Two-Dimensional Models' Processing Using Principles of Knowledge-Based Architecture

Andrejs Bajovs, Oksana Nikiforova
Faculty of Computer Science and Information Technology
Riga Technical University
Riga, Latvia
e-mail: andrejs.bajovs@rtu.lv, oksana.nikiforova@rtu.lv

***Abstract* — Presently, the technological diversity increases the attention to Model Driven Software Development, which provides system modeling at the high level of abstraction and further generation of software components. In this aspect, the task of the automatic code generation starts to play an important role and requires a new generation of the research directed to the quality of model and model transformation result. This paper discusses an ability to use several principles of artificial intelligence and knowledge management and offers so called knowledge-based architecture for code generation from the Unified Modeling Language class diagram and a verification of a class diagram itself.**

Keywords- UML class diagram; code generation; knowledge base; model verification

I. INTRODUCTION

An increasing impact of the role of system modeling during software development facilitates the leading positions of Object Management Group (OMG) [1] and its solution for system abstraction, modeling, development, and reuse – Model Driven Architecture (MDA) [2]. A key component of usages of MDA is Unified Modeling Language (UML) [3], which defines several kinds of diagrams, their elements and notation. UML diagrams describe the system from different aspects: static diagram represents system structure, dynamic diagrams represents system behavior. Fully automated transformation of system model, defined at platform independent level into platform-specific source code, is the main goal of MDA.

Currently, this goal has not yet been achieved completely, due to problems with definition of system dynamic aspects and their translation into code components [2]. But even description of system static elements would give a good initial preparation for system development and its further refinement with dynamic aspects. This static system representation in the form of UML class diagram and further generation of software components could replace significant amount of routine work performed by programmers during software development. Reducing its amount could give developers an opportunity to focus on more important tasks, thus helping to improve the quality of computer systems' developing process.

Model-Driven Architecture defines that the system's models could be automatically transformed from one level of abstraction into another. These levels involve not only graphical, but also textual models, including a source code. So, according to MDA, a graphical model could be automatically transformed into a source code. Such transformation process is commonly called code generation.

The idea of automatic code generation is not new. The first code generators were compilers which appeared in the middle seventies and used text-to-text generation techniques [4]. Since then, a significant amount of different standards appeared to support the idea of automatic code generation, however the practical side of this field was left almost untouched. Nowadays, a significant amount of different tools exists, which implement the most popular code generation approach – text templates. However, the authors' previous study shows that the code generation as a result of the UML class diagram transformation is of a low quality [5]. As designed for the concrete situations (thus, required to be frequently rewritten), templates, possibly, limit the functionality of some popular code generators. The other problem is that the code generators do not "think" like a human while doing their job and should be endowed with means of at least artificial intelligence.

Therefore, authors state code generation as an object to research and propose knowledge-based code generator architecture, which allows not only generating the source code, but also verifies the correctness of a model and thus a model transformation result.

The goal of the paper is to describe how the basic principles of artificial intelligence could be used to increase the quality of the code generation process. This paper specifies the background of the term "code generation" and reveals the related problems. In order to solve them, the hypothesis of the knowledge-based code generator architecture is described. In addition, the small practical example is presented to reveal the essence of the proposed theory.

The paper is structured as follows. The second section describes the roots of code generation and related problems which disturb its evolution. Section three introduces the knowledge-based code generator architecture and describes its parts, advantages, and disadvantages. The mechanism of how the introduced architecture works is explained in section four. The fifth section gives an overview of the researches related to the code generators, which use artificial intelligence. Section six concludes the paper.

 ISBN: 978-1-61208-304-9

II. CODE GENERATION: STATE OF THE ART

The term code generation has several interpretations. One of them is defined by OMG's MDA. It states that implementation of a concrete target platform is generated from a model containing the target platform's specific details using pre-defined and tool supported transformations. Actually, OMG did not invent anything new, but standardized older framework – Model Driven Software Development (MDSD) [6]. Both of MDA and MDSD are related to a term "model", which according to [1] is *"... a description or specification of a system and its environment for some certain purpose.*" However, MDA considers models to be central in the development process (assuming that the model represents a set of diagrams that express the whole software system) [7]. According to MDA, these diagrams are used to build the systems for any platform, however MDSD does not claim such portability at all. In contrast of MDSD, MDA suggests using only UML diagrams to describe the system at a high level of abstraction. In general, MDA is more strict than MDSD, which allows much more ways of building the computer system by using models [6].

