ANALYTICAL FOUNDATION OF MODEL TO CODE TRANSFORMATION ACTIVITIES

Anton Shyrokykh

Abstract: *The Codegeneration problem is not new in AGILE approach. Many wide spread used tools and formal approaches and papers are devoted to this problem.*

From the other hand codegenetation tools, that are used in development practices, have some drawbacks that not allow to transform structure of class diagram to code without mistakes (Shyrokykh, 2020).

It is explained by peculiarities of human cognitive comprehension. When a developer "reads" structure of software represented in graphical notation of UML class diagrams some details are convenient for visual representation, for example interrelations between classes. Proposed transformation rules of model to code transformation languages must consider structure of class diagrams elements in more detailed way.

Paper is devoted to designing of a new codegeneration approach based on idea of preliminary refinement of class diagrams before model to code transformation and further transformation using newly proposed transformation rules. Formal foundation of approach is grounded on model to model transformation language. Aim of this approach is to design intermedia analytical representation of class diagram using algebra describing static software models (Chebanyuk. 2013).

Keywords*: Codeneration, Class Diagram, Transformation Rules, Model to Model Transformation.*

Introduction

Model to code transformation operation is one of the activities reducing development efforts. Software models, represented as UML diagrams, easily comprehend by specialists in comparison with skeletons of code, represented as texts.

Challenges to use modeling notations in real software development companies are the next:

- it must be flexible to represent future software system from different points of view with different levels of details.
- it must be supported by variety of application life cycle management tools

– it must be grounded on professional standard to be easily implement of different specialists

There is no modeling notation that fully answers to the challenges listed above. From the other point of view UML is more close standard to formulated challenges.

Implementing chain of operations supported sequence of transformations from high-level behavioral software models, (represented as UML Use Case or Communication diagrams) to the code allow reduce development efforts to design a chain of software development artifacts that correspond to requirement specification (architectural solutions, code modules, test cases, etc.).

Architectural solutions, represented as UML Components or Class Diagrams, are initial sources of codegeneration procedures. Today in the market different codegeneration tools are represented as separate software tools and as plug-ins to IDE (Shyrokykh A, 2020).

Many codegeneration tools have drawbacks that are sources of loosing some parts of UML class diagrams' or incorrect transformation of structure when model to code transformation is performed (Shyrokykh A, 2020). Different tools needs different efforts after generating skeleton of code. That why correctness of final code structure depends upon qualification of designer (developer) and his efforts to avoid refine mistakes.

Review of papers

Problems of codegenaration approach in different types of software development considered in different papers.

One of the fundamental papers in codegeneration approach in domain specific modeling area (Midingoyi, C et. al, 2020) explaines difficulties of implementing codegeneration focuses by several reasons: pure coding concepts are, in most cases, too far from the requirements and from the actual problem domain. Models are used to raise the level of abstraction and hide the implementation details. In a traditional development process, models are, however, kept totally separate from the code as there is no automated transformation available from those models to code (Midingoyi, C et. al, 2020).

In paper (Midingoyi, C et. al, 2020) several approaches of collaborating models and code are considered. Approaches are represented on figure 1.

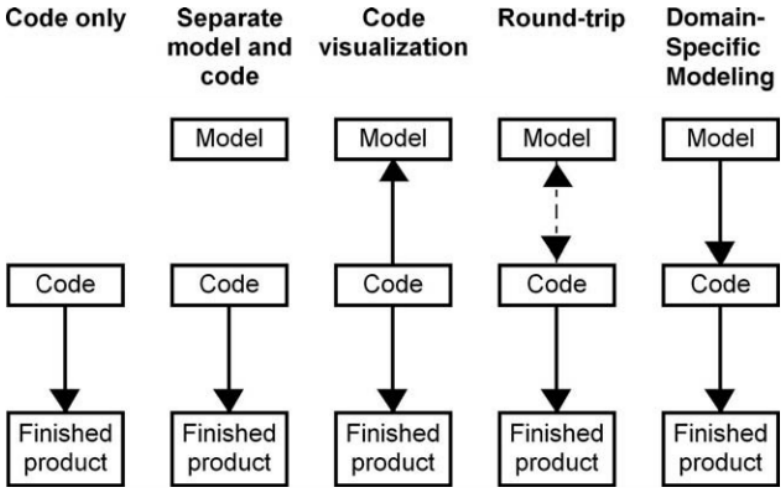

Figure 1 Code and software models collaboration (Midingoyi, C et. al, 2020).

