Intelligent Software Development Method by Model Driven Architecture

Keinosuke Matsumoto, Kimiaki Nakoshi, and Naoki Mori
Department of Computer Science and Intelligent Systems
Graduate School of Engineering, Osaka Prefecture University
Sakai, Osaka, Japan
email: {matsu, nakoshi, mori}@cs.osakafu-u.ac.jp

Abstract—**Recently, Model Driven Architecture (MDA) attracts attention in the field of software development. MDA is a software engineering approach that uses models to create products, such as source code. On the other hand, executable UML consists of activities, common behavior, and execution models. However, it has not been put to good use for transforming into source code. This paper proposes a method for transforming executable UML and class diagrams with their association into source code. Executable UML can detail system's behavior enough to execute, but it is very difficult for executable UML to handle system's data. Therefore, the proposed method uses class diagrams and executable UML to transform into source code. The method can make models independent from platform, such as program languages. The proposed method was applied to a system. Source code of Java and C# was generated from the same models of the system, and development cost was verified. As a result, it was confirmed that this method could reduce cost very much when models are reused.**

Keywords—executable UML; activity diagram; model driven architecture; UML.

I. INTRODUCTION

In today's software development, software reuse, modification, and migration of existing systems have increased rather than new development. According to an investigative report [1] of Information-Technology Promotion Agency (IPA), reuse, modification, and migration of existing systems account for about 59.4% of software development and new development for about 40.6%. Many software bugs enter at upper processes, such as requirement specification, system design, and software design. However, the bugs are mostly discovered at lower processes, such as testing process. The request to detect bugs at upstream is coming out. Under such a situation, software developers require development technique that is easy to reuse and deal with changes of implementation technique. Model driven architecture (MDA) [2] is attracting attention as an approach that generates source code automatically from models that are not influenced of implementation [3][4][5]. Its core data are models that serve as design diagrams of software. It includes a transformation to various types of models and an automatic source code generation from the models. Therefore, it can directly link software design and implementation.

The final goal of MDA is to generate automatically executable source code for multiple platforms. For that purpose, it is necessary to make architecture and behavior of a system independent from platforms. Platform Independent Model (PIM) that does not depend on platforms, such as a programming language. Executable UML [6][7] is advocated as this type of model. This expresses all actions for every kind of processing, and input and output data by a pin in an activity diagram, which is one of UML [8] diagrams. The source code for various platforms is generable from one model since processing and data are transformed for every platform if this executable UML is used.

In this study, a method is proposed that generates source code automatically from executable UML. It is very difficult for executable UML to handle system's data. To solve this problem, this paper proposes a modeling tool that associates an executable UML with class diagrams and acquires data from them. It can treat not only data, but can introduce the hierarchical structure of class diagrams in executable UML. If the platform of future systems, such a programing language, is changed, software developers could not reuse existing source code, but they can reuse UML models to generate automatically new source code of the new programming language.

The contents of this paper are as follows. In Section II, related work is described. In Section III, the proposed method is explained. In Section IV, the results of application experiments confirm the validity of the proposed method. Finally, in Section V, the conclusion and future work are presented.

II. RELATED WORK

This study uses related work called Acceleo and executable UML.

A. Acceleo

MDA's core data are models that serve as software design drawing. The models are divided into platform dependent and independent models. Specifically, Acceleo [9] transforms models from PIM to Platform Specific Model (PSM) that depends on a platform, and generates source code automatically. Transformation of PIM is important and it can generate source code of various platforms from PIM by

 ISBN: 978-1-61208-576-0

replacing transformation rules to each platform. Acceleo is plug-in of integrated development environment Eclipse [10], and a code generator that translates MetaObject Facility (MOF) [11] type models into source code on the basis of the code transformation rules called a template. Acceleo can translate the models directly, but the template has many constraints. For example, it cannot hold and calculate data. Therefore, it cannot recognize what type of model elements have been read until now. It is impossible to search the connection between nodes by using graph theory. When branches and loops of activity diagrams are transformed, Acceleo has a problem that it cannot appropriately transform them because it does not understand the environment.

B. *Executable UML*

Executable UML is a model that based on activity diagrams, like shown in Fig. 1. It has the following features:

- An action is properly used for every type.
- Input and output data of each action are processed as a pin, and they are clearly separated from the action.
- Model library that describes the fundamental operation in the model is prepared.

Each type of action has respectively proper semantics, and transformation with respect to each action becomes possible by following the semantics. The type and its semantics of action used in executable UML are show in the following.

1) ValueSpecificationAction: It outputs a value of the primitive type data like an integer, real number, character string, and logical value.

