo″).isEqualTo(″cheese″))
        .to(″seda:c″)
      .otherwise().to(″seda:d″); }};
```

The content of the code is not as important as its structuring. Code composition approach in the example differs from traditional imperative Java programs. It is self-explanatory and benefits from expressive code indentation technique. The desired effect is achieved using language idiom called method chaining. Method chaining is an approach, when method calls on an object return the referred object instance itself as a result. This makes it possible to invoke the subsequent method immediately without terminating each line with a semicolon. This approach requires less code and application of code-completion capabilities of modern text editors in the optimal way. The provided example shows the boundaries of internal Java based DSLs. Even though the code is easier to read and understand to a non-programmer, there are a number of constraints mandated by the host language. Each expression is required to end with a semicolon. Curly brackets should separate classes and methods. Java compiler will not accept the program without mentioned rules being followed.

The same approaches and patterns for Fluent Interface implementation are used in other modern Java frameworks such as Google Guice [17]. Java language-based internal DSLs use in common the following design patterns and programming language idioms [18]:

- method chaining;
- static factory method;
- static import;
- builder pattern;
- variadic function;
- intermediate object.

Despite the fact that the same patterns, approaches and techniques are used to implement internal DSLs, grammar definition, to the authors’ knowledge, is still a manual process. It is possible that in some cases custom automation approaches are used during internal language composition. However, complete generation of internal DSL grammar code is still an open research field. It is possible to declare that there is no traceability or automated transformation from the source problem domain model to target DSL grammar implementation. In general this leads to the necessity of model-to-text transformation process.

When target language syntax and semantics are known, along with patterns for internal DSL creation, it is possible to automate the process of language grammar generation. It is required to provide domain description as an input for the automation in some generic form. It is proposed to take UML class diagrams as one of the implementation neutral formats. As far as UML class diagram is available for the problem domain, UML modelling tools can be examined for the support of definition of semantic part of internal DSL. UML tools by themselves do allow generation of initial class structure, and they do have a clean model to code mapping for concepts such as classes, attributes and associations.

However, modern UML tools (Rational Rose, SPARX Enterprise Architect, AgroUML, Magic UML etc.) generate only a structural code. Structural code is the direct representation of UML classes, attributes and methods to the corresponding elements in target object-oriented programming language. Programmer’s task is to implement a behavioural code, which is not being generated. Structural code is not sufficient for the task of internal DSL grammar generation. Incomplete transformation process burdens the programmers with a necessity to duplicate "lost" concepts in a host programming language. Reimplementation leads to the increased efforts and lack of traceability between source and target models. This also creates a risk of having design models not in sync with the source code. Furthermore, in some cases, it is necessary to maintain several versions of internal DSLs for different target platforms. The typical scenario for this requirement is cross-language frameworks, such as Apache Camel having APIs in Java, Scala and XML.

If the generated code is in Java programming language, the output obeys JavaBeans standard conventions. Each object property has getter and setter methods and no method chaining is present. According to internal DSL creation practices, it is necessary to make object construction concise and verbose. It is possible to achieve that by dropping unnecessary and duplicating code fragments with builder pattern implementation [14] for the used object model. The structural code being generated by modern UML tools is not intended as the basis of internal DSLs; thus, other techniques are explored.

Summarizing the problems stated above, the authors can define the goal of the paper in a more specific way, as to propose an approach, which fills the transformation gap between UML class diagrams and Java internal DSL grammar by generating not only a structural code, but also some part of the behavioural code. Therefore the proposed approach has to support the possibility to generate internal Java DSL implementation for creation of the object model using well-known internal DSL patterns.

The authors assume that principles of model abstraction and meta-modelling can be useful for solving the task stated above. Next section of the paper describes keynote assumptions for application of these principles for internal DSL grammar generation based on concepts from MDA [2].

III. Solution

There is no widely adopted tool at the moment, which would automatically bridge system analysis artefacts, such as domain models, to DSL grammar definition. Problem domain description captured in unambiguous specification language, such as UML, can be used to derive and generate DSL grammar, either internal or external. This automates the process of grammar creation and decreases length of software development cycles. End-to-end processing transformation is required in order to eliminate domain knowledge duplication in two different forms – as a model and language grammar.

