Practical Experiments with Code Generation from the UML Class Diagram

Janis Sejans and Oksana Nikiforova

Riga Technical University, Faculty of Computer Science and Information Technology
Meza ¼, Riga, LV 1048, Latvia

Abstract. The paper turns an attention to the problems of code generators in advanced CASE tools from the UML class diagram. Authors give a general introduction to code generator types, describes their structure and principles of operation. Three tools are analyzed within the correspondence to their abilities to generate program code from the UML class diagram. They are two modelling tools, namely, Sparx Enterprise Architect and Visual Paradigm, and the programming environment Microsoft Visual Studio .NET. Program code is generated from different fragments of the UML class diagram in all three tools and the obtained code lines are compared with the expected ones based on the model semantics and syntax of the programming language C#. Authors summarize the results of the practical experiments with code generation by stressing different types of errors in the generated code and make conclusion about the directions of the evolution of code generators in the close future.

1 Introduction

Despite high levels of IT technology and IT fields, as such, the development problems that have traditionally attributed to the development of software is still not completely solved. Of course, development time and costs may always want to decrease, but this must be understood that there is still much time that is devoted to routine work. For example, if the insurance system is developed, often the system is designed from the beginning, despite the fact that specificity of the problem domain is not so much changed and why the initial descriptions and components that were created previously could not be reused. Therefore, reusability is still an important theme, because technology is changing much faster than business processes of the problem domain. The first reason, why this problem is not yet solved, is the technological diversity, which in turn has an effect, that every system is over and over again overwritten in a case of the change in the technological architecture. Secondly, even if the problem domain is described using "traditional" CASE tools, it does not solve the problem, because created models is nothing more than documentation, so they are not automatable to be transformed in executable program code.

Even more, further requirements are implemented directly in the code bypassing the system documentation and thus it does not longer meets the actual functionality. Thus, the fundamental problem of software development is a "semantic gap" between

Sejans J. and Nikiforova O..
Practical Experiments with Code Generation from the UML Class Diagram.
DOI: 10.5220/0003581300570067
In *Proceedings of the 3rd International Workshop on Model-Driven Architecture and Modeling-Driven Software Development* (MDA & MDSD-2011), pages 57-67
ISBN: 978-989-8425-59-1

models and programming language, which permits the transformation of the problem domain description into the executable code.

Unified Modelling Language (UML) [1] was created not only as a system specification tool, but is positioned also as a mean, which will allow automatically generate code from UML models. With such a position Object Management Group proclaimed its new invention, Model Driven Architecture (MDA) [2], in the end of 2001. Just that it will be possible to generate a software system from the thoroughly developed model of the problem domain. Since then, it is now already 10 years, and 15 years since UML was standardized. At that time, a lot of different CASE tools have been developed, which are advertised as more or less able to generate the program code from system model. These are both open source tools, and commercial products. While still not yet heard about a software system that could be developed based only on principles of MDA.

Inadequate models and lack of the formalization of modelling process is considered as one of the MDA implementation disincentives [3]. About 100 different techniques, methodologies, approaches, transformation algorithms has been developed in the last 10 years, which makes it possible to automate the creation of UML class diagrams, from which the subsequent code generation is defined as it is already solved the problem. [4] presents an analysis of different approaches to transformation of the problem domain description into the UML class diagram during last 10 years, published in four digital libraries – IEEEXplore, ACM, Science Direct and SpringerLink. The survey states, that there exist enough different approaches for the generation of the UML class diagram. So far different solutions are offered for making the process of the class diagram development more suitable, more formal, or even more "user-friendly" [3]. Therefore, the authors doubt about the abilities to use class diagram elements for further generation of software components also in [3].

The authors of this paper state to investigate exactly the stage of code generation, because there are a very few works, which are devoted to deep analysis of code generation abilities in advanced CASE tools. Eichelberger with colleagues in [5] has been evaluated current UML modelling tools including the products of all major players being relevant to industry and academia. The study is meant to provide an overview of UML modelling tools as well as decision support to potential buyers and users. One of the evaluated aspects also is the code generation. Several researchers have been introduced some mechanisms for code generation improvements, like [6] and [7]. But it is still a fact, that there is no tool, where according to [5] code generation ability would be evaluated closed to 100%. But authors of [5] themselves admit that, the main focus of this study is on the availability of the modelling capabilities as defined in the UML specification. So far the analysis of code generation possibilities from the UML class diagram using advanced CASE tools is selected as a research object.

