Published as: A.DereziŒka, K. Redosz, Reuse of project code in model to code transformations, In: *Information Systems Architecture and Technology, Contemporary Approaches to Design and Evolution of Information Systems*, L.Borzemski at al. (Eds.), Oficyna Wydawnicza Politechniki Wroclawskiej, Wroclaw, Poland 2014, pp. 79-88, ISBN 978-83-7493-855-6.

Key words:
UML, code generation, C#,
generation gap

Anna DEREZIő SKA*, Karol REDOSZ*

REUSE OF PROJECT CODE IN MODEL TO CODE TRANSFORMATION

Model to code transformation can be performed many times during a development process based on the Model Driven Engineering ideas. A code project generated from structural as well as behavioral models can be further extended with program details, e.g. method bodies. A straightforward re-generation of an updated model could result in overwriting of the implemented parts of a project. Therefore, it is beneficial to reuse a code originated from the previous project during a model to code transformation realized in a next development stage. Different approaches to code generation gap are discussed in the paper. One of approaches, based on reuse of elements of non-modified classes, is specified in terms of reuse criteria devoted to different programming notions. The proposed criteria were implemented in an extended version of the Framework for Executable UML tool (FXU). The criteria were evaluated in experiments on UML models transformed into C# programs.

1. INTRODUCTION

In a Model Driven Engineering (MDE) approach, a model to code transformation takes a significant role [1]. In typical MDE processes a structural model, described for example by class diagrams, can be transformed into a code project. Moreover, some behavioral specifications built with CASE tools, e.g. state machine diagrams, can be used as source models in the code generation [2-5].

However, a detailed code is usually supplemented at the code project level using a development environment, which is more convenient for programmers. This kind of code refers, for example, to bodies of selected methods.

* Institute of Computer Science, Warsaw University of Technology, Nowowiejska 15/19, 00-665 Warsaw, Poland.

The problem arises when a model is modified and its transformation is repeated. A simple model to code transformation would provide to losing of an implemented code, hence it is not a satisfactory solution [6-8]. It could be beneficial to have full roundtrip engineering, but so far it is available only for some solutions based on structural models, mainly when a class diagram is generated by reverse engineering from an existing code. The reverse engineering for code generated from more extended models, including state machines, is more sophisticated and ambiguous.

In this paper, we address the code generation gap problem [8]. We present different approaches and propose code reuse criteria for a selected solution. It is assumed that a model to code transformation process is mostly automated, without need of a user interaction. The selected approach was implemented in the Framework for Executable UML (FXU) [9,10], which is a tool for generation of C# code from UML class diagrams and state diagrams. Experiments were carried out to verify and tune the criteria [11].

This paper is organized as follows. We present the gap generation problem and different solutions in the next Section. In Section 3 code reuse criteria are introduced. We discuss experiments with FXU in Section 4 and conclude the paper in Section 5.

2. APPROACHES TO CODE REUSE

The following assumptions were made about the considered MDE process. A model M^{-1} was automatically transformed into a code project. The project was further extended with some code extracts, e.g. bodies of methods or constructors were added; and some elements like fields, methods, parameters were added or deleted. The current updated project will be denoted as *P*.

In the next development stage, a new modified version of the model was prepared. It is called *M*. The model *M* will be transformed into a new code project *P***â** The transformation uses also data from the project *P*, merging information from *M* and *P*.

We considered four approaches (A-D) taking into account the following features:

- reuse of the code from the source project *P* in the target project *P***â**,

- sources of potential compilation errors because of reuse of the existing code from *P* together with the new generated code from *M*,

- involvement of a user in the selection of model/code elements to be included into the final project *P***â**

2.1. GENERATION OF THE MODIFIED CLASSES ONLY (A)

A model to code transformation is performed only for classes that are new or were modified in the model *M*. It is necessary to determine which class was modified. It can be established by a user or automatically. Automatic detection of a class modification can be realized in different ways. A simple solution is comparison of the previous model M^{-1} with the modified model *M*, but it can be inconvenient for a user to maintain both models. Another method is usage of information that identifies the previous model and is stored in the code project in the form of additional comments or annotations. A user should only not to interfere into this part of the code.