Approach, proposed this in paper is focused on forward engineering activities, namely on round-trip engineering and domain-specific engineering. Also such approaches can be used in other forward engineering software development process.

In order to perform codegeneration operations successfully authors (Midingoyi, C et. al, 2020) propose two languages, namely Crop2ML and CyML.

Language Crop2ML provides a model component specification based on XML meta-language. It consists of unified concepts. A Crop2ML model is an abstract model that may be either a unit model with fine granularity or a composite model represented as a graph of unit models connected by their inputs and outputs to manage model complexity. A model specification contains formal descriptions of the model, the inputs, outputs, state variable initializations, auxiliary functions and a set of parameters and unit tests. Thus, it allows for checking that a model reproduces the expected output values with a given precision. In order to be adopted to model to code transformation tasks CROP needs additional plug-ins and clearly documentation. Abstract model can't consider some platform specific details. For example, there is no multiple inheritance of classes, in C++ vise versa and so on.

Authors performed a great step into development of serious analytical foundation and proposed codegeneration framework, but task to read abstract model is more complex in comparison with UML duagrams. Due to this fact wide using of such languages is limited.

Formal foundation of designing model to model transformation language is proposed in paper (Chebanyuk, 2018). As our codegeneration approach generates object-

oriented code that is a type of model, consider application of the proposed formalization to this approach.

Paper also represents in clear structured way challenges to the abstract syntax of model to model transformation language, to the metamodel of language, to the concrete syntax, and to the transformational rules.

Then author describes elements of mode to model transformation language and proposes a metamodel of Model to Model Transformation language (M2MTL).It is represented in the figure 2.

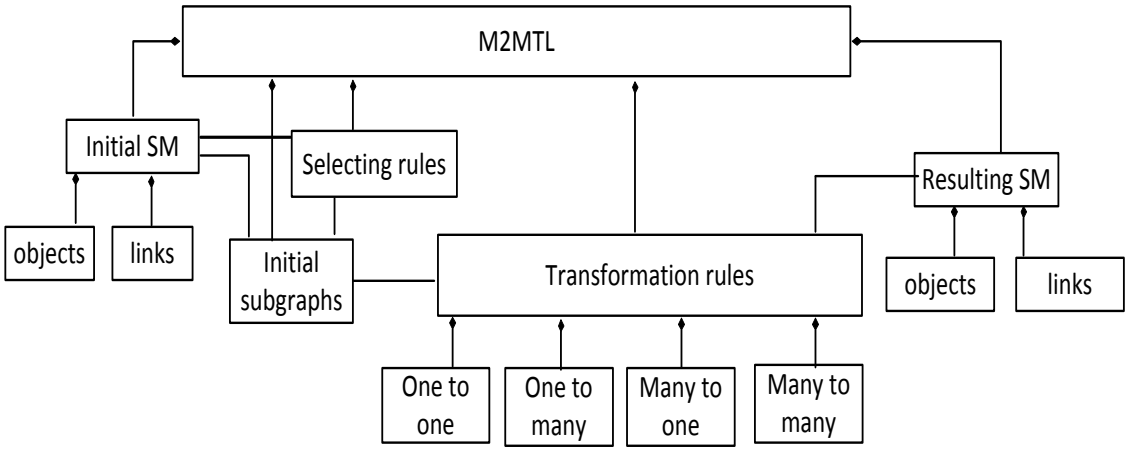

Figure 2 Metamodel of model to model transformation languages (Chebanyuk O., 2018)

Task and research questions

Task: propose analytical foundations of model to code transformation approach. As initial model class diagram is used. As resulting model object model of c# language is used.

Research questions (RQ)

RQ1: Consider architecture of Model to Code Transformation Approach in connection with specific frameworks and tools.

RQ2: Choose flexible analytical apparatus allowing represent structure both of static software model and skeleton of code.

RQ3: Perform an experiment proving correctness of proposed analytical foundation of software model to code transformation framework.

Frameworks and Tools and for proposed Model to code Transformation approach

Distribution of proposed frameworks and tools is represented according to classical schema of Model to Model transformation approach (Figure 3).