There are four basic models for systems' development proposed by MDA: computation independent, platform independent, platform specific and implementation specific model. The first one reflects to business and its models. The next two represent analysis and detailed design models of software system to be developed. The last one reflects to implementation and runtime models and, in fact, it is a system's source code. MDA also defines that each of the described models could be transformed into the others [8]. This paper focuses on the automatic transformation of platform specific model to the implementation specific model.

While the OMG organization was developing theoretical basis of the research area, practical side of code generation started to fall behind. Nowadays, a significant amount of different standards related to code generation exists [9], but no methods could completely describe how to apply all these theory into practice. The problem is that OMG invented their standards for templates and transformation languages, but almost forgot about looking at the core process of code generation itself.

Speaking about theory, the computer science describes two different code generation approaches [10], but both of them involve word mapping to model elements. In addition, the study from [5] shows that some of the nowadays most popular code generators are not producing a good quality code because of lack of smart ways to verify correctness of the models.

Authors are making experiments with different software development environments and different tools, positioned as MDA/MDD support tools [5], and have detected several inadequacies between expected code and code generated by the tool. Unfortunately, the current experiments with modeling tools that generate program code from UML class diagram show a weak and unsatisfactory results compared to the expected. Authors have identified a number of problems, which can be generally divided into two groups:

- Modeling tools allow to create improper element constructions and use incompatible keyword connections that leads model transformation into incorrect code, that can`t be compiled.
- Generated code does not correspond to notation and details used in model, which leads to loss of information in the result code.

The root problem is in the simplicity of program code generators, which just transfer the pattern of model information into the program code without any additional testing and decision making on the required information conversion for the target programming language. Generators do not have any additional knowledge support about target platform restrictions, laws and keyword combination. Some tools like SPARX Enterprise Architect [11] have code template editor with built-in transformation templates, which can be modified to support custom needs, but this does not solve the problem of the lack of base information about target platform, because restrictions might be needed for combination of elements and not one-to-one element mapping. The second mentioned group points to the complexity of the generators negligence. The result program code does not represent appropriate constructions for semantics used in the model, resulting in loss of information and devalue of the work invested to provide additional details in the model.

It means that it is not enough with simple word mapping, and machine should be taught to apply some knowledge performing code generation. Inspired by this idea, in the next sections authors propose their hypothesis of applying some principles of artificial intelligence in code generation process to supplement it with the model verification.

III. THE KNOWLEDGE-BASED CODE GENERATOR ARCHITECTURE

In this section, authors propose the hypothesis of the knowledge-based code generator architecture, which is shown in Fig. 1.

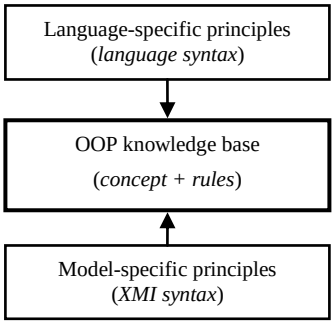

Figure 1. Knowledge-based code generator architecture.

The reason authors call it "knowledge-based" is as follows. As it was mentioned before, code generation is nothing but model transformation to code performed by computer. But how do human beings act, while transforming models to source code? It could differ from concrete

individual, but commonly, each element of the model is taken and transformed, in a step-by-step manner, into code according to some knowledge of the model's notation, programming language syntax and fundamental rules of object-oriented paradigm. In authors' opinion, the word "*knowledge*" is the keyword here. That is the reason why the proposed architecture consists of three blocks: Object-Oriented Programming (OOP) knowledge base, model-specific principles and language-specific principles. All of them are explained in the next subsections of the paper.

A. *Description of the knowledge-based architecture's blocks*

The main block of the proposed architecture is OOP knowledge base, which describes the field of object oriented programming in a high level of abstraction. It represents only the very basics and does not describe anything connected with the concrete programming languages, models or platform-specific things. This is expressed in a way of ontology [12], which keeps two main things: conceptual information about OOP and basic rules to support validating the correctness of the UML class model.