One of the ways for solving any problem is to find similar problems and existing approaches to solve them. The authors propose a hypothesis that the core principles of MDA could serve as a formal basis for solving the described model transformation problem. This approach is also called potentially effective within DSL context in [9], where application of MDA concepts to DSL generation is mentioned. It is possible to view generic grammar representation as a Platform-Independent Model (PIM) and concrete grammar implementation as a Platform-Specific Model (PSM). One of the core MDA principles is the abstraction of details, which are irrelevant on the current level. MDA brings the concept of meta-levels for describing model transformation process. Meta-Object Facility (MOF) [19] is the standard describing meta-modelling architecture in model-driven development. MOF describes a four-layered meta-modelling architecture. Level M0 represents the problem domain itself; level M1 is the model of the problem domain, such as UML class diagram; level M2 describes modelling language semantics, such as UML; level M3 is the language of MOF to build meta-models.

The authors propose to reuse the same multi-level concept for producing DSL grammars. DSL generation approach is based on the general model transformation scenario previously described in [20]. Steps required to transform the existing domain model to language grammar are depicted in Figure 1.

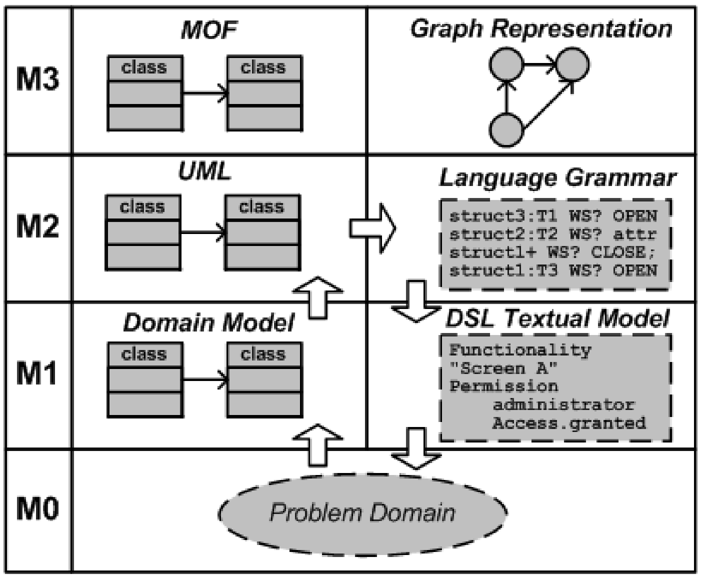

Fig.1. Model Transformation Process

There are two separate vertical meta-modelling stacks. One is for the source model and the second one – for the target model. Actual problem domain concepts reside on level M0.

Both UML class diagram and DSL textual model on level M1 describe the problem domain. DSL model is an internal textual DSL describing the same problem domain as the corresponding UML model.

UML meta-model is shown on M2 on the left side. Implementation of specific language grammar is on the right side of M2. In case of internal DSL, this is a set of instructions expressed in a host programming language. In case of external DSL, this would be a grammar definition in EBNF or a similar form.

The left part of the level M3 contains MOF as a generic meta-modelling framework for use with UML. The right part of the level M3 depicts graph representation of the DSL grammar, which may be used in order to provide a platform-independent way of expressing results. It is important to note that graph representation is platform-independent and can be used to generate grammar definition in the technology of choice.

It is proposed to move from UML to DSL lane on the level M2 since the meta-model allows transformation into the corresponding textual meta-model, which is a language formal grammar. MOF and graph representation serve as the basis for this transformation as those provide meta-modelling facilities. It is impossible to do the move on the level M3, as there is no information about the problem domain. If the transformation were done on the level M1, it would be necessary to derive the meta-model from the domain model first to produce the language grammar.

In order to transform UML domain model to a platform independent format (PIM), first, it is necessary to receive the initial graph representation of the class diagram. Algorithm of UML diagram transformation to the directed labeled graph is previously addressed in [21]. The same approach could be reused as the first step of the transformation. Graph representation allows applying formal modifications to the existing structure.