Advantages of the approach are: a simple implementation and little sources of potential errors. A disadvantage is a fact that a class with a simple change will be classified as modified, and the whole class will be substituted. In an extreme situation, a change of a class name will provide to rewriting of the class files.

2.2. GENERATION GAP PATTERN (B)

The problem is addressed also by a design pattern called *generation gap pattern*. Each class of a model is transformed into two classes related by inheritance. The base class is an abstract class that comprises elements generated from a model. No other elements are included in the class and a user should not modify it. A user works on the second class, inherited from the base one. Typically base and inherited classes are stored in different packets and/or folders with some meaningful names, such as *src-gen* and *src-once* [7, 8].

A positive of the approach is that automatically generated code is explicitly separated, and manually implemented code fragments can be easily reused. The re-generation process is fully automated. A user should only follow the implementation rules and do not modify the base classes.

A drawback is the high number of classes, duplicated in comparison to the model. The generated code is also less comprehensible. Using of a class hierarchy in a model can cause problems of merging inheritance relations from two origins: a conceptual model and the design pattern. It is especially questionable in languages that do not support multiple inheritance, as Java or C#. Repeated generation of base classes can produce errors in the inherited classes, e.g. an inherited class uses an element of the base class which was removed in the modified model.

2.3. REUSE OF NON-MODIFIED CLASS ELEMENTS (C)

Elements that were not modified in a model are reused in the target project. This approach is similar to A), but takes into account different elements not only the whole classes.

There are different ways to specify that an element was modified. An automatically generated code can be separated in a class using comments or annotations. Therefore the file structure of a project is not affected. For example, elements generated by IBM

Rational Software Architect [12] in UML to Java transformation are labelled with *@generated* annotations. Code modifications are substituted by the newly generated code based on the current model. However, those parts of method and constructor bodies that are placed between comments ñbegin-user-codeò and ñend-user-codeò are not substituted.

Moreover this approach requires criteria about reuse of various kinds of elements. One solution is a selection of only simple elements, e.g. a method or constructor body, that have a non-modified signature. However, a code extract to be copied can have range of references to other elements of the same and of different classes. A simple coping without strict rules can provide to errors. On the other hand, too severe restrictions result in the losing of too much of the code that will be not reused.

An advantage is possibility to reuse extracts of the code (methods, constructors) of classes that were partially modified. Using some selection criteria, the transformation process can be realized without a user interaction.

The approach has several disadvantages. Change of a class name in a model provides to a substitution of the whole class and its implementation. Non-modified elements of a class that were copied can depend of other elements that were modified in the model. Therefore, there is a risk of errors due to reused code. In comparison to the A) approach this solution is more complex in terms of reuse criteria and implementation.

2.4. FULL MERGING OF CODE WITH MODEL (D)

In the first three approaches we assume the fully MDE approach, i.e. a model is correct and superior in the code generation process. A modified model is used as a reference in the selection of elements to be generated in the revived project. Therefore, elements that exist in the previous code project but are not included in the new version of the model should be omitted. In result the model should be complete in the reference to the current stage development and include all elements indispensable in the project.

The fourth approach is based on different assumptions. A model and a current code project include complementary parts of the development that can be merged into the final project. A user should know which parts of the previous model can be reused. A merging engine can inquire a user about doubtful elements and automatically handle elements encountering in both a modified model and a previous code project. There are also different policies based on the automating merging rules:

I) *A model is superior.* An analysed element is generated if exists in the current model *M*, regardless weather it has a corresponding element in the project *P*.

II) *A code project is superior.* An element is copied from *P* to the target project *P*â If an element exists only in the model it is omitted during the code generation.

III) *A model supplements a code project*. Each element from the project *P* is reused in *P*â New elements, not present in *P* are generated from the model *M*.

IV) *Model and code project are equally important* and their intersection is a valid development. If an element is used in the model *M* and the project *P* then the element is reused in ***P*â** Otherwise the element is omitted.

An advantage is reusing parts of the code (methods, constructors) of classes that were partially modified. Deciding of code extract reuse is highly flexible. An amount of code that would be lost in the project re-generation can be minimized.

However, this approach has much more complex implementation than the previous ones. In several policies a user is involved into the generation process. There are high requirements on the model and the code comprehension. There are many possible sources of conflicts and errors due to merging of inconsistent code originating from a model and a project.