The first is represented as a tree structure, which shows the relationships between different concepts of OOP (e.g., class, visibility, attribute, method, etc.). The simple example of such structure is shown in Fig. 2. Due to the complexity of the OOP itself, the relations between some concepts (visibility and attribute/method, type and name, interface and method, etc.) are omitted at the example to make it more readable and simpler to understand.

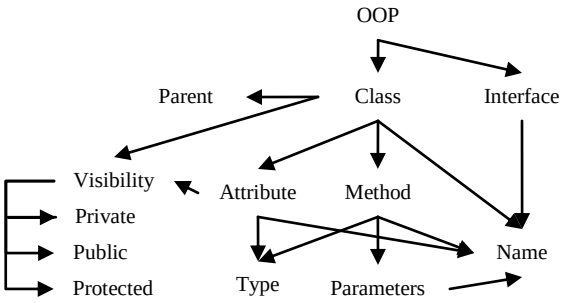

Figure 2. Example of OOP structure.

The second part of the OOP knowledge base is an alternative to Object Constraint Language (OCL). This block is represented by the set of rules, where each rule is a first-order logic (predicate) expression. This set of rules describes some restrictions which exist in the context of OOP (e.g., attribute can be only one at a time: private or public or protected).

The research of [5] defines some of the rules which are most commonly missed by code generators. They are:

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;
3. Because an abstract method must be overridden in the derived class, then it must not be private;
4. While overriding an abstract method, the access modifier ought to be the 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.

The rules mentioned above could be formally expressed in the way shown in Fig. 3.

1. has(Class, Method) & abstract(Method) → abstract(Class);
2. ¬abstract(Class1) & parent(Class1, Class2) & abstract(Class2) & inherited(Method, Class1, Class2) → overriden(Method, Class1, Class2);
3. abstract(Method) & overriden(Method) → ¬private(Method);
4. overriden(Method1, Method2) & abstract(Method2) → equals(visibility(Method1), visibility(Method2).

Figure 3. Formal expression of the model validation rules

The other block of the knowledge-based architecture is a set of language-specific principles or in other words, the syntax of different programming languages. In fact, there are several sets of such rules – each represents concrete programming language. The description of the syntax should be similar to Backus-Naur Form (BNF) notation [13] because its level of formalization allows to be easily interpreted by computer. The syntax of languages should be described using templates which associate concepts from the OOP knowledge base with its formal syntax. Although templates have some major disadvantages [5] which force to find alternatives to replace them, it is preferable to use them here. However, in this context templates should be maximally laconic and structured, describing the whole syntax of a concrete programming language rather than a particular case. The example of a simplified description of a Java class is shown in Fig. 4.

```
"class" <Name> [<Parent>]
["implements"  <Interface>
{","<Interface>}] "{"
          [{<Attribute>}]
"}"
```

Figure 4. Example of the class syntax description using BNF notation

Such markups as <*Name*> or <*Parent*> are taken from the OOP concept (see Fig. 2). During code generation the <*Name*> is replaced by the name of a particular class while <*Attribute*> is replaced by another piece of code which in case of Java is defined like this:

[<*Visibility*>] [<Scope>] <*Type*> <*Name*> [= <*Value*>];

As it was stated earlier, the BNF notation is used to specify the syntax. Thus, blocks which are enclosed inside "*{}*" are repeating blocks, but blocks inside "*[]*" are those which can not be in the code for it to be correct, etc. A word inside "<>" points to a concrete block of the syntax which is associated with the concrete OOP concept. The last is a modification which is used for proposed architecture and is not connected with BNF.

 ISBN: 978-1-61208-304-9

The third block includes the model-specific principles that, in fact, represent the mapping of the concepts from the OOP knowledge base to the Extensible Markup Language Metadata Interchange (XMI) representation of the model [14]. This should be done using slightly extended XPath language [15]. Since various modeling tools implement XMI format differently [16] this block contains various sets of described mappings which are specific to the XMI format of the concrete tool. For example, let us assume the concepts shown in Fig. 2, which are mapped to the Extensible Markup Language (XML) document shown at Fig. 5.

```
<model>
   <class id = "1" name = "A" p_id = "2">
     <attributes>
       <attribute visibility = "private"
type = "int">
           <name>A_atr1</name>
       </attribute>
     </attributes>
   </class>
   <class id = "2" name = "B"></class>
</model>
```

Figure 5. Example of the class syntax description using BNF notation

In this case, a mapping of the Class concept could be done as *//class*, which means that this concept takes its data from the *class* XML element. The name of the class in turn could be accessed as *<element>/@name* . *<element>* is a reserved directive which points to the element being iterated before a current one since all concepts make a hierarchical structure. This means that the knowledge-based code generator takes a full path *//class/@name* to access the name of the class.