In order to perform the transformation on the level M2, additional input is required. UML model corresponds to the target language semantics, but syntax should be defined elsewhere. It is possible to define a set of graph rewriting rules, which would enrich the core language concepts with additional syntax elements. In order to get to the desired language style graph enrichment is performed by appending new nodes.

When the resulting graph is prepared, it is possible to transform that into textual grammar representation. Any available model to text transformation technology can perform the task. The simplest approach is to use a programming technique called template processing. In template processing the text is composed of two components: static textual templates with placeholders and macros and the model, provided in a generic format. The resulting internal grammar definition, which is an output of template processing, is used as a part of the system source code and is being compiled or interpreted in runtime.

The outlined solution builds on top of several existing research papers. Definition of bi-directional transformation algorithm between MOF meta-models and context-free grammars, which is taken as the reference, is provided in [22]. However, the paper demonstrates mapping between existing GPL and MOF, which is a different scenario than DSL and MOF proposed by the authors. The problem of bridging meta-models and language grammars has been addressed before in [23]. The approach is limited to external DSLs and does not use UML as a meta-modelling language. Transformations between different model formats have been analyzed in several works. Topic of UML and DSL model transformation was previously covered in [24]. The difference is that the authors propose to map UML meta-model to DSL grammar, whereas [24] describes mapping of UML model to DSL model. Object-oriented model structure refinement via graph transformation rules is described in [25]. This approach is used for model enhancement during transformation sequence. Reverse approach of transforming the language grammar to UML model is proposed in [26]. Transformation process in the referenced article is also backed-up by MOF meta-model, but the direction is opposite of the described one.

IV. Example

The proposed approach is demonstrated on the example of a non-trivial problem domain. The domain complexity is sufficient to demonstrate common cases of conceptual modelling. The example shows the flow of model transformation starting with UML class diagram, representing the domain model and ending with generated internal Java-based DSL implementation. The demonstrated example is simplified, but contains all major elements from large-scale problem domains (classes, attributes, associations). The example is based on hotel room booking domain, where core entities are the following: rooms, bookings, payments and guests. UML Class diagram used as an input analysis artefact is shown in Figure 2.

Booking class is a central piece of the domain and has several associations with the other classes. The purpose of the new DSL is to cover the process of registering new bookings in the existing system and adding all required information to them.

The example describes the process of creation of internal DSLs based on Java programming language [27]. Java has been chosen due to the following reasons:

– language popularity [28];
– wide use of internal DSLs in frameworks;
– mature patterns and idioms for internal DSL creation.

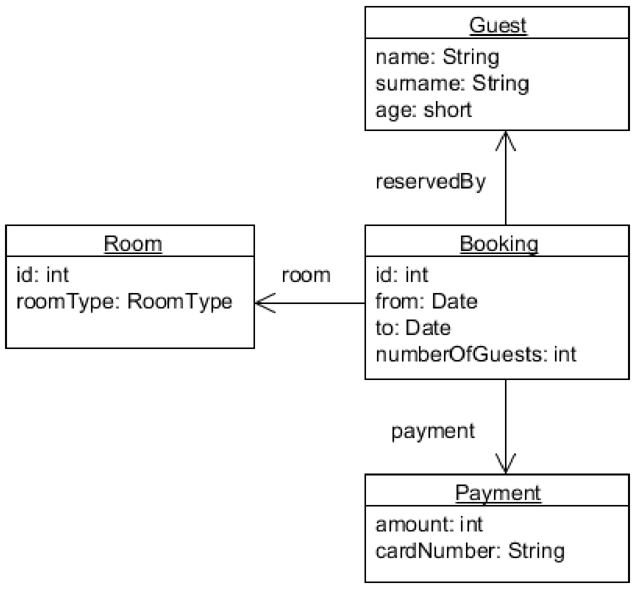

Fig.2. Class Diagram of the Domain Model

Internal DSL generation example has been automated as part of the research. Overall transformation sequence is depicted in Figure 3. In order to read the class model for transformation purposes it is proposed to read the model specification using the XML Metadata Interchange (XMI) format, supported by the most of UML tools [29]. XMI format is a textual, XML based language, which is used to exchange with UML models in a vendor independent format. It is possible to extract all necessary information from the class diagram by processing XMI file, which is exported from the modelling tool. After the extraction the domain model gets programmatically transformed into graph representation. Graph model serves as an intermediate step, suitable for further transformation to the chosen grammar definition type and language style. Therefore, transformation rules determine semantics of the language and enrich the domain model with an additional structure.