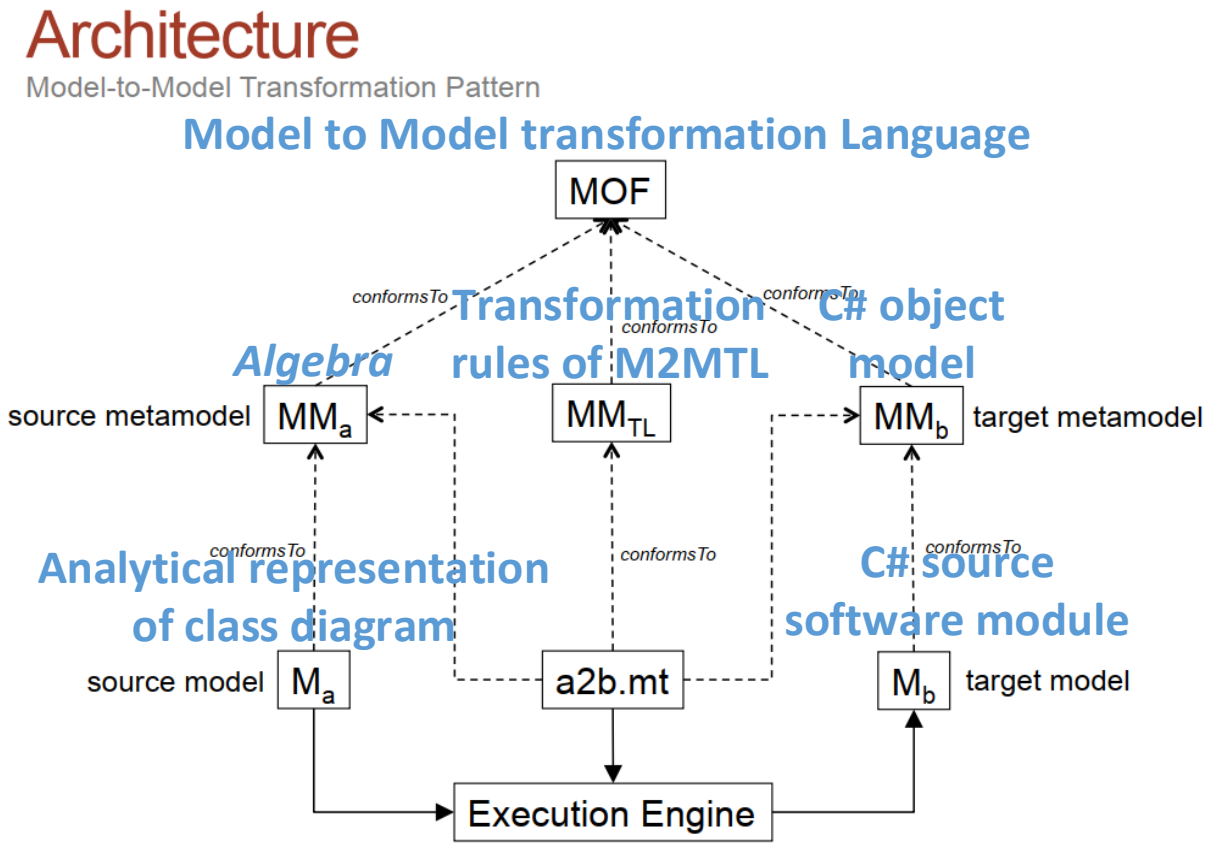

Figure 3 Model to code transformation approach (Cabot J., 2015)

In the Table 1 detailed explanation about used tools for codegeneration approach is represented.

Table 1. Description of Model-to-code transformation approaches architecture

Part of codegeneration architecture	Explanation
Metametalevel	
Metamodel of model to model transformation language (Chebanyuk, 2018).	This metamodel is flexible and contains all necessary elements to describe transformation process with necessary level of details.
Source and target Metamodels	

Source Metamodel Algebra, describing static software models (Chebanyuk, 2013)	Algebra that allows describing structure of class diagram considering all variants of class diagram elements composition.
Target Metamodel	C# object model (Microsoft, 2017). Also can be XMI standard (XMI, 2015)
Model level	Source and target models
Source model	Analytical representation of class diagram in terms of algebra describing software static models
Target models	Modules of C# source code. XMI representation of class diagrams (Chebanyuk E. & Povalyaev D., 2017). Many Modeling environments store UML diagrams in this format
Transformation rules	Rules to represent how to transform initial analytical representation of class diagram to its intermedia analytical representation. Rules how to transform intermedia analytical representation to C# or to XMI representation.
Execution engine	Visual Studio compiler