2 General Principles of Code Generators

Code generators and the idea of code generation itself is nothing new during the last 30 years. In fact, currently popular software development environments like Visual

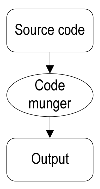
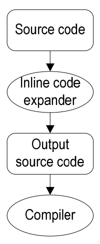
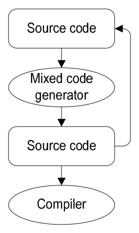
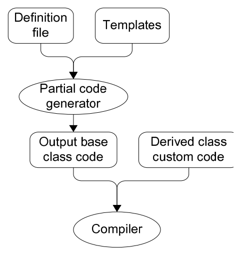
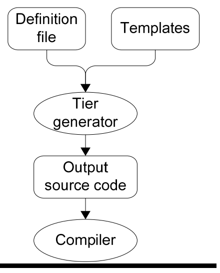
Studio and Eclipse are generating part of the source code thru snippets, templates and other GUI interfaces. Well-known example of code generation are GUI forms and dialog wizards, which in result allow the developer to not think about control coordinates in window or connection between entities (because this part of the code is generated).

Herrington in [8] defines the field of code generation as it is about writing programs that write programs. With today's complex code-intensive frameworks, such as Java 2 Enterprise Edition (J2EE), Microsoft's .NET, and Microsoft Foundation Classes (MFC), it's becoming increasingly important that software developers use their skills to build programs which aid developers themselves in building their applications. What can be generated with code generators at all? Basically anything that you can think of as automatable and for which you can describe the structure and define a template. It is possible to generate code structures, skeletons, accessor methods, scripts for database, etc. Also, it is possible to generate classical unit tests, CRUD (create, read, update, delete) operations, so there is no need to hand-code them, as well it is possible to generate documentation and code analysis based on the source code. Furthermore generators can be used to convert one form of information into another representation, for example from text file into HTML form. So as stated above – anything for which the template can be created.

Code generators are divided into two categories: active and passive. The passive code generators are often implemented as wizards, where the configuration parameters are set and the result is generated. These generators are not taking any responsibility on further corrections or maintenance of the generated code. In turn the active code generators are taking the responsibility of the generated code and allow make corrections and regenerate the result, so it is updated.

It is possible to distinguish five different types of active generators, though they are not isolated and distinct to each other, but complement [8]. All of them are schematically described in Table 1.

Table 1. The essence of different types of code generators.

Code munger	**Inline-code expander**	**Mixed-code generator**	**Partial-class generator**	**Tier or layer generator**
				

1) Code munger (see in Table 1) is the simplest type of the active generators. „Munging" means transforming something from one form into another form, thus schematically the generator has an input information, then it is processed and the output form is created. This kind of generator can be used to generate documentation, retrieve and collect some specific information or generate some kind of the input

analysis result.

2) An Inline-code expander (see in Table 1) reads the source code and, where it finds the predefined mark-up, it inserts the production code. The difference to code munger is that the output is the same as input, but with corrections - additions. This kind of generator is used to embed SQL code, ASM code or some security code.

3) Mixed-code generator (Table 1) is a combination of the previous two types. The difference is that the generator output can be used as input (modifications are done in the input file), so the result can be regenerated. Similarly to inline-code expander the generator reads the source code and finds the predefined mark-up, but instead of inserting production code, the mixed-code generator replaces area in between the mark-up. This way the mark-up is left in the result (output) and it can be used as input again.

4) The partial-class generator input is definition file (input information) and template files by analysis of the definition file, the template is filled and the output is produced. The template file would contain special mark-up keywords which would be replaced according to the input and thus producing the new output (dependant on input information). The partial-class generator (see in Table 1) is used for generating base structures or base classes which afterwards are manually updated with final functionality. In case of n-tier application the partial class generator could generate part of tier code.

5) Tier or layer generator (see in Table 1) is similar to partial-class generator, except it takes the responsibility to completely generate one tier of an n-tier application. The generated code has enough functionality for working tier or layer, not only base classes. Tier or layer generator can be positioned as a model-driven generator, where the UML model is an input and one or more tiers of the system is an output. Tools for code generation examined in the next section are of this type of code generators.