After the transformation and application of graph rewriting rules the graph is transformed into Java classes using FreeMarker template processing library [30]. This is a model to text transformation step. The resulting Java code forms the necessary building blocks for the internal DSL. The generated code is created in accordance with the mentioned design patterns for internal DSL creation. Text templates define the syntax of the generated language. By changing the templates, it is possible to produce results for other host programming languages or change the style of the generated language.

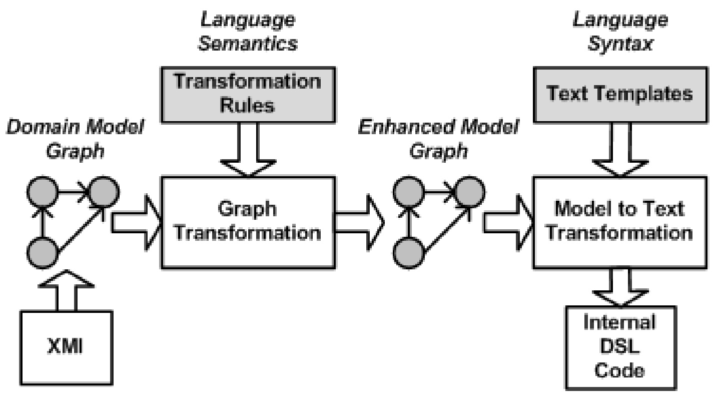

Fig.3. Model Transformation Steps

Transformation rules and text templates form the unique language type and can be reused for different domain models. In order to generate a full solution, the following steps should be completed:

– Generate JavaBeans for domain classes. It can be done with the existing UML tools.

– Build a fluent interface definition. The tool implemented during the research performs this successfully.

The described model transformation sequence outputs the source code of internal Java-based DSL, which in combination with domain classes, could be used as an internal DSL. The grammar implementation, which is necessary for the shown code to be compiled and run, is generated automatically out of UML class diagram previously shown in Figure 2. The resulting program excerpt, which uses the internal DSL, is demonstrated below.

```
Booking booking = booking()
  .id(123456l)
  .from(new Date()).to(new Date())
  .room(room()
    .number(23)
    .roomType(Room.RoomType.FAMILY)
    .build())
  .reservedBy(guest()
    .name(”John”)
    .surname(”Smith”)
    .age(34)
    .build())
  .payment(payment()
    .amount(new BigDecimal(100))
    .cardNumber(”12345678”)
    .build())
  .noOfGuests(2)
  .build();
```

When running the sample code, it is possible to receive a new booking object populated with all stated information. The source code is being compiled without problems due to the fact that required supporting classes and methods have been generated out of the domain model description. The resulting code uses well-known patterns and techniques (such as method chaining and builder pattern) used in modern frameworks, where an internal DSL definition is hand-crafted.

It is possible to compare the resulting DSL code with the traditional coding approach.

```
Booking booking = new Booking();
booking.setId(123456l);
booking.setFrom(new Date());
booking.setTo(new Date());
Room room = new Room();
room.setNumber(23);
room.setRoomType(Room.RoomType.FAMILY);
booking.setRoom(room);
Guest guest = new Guest();
guest.setName(”John”);
guest.setSurename(”Smith”);
guest.setAge(34);
booking.setReservedBy(guest);
Payment payment = new Payment();
payment.setAmount(new BigDecimal(100));
payment.setCardNumber(”12345678”);
booking.setPayment(payment);
booking.setNoOfGuests(2);
```

Both DSL and traditional coding approaches produce the same result. However, internal DSL approach is smaller in size and is more readable. Language composition patterns used in a given example are equivalent to those used in modern Java frameworks. Therefore, it is possible to state that the proposed approach is applicable for solving real-world problems. With the proposed approach it is possible to automate the creation of internal Java DSLs based on the provided UML domain meta-model. This transformation step automates the routine, which was previously done manually.