It is also important that language-specific and model-specific principles' blocks could include overloading of some of the classic OOP rules from the OOP knowledge base according to the concrete programming language or model. Section IV shows how all of these blocks work together.

B. Analysis of the knowledge-based architecture

The proposed architecture does not have an ability to autonomously derive code as logical consequence of the knowledge-base like advanced AI code generators do. Basically, the approach does the standard template-based model-to-code transformation where additional intelligence is reflected into using such fundamental AI structures as ontology and first-order logic rules. Thus, ontology, syntax description and rules proposed by the authors could be represented as the equivalent of MDA meta-model, OCL and the templates, but their specter of appliance is wider, as well as they are more universal. For example, OCL is designed directly for UML and is much more oriented on constraining values rather than the structure of the models. In contrast, predicate rules do not depend on any concrete syntax so they could constrain every model by working directly with the essence of OOP itself. As for the proposed templates, they have less complex structure and focus on describing language's syntax rather than simple XMI mapping.

The main advantage of the proposed code generator architecture is its precise structure. Knowledge-based architecture defines the exact set of tasks for each of its blocks. It also specifies different levels of abstraction for describing contents for its blocks. The architecture gives an opportunity to split block creation tasks between different independent specialists where each of them should work on concrete task at a specific level of abstraction. Moreover, the OOP principles are a kind of bridge between a model and a programming language. This means that theoretically, each of the templates can be used with each of the model-specific principles. Rewriting or adding new ones also do not affect the opposite part. In addition, OOP knowledge base is the bridge which stands between the problem and solution domain. This is reflected in Fig. 6.

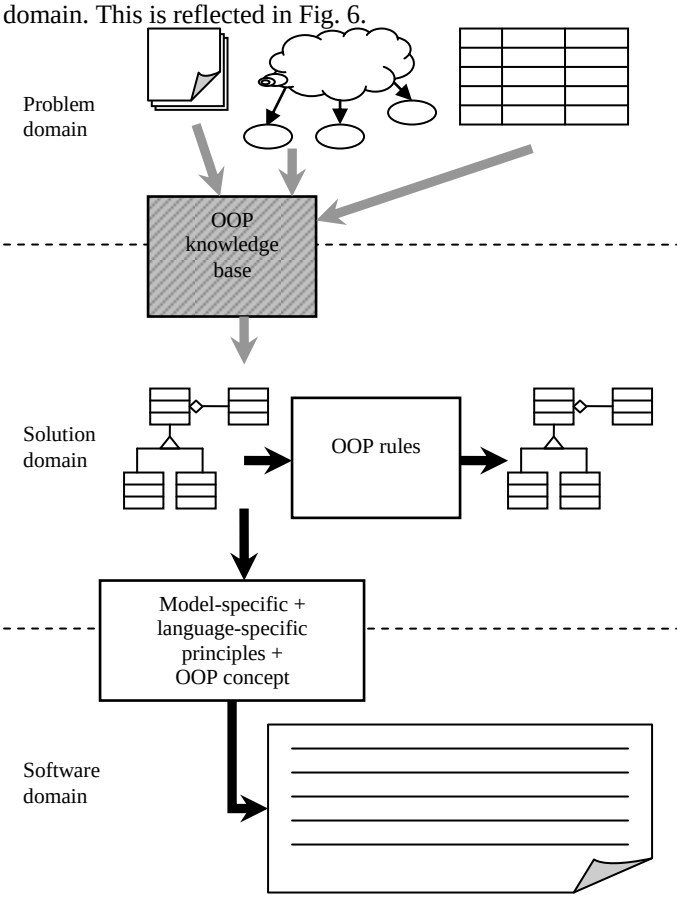

Figure 6. Relation of the knowledge-based code generator with sotware development domains

Theoretically, the OOP knowledge base can be used to transform some artifacts from the Problem software development domain into the model. However, such transformation is out of the knowledge-based generator's scope and thus, it will not be described in this paper.