Analytical representation of transformation rules

Transformation rules must allow to influence on changing structural characteristics of class diagram elements. In order to represent transformation results corresponding analytical approach must be involved. After review of different analytical approaches aimed to reflect information about static diagrams algebra, describing software static model is chosen (Chebanyuk, 2013). Let's describe transformation rules allowing to precise class diagram structure before codegeneration operation.

The first transformation rule aimed to improve structure of class diagram before codegeneration. It is formulated as follows:

If in the class diagram there is class with composition relationship with other classes (denote is as C) these classes must be included to the list of properties of C.

According to (Chebanyuk, 2018) the first step of realization of this rule is to find on class diagram all classes that are connected with other ones with composition relationship.

General selection riles are denoted as follows:

$$select\ S\ \ from(SMI_{type}) = SMI_{sub} \tag{1}$$

Forming SMI_{sub} , allowing to form a set of classes that have composition links with other classes, (namely *COMP*) according to this rule is performed by the following:

$$select\ F^{comp}(C)\ \ from\ ClassDiagram = COMP \tag{2}$$

Transformation rule is consisted from two steps. The first step is to refine class structure, namely for every class from $F^{comp}(C) \in COMP$ add attributes that matches with types of the connected classes. This transformation rule is aimed to refine class diagram structure and it is denoted by the following:

$$\begin{gathered} F^{comp}(class) \longrightarrow F^{comp}(A \cup \bigcup_{i=1}^{n} \alpha(class_i(name)) \times X \times B), i = 1,...,n \\ F^{comp}(class) \in COMP \end{gathered} \tag{3}$$

The second step is to fill class attributes according to the specific template of some programming language. According to C# language template is looking by the following:

$$\begin{gathered} class\ www\{ \\ \quad public\ type1\ attribute1\{get; set;\} \\ \quad ..... \\ \quad public\ \ class_1\ \ \alpha(class_1(name)\{get; set;\} \\ \quad ..... \\ \quad public\ \ class_n\ \ \alpha(class_n(name)\{get; set;\} \\ \} \end{gathered} \tag{4}$$

Second transformation step is represented by the following:

$$\begin{gathered} F^{comp}(A \cup \bigcup_{i=1}^{n} \alpha(class_i(name)) \times X \times B) \longrightarrow class\ www\{ \\ \quad public\ type1\ attribute1\{get; set;\} \\ \quad ..... \\ \quad public\ \ class_1\ \ \alpha(class_1(name)\{get; set;\} \\ \quad ..... \\ \quad public\ \ class_n\ \ \alpha(class_n(name)\{get; set;\} \\ \} \end{gathered} \tag{5}$$

Let's describe other transformation rule, namely adding a list of methods to skeleton of class when class inherits an interface. The rule selecting all classes that inherit interfaces (set *INTERF*) is represented by the following:

$$select\ F^{inh}(C^{public})\ from ClassDiagram = INTERF \quad (6)$$

Transformation rule allowing to complete classes from the set *INTERF* by a set of methods from interfaces is written by the following:

$$\begin{gathered} F^{inh}(C^{public}) \longrightarrow F^{inh}(A \times B \times X \bigcup_{i=1}^{m} B^{j}),\ j = 1,...,m \\ F^{inh}(C^{public}) \in INTERF \end{gathered} \quad (7)$$

The second step is to fill class attributes according to the specific template of some programming language. According to C# language template is looking by the following

$$\begin{gathered} class\ www\{ \\ \quad public\ \beta_{1,1}; \\ \quad public\ \beta_{1,2}; \\ \quad ..... \\ \quad public\ \beta_{1,m_1}; \\ \quad public\ \beta_{2,1}; \\ \quad ..... \\ \quad public\ \beta_{n,k}; \\ \} \end{gathered} \quad (8)$$

Where $\beta_{1,1}$ - is a signature of the first method of the first interface, and generally $\beta_{i,j}$ public signature of method *i* of interface *j*.