3. EVALUATION CRITERIA FOR REUSE OF CODE EXTRACS

General policies discussed above have to be companioned by specific criteria that try to avoid situations of the inconsistent code. While extending the Framework for Executable UML (FXU) [10], the approach C) was selected to be implemented. It is a compromise solution that supports an automatic selection of code extracts to be reused. It is not necessary for a user to have a complete knowledge about a previously generated project or modifications that were introduced into a model.

In general a modified model *M* was treated as a superior correct model. It was assumed that if an element existed in the previous model M^{-1} but was omitted in *M* it was deleted by a user on purpose. In the reuse mechanism it was assumed that each system change will be introduced in a model. An exception is a constructor, as in FXU constructors are not generated from a model.

The following proposed criteria can be in the most cases applied to any object-oriented language. However, they were designed in the context of the C# language. The criteria were specified precisely using formulae. As an example, one formula for classes is shown. The remaining are omitted due to brevity reasons.

A packet encounters in the most of programming languages, including C#, represented by a folder structure. A packet is an important structural element. A packet P_M defined in a model is equivalent to a packet P_P generated in project if the following conditions are met:

- both packets have the same names, both packets are not included in any superior packet, or are included in equivalent packets.

An interface includes declarations of methods. The methods should be implemented in a class (or classes) that realize them. Assuming that a model is complete there should be no implementation elements that are not present in the model. Therefore interfaces are generated from models and do not have special criteria for matching with the code.

A class is structurally considered as a configuration of various sub-elements. A class C_M defined in a model is equivalent to a class C_P generated in project if the following conditions are met:

- the classes are comprised in equivalent packets,

$$((C_M \in CLASSES(P_M)) \wedge (C_P \in CLASSES(P_P)) \wedge (P_M \equiv P_P)) \qquad (1)$$

where *CLASSES*(P_X) means a set of classes in the packet *X*, and $\equiv$ denotes equivalence.

- the classes have the same names, and have the same descriptors, e.g. *abstract.*

It should be noted that the criteria do not take into account inheritance or interface realisation. It was assumed that these conditions would be too restrictive.

An attribute can be fully generated from a model. It can have a type and an initial value that is initialised in its definition or in a constructor. A problem erases when attributes are created or deleted in implementation. Such modifications should be reflected in a model. It can be done by hand or by simple reverse engineering facility.

For the further merging step, the following criteria state that an attribute A_M from a model is equivalent to an attribute A_P from a project, if:

- the attributes belong to the equivalent classes, have the same names, are of the same types, have the same visibility, and have the same descriptors, e.g. *static, abstract*.

A constructor is typically created as a default constructor**.** This constructor can be supplemented in implementation, and/or additional constructor can be developed. They can be used in a new project. A constructor is copied from a previous project to a new one, if its class in a model includes all attributes with the same names and types, as attributes defined in the equivalent class in the project.

A method has, apart from its signature, a body that is usually not represented in a model. A method code can be written in a model, but it is often inconvenient as it is not verified on-line. Generating a new code from a model can lead to losing an implemented code. Therefore method code from a project should be reused in the new project after model to code transformation. A method M_P from a project can be merged with a method M_M defined in a model, if the following criteria are met:

- the methods are in the equivalent classes, have the same name, have the same return type, have the same visibility, and have the same descriptors, e.g. *static*.

- the methods have parameters of the same names and types, but the parameter order can be different.

However, there also exist special methods, e.g. methods generated to reflect behavioral models that should be not modified by a user. They are generated from a model disregarding the above criteria.

4. EXPERIMENTS

4.1. FXU ï FRAMEWORK FOR EXECUTABLE UML

The Framework for eXecutable UML (FXU) is an environment for generating C# programs from UML models [10]. The first version of the tool was created in 2006 [9]. It consists of two parts FXU Generator and FXU run-time Library. FXU Generator transforms UML classes and their state machines into a C# project. The project includes skeletons of the classes, the code reflecting structures of the state machine elements and references to library items. The Library comprised the code corresponding to realization of all state machine elements used in run-time of an application. Further versions of FXU were developed, which among others support extended GUI, model tracing interfaces, and different UML semantic variants [13], [14]. The 5th version of the FXU generator was enhanced with the code reuse facility [11].