The generative language grammar definition approach facilitates model reuse and platform independence. In the given example, the approach is demonstrated using the pre-defined technology stack. The approach can be ported to another host language other than Java, or support several code generation targets. It is possible to change the generated grammar output by modifying the model transformation rules to rely on a different set of language building patterns and idioms. It helps to achieve separation of platform independent models from platform specific models.

V. Conclusions

The main novelty of the research is the description of a process of automated conversion of UML-based domain model into internal DSL grammar. The approach is the original contribution explaining the conceptual approach of the transformation chain. The approach has been prototyped successfully using a simplified domain model as the input and an internal Java-based DSL grammar as the output. The prototype itself has been developed in Java programming language.

Unlike the existing DSL definition tools, which concentrate on one or several predefined grammar description formats, the proposed approach is extensible and independent both from technology and platform specifics. The proposed transformation sequence benefits from meta-modelling concepts inspired by MDA and MOF. The same approach was previously used in [20] to generate external DSL grammar definition using ANTLR [31] grammar format. These two experiments have led to the conclusion that the proposed approach is feasible both with internal and external DSL grammar creation.

The solution developed for generation of DSL grammar can be used as a starting point for creation of a new language. This could be done after the domain analysis is conducted. When the changes to the existing domain model are required, it is possible to change the source model itself and re-generate the grammar instead of applying manual modifications. Implementation of the internal DSL will follow widely accepted patterns and practices and will be cleanly separated from a hand-written code. The limitations of the demonstrated example are the lack of constraint checks on the model and limited flexibility of graph rewriting rules, since those are implemented in an imperative fashion.

Further research will be concentrated on testing and adapting the proposed transformation process to the different implementation platforms and grammar definition formats. One of the next steps of the generator is to start processing constraints defined in the source model, such as OCL expressions. Another challenge is to produce an extensible transformation engine, which is able to support multiple DSL types and technical platforms as its output targets.

Acknowledgements

We thank the organizers of conference "Engineering for Rural Development" from Technical faculty of Latvia University of Agriculture for permission to use their publication template for our needs. The research presented in the paper partly is supported by Grant of Latvian Council of Science No. 09.1245 "Methods, models and tools for developing and governance of agile information systems".

References

[1] F. P. Brooks, *The Mythical Man-Month: Essays on Software Engineering*, Addison Wesley, 1995.
[2] OMG, "Model-Driven Architecture Guide Version 1.0.1." OMG, 2003.
[3] W. Taha, "Domain-Specific Languages," *The 2008 IEEE International Conference on Computer Engineering and Systems (ICCES 2008)*, 2008.
[4] A. V. Deursen and P. Klint, "Little languages: Little maintenance?" *Journal of Software Maintenance*, 1998.
[5] J. Bentley, "Programming Pearls: Little languages," *Communications of the ACM*, 29(8):711721, 1986.
[6] M. Fowler, *Domain-Specific Languages*, Addison Wesley, 2010.
[7] S. Dmitriev, "Language-Oriented Programming: The next programming approach," *On-Board*, 2004.
[8] D. Spinellis, "Notable Design Patterns for Domain-Specific Languages," *Journal of Systems and Software*, vol. 56, pp. 91–99, Feb. 2001.
[9] P. Klint, R. Lammel, and C. Verhoef, "Toward an Engineering Discipline for Grammarware," *ACM Transactions on Software Engineering Methodology*, vol. 14, no. 3, pp. 331–380, 2005.
[10] S. Freeman and N. Pryce, "Evolving an Embedded Domain-Specific Language in Java," OOPSLA 2006.
[11] H. C. Cunningham, "A Little Language for Surveys: Constructing an Internal DSL in Ruby," *ACM-SE 46: Proceedings of the 46th Annual Southeast Regional Conference on XX, (New York, NY, USA)*, pp. 282–287, ACM, 2008.
[12] JetBrains, "Meta-Programming System." Internet, 2009. http://www.jetbrains.com/mps/. [Accessed September, 2011].