The main disadvantage of the knowledge-based architecture is a significant amount of the work required to build a knowledge base and map its concepts with the syntax and XMI. However, after this job is done, the knowledge-based code generator potentially can be more powerful. The other disadvantage is that there is a significant amount of

 ISBN: 978-1-61208-304-9

different ways to organize knowledge base as well as some variants to write a syntax templates, which means that functionality of such code generator can strongly vary depending on specialists and many other factors.

In addition, the proposed architecture can be used in two different dimensions: vertical (to generate a code from a model) and horizontal (to verify if model is correct). Normally, generating the code, both dimensions should be involved, but their separate usage is also possible depending on the task. When the model is only verified, the code generator uses mostly the rules from OOP knowledge base, but while performing only the code generation, all other parts of the proposed architecture are used. Fig. 6 shows how these dimensions are related with the software development domain. The reason of calling these two concerns as dimensions is also reflected there. Models are at the same level of abstraction – solution domain, so, while validating them, the code generator is staying within its bounds. That is why the dimension is horizontal. As for the vertical dimension, code generation transfers the model between the different states of the various domains – vertically.

IV. USAGE EXAMPLES OF THE KNOWLEDGE-BASED ARCHITECTURE

As it was mentioned before the architecture of the knowledge-based code generator can be used in two different dimensions: horizontal (to verify the correctness of the model) and vertical (to perform the code generation). The subsections below show the examples of both dimensions.

A. Vertical Dimension (Code Generation)

The knowledge-based code generator works with the OOP knowledge base in the first place. It iterates through the defined concepts starting from the root of the structural tree by jumping between elements according to the relations of these concepts. First, code generator takes an appropriate mapping from the model-specific principles and tries to find a value according to this mapping inside the XML meta-data. If the value is found, then, the code generator takes a syntax template for the OOP concept currently being iterated and produces an output. If the template interpreter finds any markup (text enclosed in "<>") then, it refers to the appropriate concept from the OOP knowledge base, searches for the values according markup from the model-specific principles and finds another template of the text to produce. When the code generator meets a structure enclosed in "*{ }*" it assumes that the model could contain none or more than one element that is represented by the markup inside. Therefore, it takes each of them, repeating the text and iterating through every other concepts enclosed in figure brackets as much as model elements it had found. If the code generator meets something inside "*[]*" then it produces an appropriate text if it finds any values inside the XML documents, otherwise it does not. If the code generator does not find any model elements which are enclosed in "*{}*" or "*[]*" brackets, it will not produce any text inside of them.

Concerning the example shown in Fig. 2, Fig. 4 and Fig. 5, the root is "*OOP*" and its children are "*Class*" and "*Interface*". The code generator will not find anything connected with "*Interface*" because XML document does not contain anything about it. But since a markup of interface is included inside the square as well as figure brackets, the code generator will not insert anything at the place of markup "*<Interface>*", as well as it will not produce a text "*implements*" and ",". The situation with the concept "*Class*" is different. Let us assume that this concept has a markup "*//model/class*". The code generator will use it to state that the XML document contains two elements expressed with this path so it will iterate through them. First of all, the code generator will produce the text "*class* " and meet the markup "*<Name>*". The knowledge base describes the concept with the same name, so the code generator will jump to a model-specific principle and find a markup for this concept. Let us say it is "*<element>/@name*". As the parent concept of the current one is Class, the full path to determine its name is "*//model/class/@name*". Using this, the code generator finds out the name of the class and produces the following code *"A {"*. After that it will return to the parent (which is the the concept Class) and continue parsing the template. The next stop will be a markup "*<Attribute>*". Here, in the same way, the code generator will take a visibility, type and the name of the attribute and construct a piece of code "*private int A_atr1;"*. Since no more information about the class A is provided the code generator will iterate further producing a text "class *B {}* ".

At first glance this mechanism is very similar to the ordinary templates, but the difference is that template is fully separated from the markup. A markup for the Class could possibly be "*//diagram/elements/class*" but for its name –

"*//diagram/attributes[@id = <element>/@id]/name*". This never affects the template and vice versa because these two blocks are connected through the knowledge base which is static. That gives an opportunity to switch between markups easily without making any changes inside the templates.