4.2. EXPERIMENT SETTINGS

In the discussed process, modified models are transformation sources in consecutive development stages. Therefore a simple *modification process benchmark* was proposed. The following basic actions can be distinguished in a process: addition, removal and modification. Each of these actions can be applied to different software elements ï action subjects: a packet, class, class attribute, method and class constructor.

Considering the program generation cycle, there are three possible areas when an action can be completed: in a model only, in a code project only, both in a model and in the code project. Combination of the above variants (i.e. action type, action subject and application area) constitutes a modification process benchmark used in the experiments.

The experiments were evaluated in terms of the usefulness of the implemented approach and the applied criteria. Therefore some metrics were proposed to evaluate the process in a quantitative way. Utilization of the code from the project (*P*) during the generation of the new project (***Pô***) was calculated by metrics:

E - a rate of the code added to project *P* but not reused in the project ***Pô***

and *R*=100-*E* ï a rate of the code added during project *P* implementation and reused in ***Pô***,

$$E = \left(\frac{PLOC - GLOC_{mrg}}{PLOC - GLOC_{all}} \right) * 100 \qquad (2)$$

where *PLOC* ï number of code lines in the Pô

$GLOC_{all}$ – number of code lines generated in P′, when no reuse mechanism was applied,

$GLOC_{mrg}$ – number of code lines generated in P′, when the reuse mechanism was applied.

Lines of code were calculated without lines of comments and white lines.

The experiments were performed on two models, so-called a *training model* and a *test model*. As a training model we used a model of a control system of an orthotic robot [11]. The model was enhanced in order to cover various elements of state machines modelled in UML. This model was used as a benchmark during evaluation of transformations of a UML class and state machine model to code. Transformations carried out by three tools were experimentally compared, namely FXU, IBM Rational Software Architect and IBM Rhapsody [11]. In the experiments referred in this paper the same model was applied for the preliminary verification and tuning of the reuse criteria.

Preliminary version of reuse criteria were implemented in FXU and applied to the training model. After experiments, some criteria concerning reuse of constructor and methods were modified. The criteria described in Section 3 cover already those modifications.

The second model used for the final testing of the discussed code generation approach was based on an open source C# project. It was published in the GitHub service an open source repository [15]. This is a SharpUnit library devoted to creation of unit tests in the C# language. Therefore it was a real commonly used project developed independently of the FXU environment and experiment participants. Its UML model was generated by reverse engineering methods, using facilities included in the IBM Rational Software Architect environment [12].

Both models were modified by different actions according to the modification process benchmark.

4.3. EXPERIMENT RESULTS

In Table 1, we presented how performed modifications influence on the rate of the code reuse. Only such modifications were considered that have an impact on conditions deciding on an element reuse. For example, a method modification in which only an attribute order is changed is not taken into account.

The results of the test model showed that the automatic reuse was applied to almost 90% of the code. This is substantial improvement in the comparison to the pure model re-generation, but still it is not fully satisfactory.

Merging of code is threatened by inconsistency errors in the final project. A source of errors can be deletion or modification of an attribute in a model that was used in a method body in the previous project. The lines of such kind are not counted as reused in the R and E measures. It was observed that R rate was therefore lowered of 0.2%.

Table 1. Reuse rate (*R*) of implemented code during model to code re-transformation

Modification type (action, subject)	Modification area		
	In model	In project	In model and project
No changes	89.3 %	89.3 %	89.3 %
Attribute add	88.8 %	88.6 %	88.8 %
Attribute delete	87.7 %	89.6 %	89.8 %
Attribute modify	87.0 %	86.8 %	89.3 %
Method add	87.1 %	86.4 %	89.5 %
Method delete	89.1 %	88.3 %	89.1 %
Method modify	86.8 %	86.1 %	89.3 %
Constructor add	-	89.6 %	-
Constructor delete	-	89.0 %	-
Constructor modify	-	89.3 %	-

Calculated values of metric *E* present a distribution of code extracts that have to be implemented in a new code project. If a project is generated from a non-modified model the most of the lost code (about 7.4%) was associated with attributes. This effect was caused by C# specific way of defining *getter* and *setter* accessors, which was not considered in the presented reuse approach. The modification of the criteria of attributes would improve this result.