3 Analysis of Code Generation in Several CASE Tools

The number of modelling tools with an ability to generate program code from different types of the diagram is increasing. Currently, there are as well as powerful commercial products, as equivalent open source analogues. Often there is a problem to choose an appropriate tool, even if it is mentioned in the descriptions of tools that they are model-driven software development tools and are able to generate program code.

The authors of this paper made a research of system modelling tools with the aim to identify which programming languages are supported for which code generation. The results of [9] shows, that the number of declared programming language is large enough. However, the answer on the question, whether modelling tools are able to generate high-quality programming code, is the goal of this article.

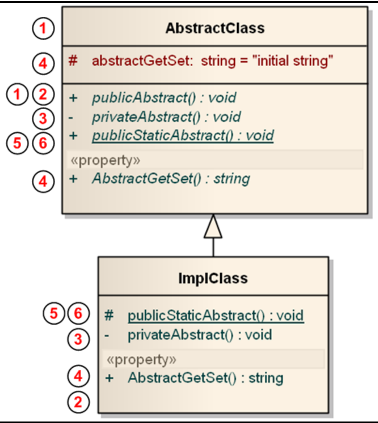
3.1 Experiments with Code Generation from the Fragment of the UML Class Diagram

Three tools are chosen and used for the experiments. They are Sparx Enterprise Architect and Visual Paradigm for UML, which seems to have wider spectrum of abilities to support different programming languages [9], and thus possibly better built-in code generator. Both of these tools are mainly recognized as UML modelling tools with a capability to generate program code, compared to Visual Studio, which is a development environment. In author`s opinion a tool which is a development environment could have more support for required modelling structures from the code perspective, as well as better algorithms for code generation from the models. Given that the latest Visual Studio 2010 have ability to design UML diagrams and generate program code, it was chosen as a third tool.

The input for code generator is a fragment of the UML class diagram. The output or a target programming language is C#. For code generator testing, the black-box testing strategy is chosen, that means we provide the input and analyse the output. The input model is created based on the knowledge of the notation of UML elements and its semantics (from UML specifications) and on the other hand, the knowledge about C# rules and its syntax. The sum of this allows create a kind of a UML class diagram test models [10] for examining the code generator by checking the generated source code correspondence with model notation semantics and appropriate syntax of programming language. In parallel with the test model [10] the expected source code file is created. The generated result is compared to the expected result, which allows to determine whether the generated source code corresponds to the model semantics. The comparison allows reveal the information loss and noncompliance with syntax rules.

For demonstration purposes we will describe and illustrate code generation results from an abstract class and its derived class. Based on the purpose, definition and programming language C# rules of an abstract class, the rules – requirements defined in Table 2 should be applied.

Table 2. Rules and test model for definition of an abstract class in UML.

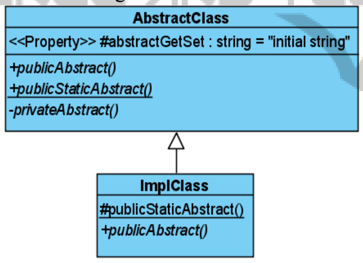
Rules to be applied	Test model for an abstract class
1. If class contains at least one abstract method, then it must be marked as ***abstract***; 2. A non-abstract class that is derived from an abstract class must include implementations of all inherited abstract methods, using ***override*** keyword. 3. Because an abstract method must be overridden in the derived class, then it must not be ***private***; 4. Because an abstract method has no method body, then the method declaration ends with semicolon (must not end with curly braces), unless it is an accessor method; 5. While overriding an abstract method, the access modifier must be same as for the overridden base method, e.g. if it is ***public***, then in the derived class it can not be ***protected***, because it must be ***public***; 6. It is an error to use the ***static***, ***virtual***, ***sealed*** or ***new*** modifiers along with ***abstract*** keyword.	 1 AbstractClass 4 # abstractGetSet: string = "initial string" 1 2 + publicAbstract() : void 3 - privateAbstract() : void 5 6 + publicStaticAbstract() : void «property» 4 + AbstractGetSet() : string ImplClass 5 6 # publicStaticAbstract() : void 3 - privateAbstract() : void «property» 4 + AbstractGetSet() : string 2

To test the knowledge of code generator on C# rules, each of the mentioned statements (rules) is created in model in an inverse form. For example, if class contains at least one abstract method, then it must be marked as ***abstract***, in the model we define an abstract method ***publicAbstract***, but we don`t mark the class as ***abstract***. As a result we expect, that the code generator will be smart enough and by knowing this rule, will automatically mark the class as ***abstract***.