B. Horizontal Dimension (Model Verification)

The rules which are used to validate the model are described in Fig. 3. The mechanism of the model verification is conceptually simple: the model's every element is tested on matching the defined rules and if at least one of them does not match, the model is considered incorrect. Despite appearing primitive in theory, this part of the proposed architecture is both the most creative and complex because the rules can be translated into logical expressions in a variety of ways. Each rule contains standard symbols defined by predicate logic [12] (terms, predicates, and, or, not, etc.), as well as references to the concepts from the OOP knowledge base expressed as variables. But in contrast to the model-specific and language-specific principles not every OOP concept must be described in the rules. The other part which is skipped in this example is putting some sense in predicates or, in other words, explaining to a computer what does they mean. The programming language, such as Prolog [17] is used to accomplish this. Although it does not fully feat in the concept of the knowledge-based architecture as well as in the code generation itself, it is specially created to work with logical expressions.

 ISBN: 978-1-61208-304-9

V. RELATED WORK

The first code generators in the World were related to text-to-text transformation. They were nothing but high-level compilers. According to [4], the first scientist who started to talk about the code generation was Wilcox. In 1971, he described his compiler, which was based on two internal forms: Abstract Program Tree and Source Language Machine. The first one was translated into the second one, which in turn was transformed into the machine code.

The first popular code generator which was able to transform model into a source code was Rational Rose [18] developed by Rational Software in 1997. Later, this company was consumed by IBM resulting with evolution of Rational Rose into Rational Software Architect [19]. The tool's integration into an Eclipse environment allows users to customize their transformations more flexibly. Flexibility is a distinctive feature of Eclipse, so some other tools operating under this platform exist: Acceleo [20] and XPand [21]. The other popular tools – the "monsters" of today's industry which provide a code generation opportunities are such tools as SPARX Enterprise Architect [11] and Microsoft Visual Studio [22]. This list could be populated with a significant amount of other smaller tools, and basically, all of these code generators use their own different transformation mechanisms which are mostly based on templates. In addition, none of these tools are positioned to use artificial intelligence to perform code generation.

The template based programming originated in the 1960s and became especially popular thirty years later [23]. Eighteen years later, in 2008, the template-based code generation approach was also standardized by OMG [24]. However, since then, no new versions of this specification appeared.

The idea of using artificial intelligence in the field of the code generation was expressed by bloggers-enthusiasts as well as by scientists. Danilchenko and Fox [23] describe their system called the Automated Coder using Artificial Intelligence (ACAI), which as they claim is "… *a first pass at a purely automated code generation system*". ACAI generates the code through some simple steps: first, it generates a plan(s) to solve the problem; next, it takes reusable code components from the library and weaves them according to a created plan. The result is a text template which has been processed to get a working source code. ACAI uses an artificial intelligence technique called Case-Based Reasoning which can be used to maintain a reusable library of code components. Case-Based Reasoning is popular, and also is used in the other code generation systems: CHEF [25], Software Architecture Materialization Explorer [26] and The Individual Code Reuse Tool [27].

The knowledge-based code generator studies, which are mentioned above are advanced and actually they are far from the classic MDA concept. The studies are based on building the program's text from the reusable code components. The knowledge-based architecture, however, describes more simple mechanism which uses only basic AI principles but in fact is much similar to the ideology of the Model-Driven Architecture.

VI. CONCLUSION

Abstraction is the process by which we extract and distill core principles from a set of facts or statements. A model is an abstraction of something in the real world, representing a particular set of properties. There are two primary reasons developers build the model [28]: understanding a process or a thing by identifying and explaining its key characteristics and documenting ideas what developers need to remember and to communicate those ideas to other. OMG's last initiative – Model Driven Architecture offers the third reason on using the models during software development [29]. Using models as a basis for the further code generation and UML class diagram plays the central role on moving an idea about the code generation into the industry.