Second transformation step is represented by the following:

$$\begin{gathered} F^{inh}(A \times B \times X \bigcup_{i=1}^{m} B^{j}) \longrightarrow \quad class\ www\{ \\ \quad public\ \beta_{1,1}; \\ \quad public\ \beta_{1,2}; \\ \quad ..... \\ \quad public\ \beta_{1,m_1}; \\ \quad ..... \\ \quad public\ \beta_{n,k}; \\ \} \end{gathered} \quad (9)$$

Proposed approach

1. Parse input XMI of class diagram to obtain analytical representation of class diagram according to algebra describing software static models. Parse rules are represented in paper (Chebanyuk E. & Povalyaev D., 2017).
2. On analytical representation of class diagram search fragments that satisfy patterns of transformation rules using (2) and (6).
3. Obtain skeletons of source codes according to transformation rules represented in (5) and (9).
4. Using visual studio codegeneration environment obtain *.cs files with source codes in C# language.
5. Merge textual representation of class diagram fragments, obtained after performing of previous point and skeleton of source code obtained in point 3.
6. Optional point for testing – compile obtained source module by visual studio compiler.

Case study

Let's consider proposed codegeneration approach investigating Visual Studio class designer plug-in (Microsoft, 2018). As it was mentioned in (Shyrokikh, 2020) Visual Studio does not contain composition relationship only association one (figure 4).

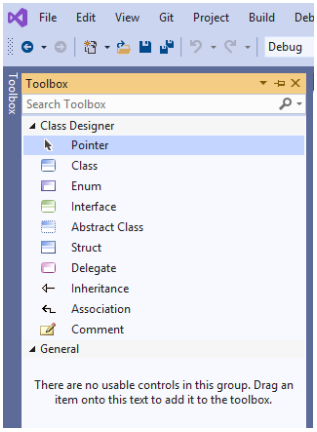

Figure 4. Visual Studio Class designer components

Codegeneration rules of Visual Studio plug-in add datatypes of classes to the central (composition) class. From the other hand it is important to mention that when designer establish association relationship between two classes property pointing that one class becomes a field of other is added automatically. Then developer must think about semantic of generated code and spend additional time for software module analysis and editing.

For example – class diagram that is represented on the figure 5 contains three classes. Screen is a part of SmartPhone, and classes Screen and Smartphone are connected by composition relationship. Classes User and Smartphone must be connected through association relationship.

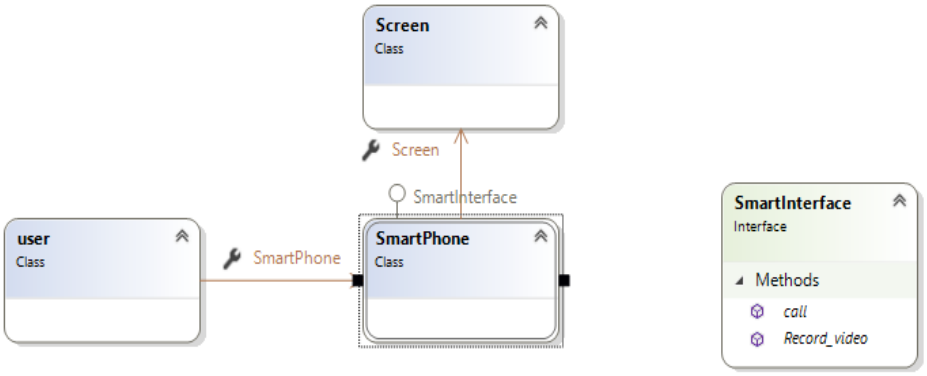

Figure 5. Example of class diagram

Codegeneration plug-in assumes that these two association links are the same. Result of codegenaration plug-in is represented below.

```
  public class user
   {
Extra fragment needed to be deleted
      public SmartPhone SmartPhone
      {
          get => default;
          set
          {
          }
      }
   }
public class SmartPhone : SmartInterface
   {
      public Screen Screen
      {
          get => default;
          set
          {
          }
```

```
    }
Absent fragment – needed to be added
    void call();
    void Record_video();

  }
 public class Screen
  {
  }
public interface SmartInterface
  {
    void call();
    void Record_video();
  }
```

Representation of transformation rules is given in the Table 2.