Since the experimental model (see left column of Table 3) contains erroneous structures, it is fine in case a tool doesn`t allow to create such structure. Because that means that the tool is checking the correctness of the model, and that may result into more precise source code. Unless the restriction is related to missing notation (feature), which means the generated source code will not contain required structure at all (because there is no way to define it). The expected source code is shown in the right column of the Table 3. By compiling the generated source code and comparing it with the expected one, the results are classified into the following categories: Compilation error, Execution error, Information loss, Missing notation, Different and Correct.

Table 3. Fragment of experimental model defined in the investigated tools.

Definition of the abstract class by the UML class diagram	Expected source code for an abstract class test model

Visual-Paradigm for UML:  AbstractClass
<<Property>> #abstractGetSet : string = "initial string"
+publicAbstract()
+publicStaticAbstract()
-privateAbstract()
ImplClass
#publicStaticAbstract()
+publicAbstract()
The same fragment is developed also in SPARX Enterprise Architect and Microsoft Visual Studio.
	`using System;`
	`public abstract class AbstractClass {`
	`  protected string abstractGetSet = "initial`
	`string";`
	`  public abstract string AbstractGetSet {`
	`    get;`
	`    set;`
	`  }`
	`  public abstract void publicAbstract();`
	`  public abstract void PrivateAbstract();`
	`  public abstract void PublicStaticAbstract();`
	`}`
	`public class ImplClass : AbstractClass  {`
	`  public override string AbstractGetSet {`
	`    get {`
	`      return abstractGetSet;`
	`    }`
	`    set {`
	`      abstractGetSet = value;`
	`    }`
	`  }`
	`  public override void PublicStaticAbstract() {`
	`    throw new System.NotImplementedException();`
	`  }`
	`  public override void PrivateAbstract() {`
	`    throw new System.NotImplementedException();`
	`  }`
	`  public override void PublicAbstract() {`
	`    throw new System.NotImplementedException();`
	`  }`
	`}`

The analysis of the generated source code for an abstract class test model is shown in Table 4. The analysis of the results defined in terms of numbers of error's categories shows that the Enterprise Architect has the worst result, only one of six tests is correct (five are defined as containing compilation errors). Visual-Paradigm has essentially different result, only one of six tests is failed. The third tool – Visual Studio, didn`t meet our expectations, generated source code contains four compilation errors, one information loss and one correct result.

Table 4. Code generation result in each of the tools (error keywords are underlined).

Tool	Result category and comment	Transformation result
Class contains an abstract method, but it isn`t marked as *abstract*		
VP	Correct – class marked as ***abstract***	`public abstract class AbstractClass`
EA	Compilation error – missing ***ab-stract***	`public class AbstractClass`
VS	Correct – class marked as ***abstract***	`public abstract class AbstractClass`
Class contains an abstract method with *private* **access**		

VP	Correct – automatically converted into ***public*** access, same in derived class	`public abstract void PrivateAbstract();`
		
		`public override void PrivateAbstract()`

EA	Compilation error – ***private*** access is left in abstract class and in derived class	`private abstract void privateAb-stract();`
		
		`private override void privateAbstract()`

VS	Compilation error – ***private*** access if left in abstract class and in derived class; missing ***override*** in derived class method	`private abstract void privateAb-stract();`
		
		`private void privateAbstract()`

Class contains an *abstract* **method with** *static* **modifier**		

VP	Correct/Compilation error – automatically removed ***static*** modifier in abstract class, but not in derived class	`public abstract void PublicStaticAbstract();`
		
		`protected static override void PublicStaticAbstract()`

EA	Compilation error – ***static*** modifier is left in abstract class and in derived class	`public abstract static void publicStaticAbstract();`
		
		`protected static override void publicStaticAbstract()`

VS	Compilation error – ***static*** modifier is left in abstract class and in derived class; missing ***override*** in derived class method	`public static abstract void publicStaticAbstract();`
		
		`protected static void publicStaticAbstract()`

Class contains *protected* **attribute with abstract accessors (***get-set*** method)**		

VP	Different – abstract class doesn`t contain ***protected*** attribute, only abstract accessor method, instead attribute appears in derived class with ***private*** access	`//abstract class`
		`public abstract string AbstractGetSet { get; set; }`
		`//derived class`
		`private string abstractGetSet = "ini-tial string";`
		`public override string AbstractGetSet {`
		`  get {`
		`    return abstractGetSet;`
		`  }`
		`  set {`
		`    abstractGetSet = value;`
		`  }`
		`}`

Table 4. Code generation result in each of the tools (error keywords are underlined). (cont.)