[13] S. Effinge, M. Voelter et al., "openArchitectureWare." Internet, 2008. http://www.openarchitectureware.org/. [Accessed September, 2011].
[14] Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1994.
[15] OMG, "OMG Unified Modeling Language Specification" OMG, 2003.
[16] Apache Foundation, "Apache Camel." Internet, 2010. http://camel.apache.org/index.html. [Accessed September, 2011].
[17] Google, "Google Guice." Internet, 2010. http://code.google.com/p/google-guice/. [Accessed September, 2011].
[18] Jeff Bay and Alex Ruiz, "An Approach to Internal Domain-Specific Languages in Java," Internet, 2008. http://www.infoq.com/articles/internal-dsls-java. [Accessed September, 2011].
[19] OMG, "Meta Object Facility (MOF) Specification, version 1.4" OMG, 2002.
[20] D. Buzdin, "Generative Approach to DSL Grammar Definition," *in Proceedings of 43rd Spring International Conference MOSIS'09, Modeling and Simulation of Systems* (J. Stefan and P. Peringer, ed.), (Ostrava), 2009.
[21] P. Ziemann, K. Holscher, and M. Gogolla, "From UML Models to Graph Transformation Systems," *Electronic Notes in Theoretical Computer Science*, 127, pp. 17–33, 2005.
[22] M. Alanen and I. Porres, "A Relation between Context-Free Grammars and Meta Object Facility Metamodels," *TUCS Technical Report* No 606, 2003.
[23] F. Jouault, J. Bezivin, and I. Kurtev, "TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering," *GPCE06*, 2006.
[24] M. Wimmer, A. Schauerhuber, M. Strommer, W. Schwinger, and G. Kappe, "A Semi-Automatic Approach for Bridging DSLs with UML," *OOPSLA'07*, 2007.
[25] X. Liu, Z. Liu, and L. Zhao, "Object-Oriented Structure Refinement - a Graph Transformation Approach," *Electronic Notes in Theoretical Computer Science* 187, pp. 145–159, 2007.
[26] A. Kunert, "Semi-Automatic Generation of Metamodels and Models from Grammars and Programs," *Electronic Notes in Theoretical Computer Science*, vol. 211, pp. 111–119, 2008.
[27] Oracle, "Java." Internet, 2010. http://www.java.com/en/. [Accessed September, 2011].
[28] Tiobe Software, "Tiobe Programming Community Index." Internet, 2011. http://www.tiobe.com/tpci.htm. [Accessed September, 2011].
[29] OMG, "XML Metadata Interchange." Internet, 2007. http://www.omg.org/spec/XMI/. [Accessed September, 2011].
[30] FreeMarker, Internet, http://freemarker.sourceforge.net/. [Accessed September, 2011].
[31] T. Parr, *The Definitive ANTLR Reference*. Pragmatic Programmers, 2007.

Dmitry Buzdin
Educational background
B.sc. in computer science, 2005
M.sc. in computer science, 2007
Work experience
Software Developer, 2005 – 2006
Software Architect, C.T.Co Ltd., 2006 – now
Past research interests: modelling languages, design patterns, object-oriented programming. Current research interests: domain-specific languages, code generation, model transformation.
E-mail: buzdin@gmail.com

Oksana Nikiforova received the doctoral degree in information technologies (system analysis, modelling and design) from Riga Technical University, Latvia, in 2001.
She is presently a Full Professor at the Department of Applied Computer Science, Riga Technical University, where she has been on the faculty since 1999. Her current research interests include the object-oriented system analysis and modelling, especially the issues in the framework of Model Driven Software Development (MDSD). She has published extensively in these areas and has been awarded several grants. She has participated and managed several research projects related to the system modelling, analysis and design, as well as participated in several industrial software development projects.
She is a member of RTU Academic Assembly, Council of the Faculty of Computer Science and Information Technology, RTU Publishing Board, RTU Scientific Journal Editorial Board, etc. She is a co-chair of workshops focused on MDSD – MDA 2009 in conjunction with ADBIS, MDA&MTDD 2010 and MDA&MDSD 2011 in conjunction with ENASE. She was awarded as RTU Young Scientist of the Year 2009.
E-mail: oksana.nikiforova@rtu.lv