Table 2. Representation of transformation rules on metalevel and model level

Analytical representation of class diagram initial fragment	Analytical representation of class diagram resulting fragment
Metalevel	
$P(Classes) =$ $\exists class \in Classes$ $where\, F^{comp}(class) \neq \varnothing$	$class^{comp} = A \times X \times B$ $A = A \bigcup_{i=1}^{n} name_i$
Model level	
$F(SmartPhone)^{comp} =$ $= F(SmartPhone) \cup F(Screen)$ $SmartPhone \subseteq A \times X \times B$	$F(SmartPhone)^{comp} =$ $= F(SmartPhone) \cup F(Screen)$ $SmartPhone \subseteq A \times X \times B$ $A^* = A \cup Screen$
Metalevel	
$P(Classes, Ic) =$ $\exists class \in Classes\, \forall I = \{i_i, ..., i_k\}, i \in Ic$ $where\, F^{inh}(class) \bigcup_{i=1}^{n} I_i \neq \varnothing$	$class^{inh} = A \times X \times B$ $B^* = B \bigcup_{i=1}^{n} \bigcup_{j=1}^{m} \beta_{i,j}$
Model level	

$F(SmartPhone)^{inh} = F(SmartPhone) \cup$ $\cup F(SmartInterface)$ $SmartPhone \subseteq A \times X \times B$	$F(SmartPhone)^{inh} = F(SmartPhone) \cup$ $\cup F(SmartInterface)$ $SmartPhone \subseteq A \times X \times B$ $B^{*} = B \cup B(SmartInterface)$

Conclusion

In this paper Codegeneration approach is proposed. Advantages of the proposed approach are the next:

- It uses flexible analytical apparatus for representation of class diagram with given level of details;
- such a representation allows to set transformation rules to improve drawbacks of codegeneration of different designing environments;
- remain for codegenetation environment possibility to design class diagram convenient for human cognitive perception (for example represent relationship between classes graphically).
- transforming analytical representation to XMI and vise versa (Chebanyuk E. & Povalyaev D., 2017) proposed codpgeneration approach can be used to improve round trip engineering activities.

Bibliography

(Cabot J., 2015) https://www.slideshare.net/jcabot/modeldriven-software-engineering-in-practice-chapter-8-modeltomodel-transformations?qid=551330ce-3800-43a8-96b1-0823d778e798&v=&b=&from_search=6

(Chebanyuk O., 2018) Chebanyuk O. Designing of Software Model to Model Transformation Language. International Journal of Computers, Number 3, 2018, 120-129.

(Chebanyuk E., 2013) Chebanyuk O. Algebra, describing software static models. International journal “Information Technologies and Knowledge”, Vol. 7, №1, 2013, 83-93

(Chebanyuk E. & Povalyaev D., 2017) Chebanyuk E. & Povalyaev D. An approach for architectural solutions estimation International journal “Informational Technologies and Knowledge”, Vol 11, number 2, 2017, 114-143

(Microsoft, 2018) How to: Add class diagrams to projects. Access mode: https://docs.microsoft.com/en-us/visualstudio/ide/class-designer/how-to-add-class-diagrams-to-projects?view=vs-2019

(Microsoft, 2017) System.Collections.ObjectModel Namespace Access mode: https://docs.microsoft.com/en-us/dotnet/api/system.collections.objectmodel?view=net-5.0

(Midingoyi, C et. al, 2020) Midingoyi, C. A., Pradal, C., Athanasiadis, I. N., Donatelli, M., Enders, A., Fumagalli, D., ... & Martre, P. (2020). Reuse of process-based models: automatic transformation into many programming languages and simulation platforms. in silico Plants, 2(1), diaa007.

(Shyrokykh, A., 2020) Shyrokykh, A. Review of drawbacks of existing tools transforming platform specific models to code. International Journal "Information Models and Analyses" Volume 9, Number 2, 2020, 185-199

(XMI, 2016) XML Metadata Interchange (XMI) Specification. Version 2.5.1 Access Mode: https://www.omg.org/spec/XMI/2.5.1/PDF

Authors' Information

Anton Shyrokykh *– National Aviation University, Faculty of Cybersecurity, computer and software engineering, graduate student. Kiev, Ukraine; e-mail: anton.black777@gmail.com*

Major Fields of Scientific Research: Model-Driven Development, Distributed long-living transactions