A significant amount of different standards in the code generation area overwhelmed it and as a result, led to the lack of ways of using them in practice. However, a significant amount of tools exist that have an ability to generate a more or less working source code. In general, all of them are using templates as a code generation technique, and this could be a reason why those code generators have not got an ability to work perfectly yet. The main problem is that templates do not provide any mechanism to verify a model which could be wrong from the start. Thus, as long as completely new approaches of code generation will not be found, the idea of using MDA for making the process of implementing fully functioning system more easy, affordable and reliable will remain nothing but a utopia. For now, templates could not be fully replaced, that is why they must be used in conjunction with the other methods.

The authors of this paper wanted to make a computer "smarter" for the code generation tasks. This could be achieved by applying some principles of the artificial intelligence. Therefore, authors propose a knowledge-based architecture which separates a code generator into three main blocks: model-specific, language-specific principles, and OOP knowledge base. The first one is used to perform meta-model mapping, the second one describes the syntax of a programming language, and the third one keeps the main principles of OOP, as well as it serves as a bridge between the first and the second block. In the opposition to the simple template, the proposed architecture keeps the meta-model mapping independent from templates. It allows not only to use different syntax with different mapping cases but also involving different specialists to work with them independently in turn to save the time.

The key contribution of this paper is extending an ideology of the MDA central components, such as templates, meta-model and constraints. According to the architecture proposed by authors, the templates are no longer overwhelmed by complex directives but contain only references to the OOP knowledge base – the names of OOP concepts. They also represent not only concrete code mapping situations, but a whole syntax of the particular programming language. The templates are independent from the XMI mapping rules because of the OOP knowledge base which is restricted by the first order logic rules that are an alternative to MDA OCL. In contrast of this language, the

 ISBN: 978-1-61208-304-9

predicate rules are also independent from any concrete syntax and XMI, as well as they describe global OOP constraints based on the knowledge base. In addition, the described architecture's components do not only reflect the basic MDA components, but also represent the basic AI structures, which means that they have a potential for future studies of making code generator cleverer.

The code generator, which is based on such an architecture, can be used not only to perform the code generation, but also to verify the model. The both tasks could be performed separately as well as together. The knowledge-based code generator has a potential ability to become powerful, however it is very important to make a good OOP knowledge base.

The further researches will be connected with adding details to each of the three described levels: finding better structures to express them, forming some restrictions and formal rules for this task. When the concept of the knowledge-based architecture is fully ready, the tool should be implemented to realize it practically. This tool could be used to validate the presented approach by systematically applying some tests, which display the most problematic aspects of the model to code transformations, including those which other tools can not handle.

ACKNOWLEDGEMENTS

The research presented in the paper is partly supported by Grant of Latvian Council of Science No. 342/2012 "Development of Models and Methods Based on Distributed Artificial Intelligence, Knowledge Management and Advanced Web Technologies".

REFERENCES

[1] Object Management Group, [Online]. Available: www.omg.org [retrieved: September, 2013]

[2] Model Driven Architecture FAQ, [Online]. Available: http://www.omg.org/mda/faq_mda.htm [retrieved: September, 2013]

[3] UML Unified Modeling Language Specification, OMG document, [Online]. Available: http://www.omg.org/spec/UML/2.4.1 [retrieved: September, 2013]

[4] R. G. G. Cattell, A survey and critique of some models of code generation. Tech. rep. Pittsburgh, Pennsylvania, USA: School of Computer Science, Carnegie Mellon University, 1979.

[5] J. Sejans and O. Nikiforova, "Practical Experiments with Code Generation from the UML Class Diagram," Proceedings of MDA&MDSD 2011, 3rd International Workshop on Model Driven Architecture and Modeling Driven Software Development In conjinction with the 6th International Conference on Evaluation of Novel Approaches to Software Engineering, Osis J., Nikiforova O. (Eds.), Beijing, China, SciTePress, Portugal, Printed in China, Jun. 2011, pp. 57-67

[6] T. Stahl and M. Volter, Model-Driven Software Development, Wiley, 2006, pp. 428.

[7] I. Jacobson, G. Booch, and J. Rumbaugh: The Unified Software Development Process, Addison-Wesley, 2002, pp. 512.

[8] O. Nikiforova, A. Cernickins, and N. Pavlova, "Discussing the Difference between Model-driven Architecture and Model-driven Development in the Context of Supporting Tools," Proceedings of the 4th International Conference on Software Engineering Advances, IEEE Computer Society, Sept. 2009, pp. 446-451.