Tool	Result category and comment	Transformation result
EA	Correct – class contains ***protected*** attribute and ***abstract*** accessor method; derived class overrides the accessor method	`//abstract class` `protected string abstractGetSet = "ini-tial string";` `public abstract string AbstractGetSet{ get; set; }` `//derived class` `public override string AbstractGetSet{ get{ return abstractGetSet; } set{ abstractGetSet = value; } }`
VS	Information loss – missing default value for an attribute, and accessor method is marked as ***virtual*** not ***abstract***, thus it didn`t appear in derived class	`//abstract class` `protected virtual string abstractGetSet { get; set; }` `//derived class doesn`t override the method, because it is virtual`
Class contains an abstract method, but it isn`t defined in derived class		
VP	Correct – method is overridden in derived class	`public override void PublicAbstract() { throw new System.Exception("Not im-plemented"); }`
EA	Compilation error – method is missing in derived class	`//method in not overridden in derived class`
VS	Compilation error – method is missing in derived class	`//method in not overridden in derived class`
Class contains an abstract method with *public* access, but in derived class access is changed to *protected*		
VP	Compilation error – method access is not converted into ***public***	`protected static override void PublicStaticAbstract()`
EA	Compilation error – method access is not converted into ***public***	`protected static override void publicStaticAbstract()`
VS	Compilation error – method access is not converted into ***public***; miss-ing ***override*** in derived class method	`protected static void publicStaticAbstract()`

3.2 Summarization of Results of Experiments with Code Generation in the Investigated Tools

Overall, the experimental models contain more than 60 tests. The transformation results (generated construction) were classified into the following categories:

- Compilation error – generated construction doesn`t compile;
- Execution error – construction contain potential error or unexpected behaviour;

- Information loss – generated code doesn`t correspond to the model semantics;
- Missing notation – unable to express desired semantic in the model;
- Different – alternative to expected result, it compiles and doesn`t contain error;
- Correct – construction match to the expected code.

The authors have been making experiments for the following programming structures/UML notation: access modifiers (***private***, ***protected***, ***internal***); class modifiers (***abstract***, ***sealed***, ***static***); method modifiers (***static***, ***new***, ***override***, ***abstract***); method parameter modifiers (***params***, ***ref***, ***out***); accessor methods; multiplicity; default values; read-only and derived values; constructors; destructors; stereotypes: constant, event, property; active class; constraints: ordered, unique, redefines; naming conventions and keyword usage; namespace scope; tag-values: precondition, postcondition, etc.

In many tests the investigated tools show different results. And it is not like one tool always demonstrates better result than other. There are tests, where it shows correct result, but other fails and vice versa. The analysis of the transformation result shows that the quality of the generated source code is very low (in our experiments). The overall statistics is shown in Fig. 1.

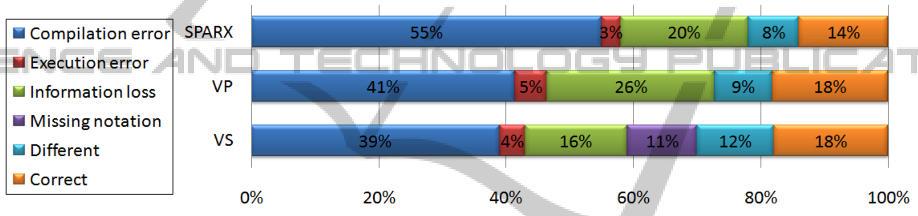


Fig. 1. Transformation result statistics grouped by categories for each tool.

The experimental models contain 69 tests in regard to class element, attributes and methods – the core element in UML class diagram. The missing notation category was added (to the list of categories) because of the Visual Studio, originally the experiments were done with SPARX EA [10] and Visual Paradigm [11], and only later tested with Visual Studio. Because it was not possible to provide tagged values and constraints, these tests resulted into missing notation. Finally only 14%/18%/18% of the tests are correct. Taking into account, that these tests didn`t include all possible elements of the UML class diagram, like, various relationship elements, the obtained results can be still considered as one of the main reasons why the class diagrams and code generation from models are not widely applied in the IT industry.