The rest of elements that should be re-implemented referred to: methods 1.92% of code, imports 1.1% and constructors 0.27%. These code losses were caused by few limitations of FXU generator, i.e. some C# elements are not generated, like virtual descriptor and calling of a base constructor in a constructor of an inherited class.

The similar analysis of a lost code (*E*) was performed for different modifications of a model. In all kinds of modifications the highest code loss (about 7%) was for attributes, because of the above mentioned reasons.

The code loss of method bodies and constructors depends on the modification type. It can be caused by an attribute modification, attribute deletion or method modification. Modifying or deleting an attribute affects a part of a constructor where the attribute is initialized. A method modification results in overwriting its body. The problem can be omitted if modification is introduced in the model and in the project.

5. CONCLUSION

An approach to code reuse in a model to code transformation was proposed and experimentally evaluated. First experiments were applied to the tuning of criteria of programing element reuse. Further experiments, performed on an independently developed

UML model, shown that about 90% of the implemented code were reused and merged with the code automatically generated from a modified model.

However, the approach needs further improvements. We have observed that accessors created for attributes, known as getters and setters, were not automatically reused. This problem can be solved by extending given reuse criteria. Moreover, further experiments should refer to a more complex modification process. A model can include many various modifications, which should be verified during model to code transformation.

REFERENCES

[1] FRANCE R., RUMPE B., *Model-driven Development of Complex Software: A Research Roadmap*, In: Proc. of Future of Software Engineering at ICSE 2007, pp. 37-54. IEEE Soc., 2007.

[2] DOMINGUEZ E.; PEREZ B; RUBIO A. L.; ZAPATA M. A., *A systematic Review of Code Generation Proposals from State Machine Specifications*, Information & Software Technology, vol. 54, no. 10, 2012, 1045 ï 1066.

[3] EDWARDS G.; BRUN Y.; MEDVIDOVIC N.: *Automated Analysis and Code Generation for Domain-Specific Models,* In: Joint Working Conference on Software Architecture & 6th European Conference on Software Architecture, 2012, pp. 161-170.

[4] DANG F.; GOGOLLA M.: *Precise Model-Driven Transformations Based on Graphs and Metamodels, sefm,* 2009 Seventh IEEE International Conference on Software Engineering and Formal Methods, 2009, pp.307-316.

[5] NIAZ I. A.; TANAKA J.: *An Object-Oriented Approach To Generate Java Code From UML Statecharts*, International Journal of Computer & Information Science, Vol. 6, No. 2, June 2005, 83-98.

[6] FOWLER M., *Domain-Specific Languages*, Addision-Weslay Prof., Boston, 2010.

[7] VLISSIDES J., Pattern hatching: design patterns applied, Addison-Wesley, 1998

[8] BEHRENS H., *Generation gap pattern*, 23 April 2009, [online] http://heikobehrens.net/2009/04/23/generation-gap-pattern/

[9] PILITOWSKI R., DEREZIő SKA A., *Code Generation and Execution Framework for UML 2.0 Classes and State Machines*, In: T. Sobh (Ed.) Innovations and Advanced Techniques in Computer and Information Sciences and Engineering, Springer, 2007, 421-427.

[10]FXU Framework for eXecutable UML V5.0 Feb 2014, http://galera.ii.pw.edu.pl/~adr/FXU/

[11]REDOSZ K., *Automatic code generation from UML models - development of the FXU tool*, Master Thesis, Institute of Computer Science, Warsaw Univ. of Technology, 2014 (in polish).

[12]IBM Rational Software Architect, http://www-03.ibm.com/software/products/en/ratisoftarch [access May 2014]

[13] DEREZIő SKA A., SZCZYKULSKI M.: *Tracing of State Machine Execution in Model-driven Development Framework*, In A. Konczakowska, M. Hasse (Eds.), IEEE Explore Proc. of the 2nd Int. Conf. on Information Technology, ICIT'2010, 2010, 109-112.

[14]DEREZIő SKA A., SZCZYKULSKI M.: *Interpretation Problems in Code Generation from UML State Machines - a Comparative Study*, In: T. Kwater (ed.) Computing in Science and Technology 2011: Monographs in Applied Informatics, Dep. of Applied Informatics Faculty of Applied Informatics and Mathematics Warsaw Univ. of Life Sciences, 2012, 36-50.

[15]GitHub , https:github.com/mgants4/SharpUnit [downloaded 2013]