[9] OMG: Catalog Of OMG Modeling And Metadata Specifications, [Online]. Available: http://www.omg.org/technology/documents/modeling_spec_catalog.htm [retrieved: September, 2013]

[10] A. Bajovs, Research of the Basic Principles of the Model-To-Code Transformation, Bachelor Thesis, Riga Technical University, 2012.

[11] Enterprise Architect – UML Design Tools and UML CASE Tools for Software Development, [Online]. Available: http://www.sparxsystems.com.au/products/ea/index.html [retrieved: September, 2013]

[12] S. J. Russell and P. Norvig, Artificial Intelligence: A Modern Approach, Second Edition, Prentice Hall, 2002, pp. 1132.

[13] M. Marcotty and H. Ledgard, The World of Programming Languages, Springer, 1986, pp. 380.

[14] OMG: MOF 2 XMI Mapping Specification Version 2.4.1, [Online]. Available: http://www.omg.org/spec/XMI/2.4.1 [retrieved: September, 2013].

[15] XML Path Language (XPath) 2.0 (Second Edition), W3C document, [Online]. Available: http://www.w3.org/TR/xpath20 [retrieved: September, 2013]

[16] A. Cernickins, O. Nikiforova, K. Ozols, and J. Sejans. "An Outline of Conceptual Framework for Certification of MDA Tools," Proceedings of the 2nd International Workshop on Model-Driven Architecture and Modeling Theory-Driven Development, In conjunction with ENASE 2010, In Janis Osis, Oksana Nikiforova, (Eds.), Athens, Greece, SciTePress, Jul. 2010, pp. 60-69.

[17] C. S. Mellish and W. F. Clocksin, Programming in Prolog: Using the ISO Standard, Fifth Edition, Springer, 2003, pp. 300.

[18] IBM Software – Rational Rose, [Online]. Available: http://www-01.ibm.com/software/awdtools/developer/rose [retrieved: September, 2013]

[19] Introducing IBM Rational Software Architect, [Online]. Available: http://www.ibm.com/developerworks/rational/library/05/kunal/?S_TACT=105AGX99&S_CMP=CP [retrieved: September, 2013]

[20] Acceleo home page, [Online]. Available: http://www.eclipse.org/acceleo/ [retrieved: September, 2013]

[21] XPand – Eclipsepedia, [Online]. Available: http://wiki.eclipse.org/Xpand [retrieved: September, 2013]

[22] Microsoft Visual Studio 2012, [Online]. Available: http://www.microsoft.com/visualstudio/eng/team-foundation-service [retrieved: September, 2013]

[23] Y. Danilchenko and R. Fox, "Automated Code Generation Using Case-Based Reasoning, Routine Design and Template-Based Programming," in the Proceedings of the 23rd Midwest Artificial Intelligence and Cognitive Science Conference, S. Visa, A. Inoue and A. Ralescu editors, Omnipress, Apr. 2012, pp. 119-125.

[24] MOF Model To Text Transformation Language, Version 1.0, [Online]. Available: http://www.omg.org/spec/MOFM2T/ [retrieved: September, 2013]

[25] K. J. Hammond, "CHEF: A Model of Case-based Planning," in Proceedings of the Fifth National Conference on Artificial Intelligence, AAAI, Aug. 1986, pp. 267-271.

[26] G. Vazquez, J. Pace, and M. Campo, "A Case-based Reasoning Approach for Materializing Software Architectures onto Object-oriented Designs," in Proceeding SAC '08 Proceedings of the 2008 ACM symposium on Applied Computing, ACM, Mar. 2008, pp. 842-843.

[27] M. Hsieh, and E. Tempero, "Supporting Software Reuse by the Individual Programmer," in Proceedings of the 29th Australasian Computer Science Con"ference, Australian Computer Society, Inc, Jan. 2006, pp. 25-33.

[28] J., W. Satzinger, R. B. Jackson, and S. D. Burd: Object-Oriented Analysis and Design with the Unified Process, Thomson Course Technology, 2005, pp. 656.

[29] D. Gasevic, D. Djuric, and V. Devedzic: Model Driven Engineering and Ontology Development, 2nd edition, Springer, 2009, pp. 378.

 ISBN: 978-1-61208-304-9