4 Conclusions

Currently, implementation forces of the main MDA statements are turned to increasing of the level of abstraction, namely, the formalization of system modelling process so as it will allow to build the system model as much as possible correct and consistent with further generating the program code from it. For this research the authors chose the lower end of the MDA implementation chain, which is an analysis of the code generation, as the main hypothesis of this research is that the failures in code generation is the main stumbling block.

The reason for such results which showed an analysis of three code generation tools is the primitivism of the transformations. In most cases, the code generator is limited to the usage of a simple notation keyword transfer into the program code, without additional analysis of the construction correctness and making any adjustments. The information loss and incompleteness of the generated source code is evidence about lack of knowledge of UML semantics and programming language rules in the transformation algorithms. Although it is possible to change the transformation templates, it doesn`t solve the problem completely, because of limitation to local scope. For example, it doesn`t solve the problem for transformations where the result depends on the context, i.e. another structure. Thus, the templates can solve local problems which can be written in „if-then" form. Still the problematic aspects of the template-based solution are as follows:

- Template workflow is based on: „if-then" blocks, which execute another template, and variables, which can store some temporal value. Despite to make some convertations, the built-in functions should be used, thus the functionality is limited;
- Templates are working in a „local scope" mode and because of each part of the class is processed by a separate template, i.e. class, attribute, operation has its own template, and thus its own scope (e.g. in SPARX Enterprise Architect), it is difficult to pass some values between templates or access other structures in the context.

Problems that have to be solved in transformation process from platform-specific model to programming language source code are: meaning of notation and its semantics, meaning of keywords and their usage in context, information control and adjustments as well as support for programming language libraries from different versions of language. The authors assume that the evolution of modelling tools and transformation frameworks will turn forward the evolution of the dictionary for MDA support for each target platform programming language in order to provide additional structural and functional standards of abstraction and templates. The research results highlight the current problems of model transformation, so that it can serve as a recommendation set to upgrade modern code generators. While in the future to solve the task of complete generation of system functionality we have to think about fundamentally different architecture of code generators and code generation process itself.

Acknowledgements

The research reflected in the paper is supported by Grant of Latvian Council of Science No. 09.1245 "Methods, models and tools for developing and governance of agile information systems" and by ERAF project "Evolution of RTU international collaboration, projects and capacity in science and technologies".

References

1. UML Unified Modelling Language Specification, OMG document http://www.omg.org.
2. MDA Model Driven Architecture, OMG web-site http://www.mda.omg.org.
3. Nikiforova, O., Sejans, J., Cernickins, A.: Role of UML Class Diagram in Object-Oriented Software Development, The Scientific Journal of Riga Technical University, Series Computer Science – Applied Computer Systems (2011) (in press).
4. Loniewski, G., Insfran, E., Abrahao, S.: A Systematic Review of the Use of Requirements Techniques in Model-Driven Development, D.C. Petriu, N. Rouguette, O. Haugen (Eds.) the Proceedings of the 13th Conference, MODELS 2010, Model Driven Engineering Languages and Systems, Part II, Oslo, Norway (2010) pp. 213-227.
5. Eichelberger, H., Schmid, K., Eldogan, Y.: A comprehensive analysis of UML tools, their capabilities and their compliance, Technical report, University of Hildesheim (2008).
6. Niaz, I. A.: Automatic Code Generation From UML Class and Statechart Diagrams. PhD Thesis (2005) University of Tsukuba, Japan, p. 104.
7. Usman, M., Nadeem, A.: Automatic Generation of Java Code from UML diagrams using UJECTOR (2009) International Journal of Software Engineering and its Applications, Vol.3, No.2, April, 2009.
8. Herrington, J.: Code Generation in Action. Manning (2003), p. 342.
9. Cernickins A., Nikiforova O., Ozols K., Sejans J. An Outline of Conceptual Framework for Certification of MDA Tools, Proceedings of the 2nd International Workshop „Model Driven Architecture and Modelling Theory Driven Development" (MDA&MTDD 2010), Osis J., Nikiforova O. (Eds.), Greece, Athens, SciTePress, Portugal (2010) pp. 60-69.
10. Sejans, J., Nikiforova, O.: Problems and perspectives of code generation from UML class diagram, The Scientific Journal of Riga Technical University, Series Computer Science – Applied Computer Systems (2011) (in press).
11. Sejans, J.: Analysis of Transformation Result from UML Class Diagram, Master thesis, Riga Technical University (2010).