2) ReadStructuralFeatureAction: It reads a certain structural characteristics. For example, it is used when the property of class diagrams is read.

3) ReadSelfAction: It reads itself.

4) CallOperationAction: It calls methods in class diagrams.

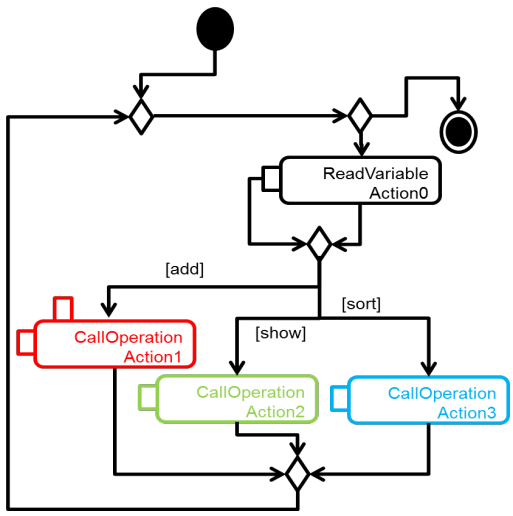

Figure 1. Example of executable UML.

5) CallBehaviorAction: It calls behaviors in behavior diagrams.

6) AddVariableValueAction: It addes a value to the variable or replace the variable by the value.

7) ReadVariableAction: It reads a variable or generate it.

8) CreatObjectAction: It creates a new object.

The model library consists of Foundational Model Library, Collection Classes, and Collection Functions. The contents of the model library are shown below

1) Foundational Model Library: It offers primitive type of data, and their behavior (four arithmetic operations, comparison, etc.) and the input-output relations.

2) Collection Classes: It offers collection class of Set, Ordered Set, Bag, List, Queue, Dequeue, and Map.

3) Collection Functions: It offers the methods (add, delete, etc.) of the collection class.

The model library is used by calling CallOperationAction or CallBehaviorAction.

III. PROPOSED METHOD

This section explains the technique of transforming executable UML to source code. Although executable UML is useful, this model has not been put to good use for automatic generation of source code. Moreover, the handling of data is inadequate by using only executable UML. To solve this problem, a method is proposed for generating source code automatically from executable UML. The method utilizes a modeling tool that associates executable UML with class diagrams. If executable UML needs data, the method gets the data from associated class diagrams.

The outline of the proposed method is shown in Fig. 2. Skeleton code is transformed from class diagrams by using Acceleo templates [12] for classes. The skeleton code consists only of class names, field, and methods that do

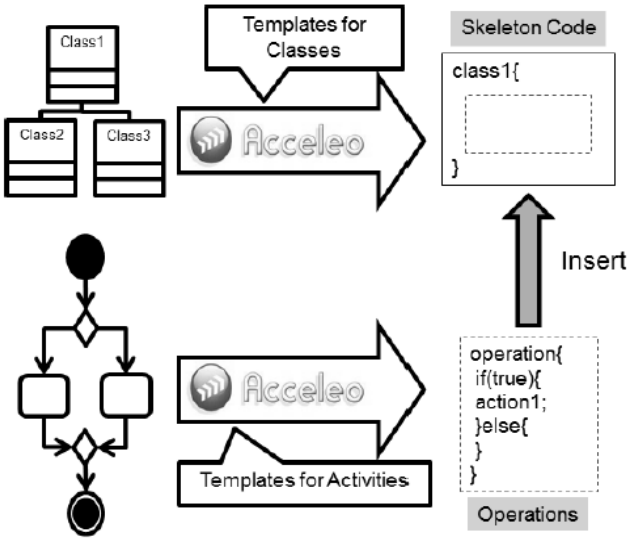

Figure 2. Schematic diagram of the proposed method.

 ISBN: 978-1-61208-576-0

not have specific values of data. Data and a method are automatically generated from executable UML afterwards. Since it is associated with class diagrams, other methods and data in the classes are acquirable using this association. Papyrus UML [13] was used for the association between these models.

A. Transformation from Class Diagrams to Skeleton Code

UML to Java Generator [14] was used as transformation rules from class diagrams to skeleton code. What are generated by these rules are shown below.

- Connection of inheritance or interface
- Field variables and methods like getter and setter
- Names and parameters of member functions

This is a template for Java. When transforming models to C#, it goes through several changes, such as deletion of constructors and addition of ":" to inheritance relationship.

B. Transformation from Executable UML to Source Code

Executable UML are based on activity diagrams. It consists of actions, data, and their flows. Although transform rules of actions and data differ from platform to platform, the flows are fundamentally common. Therefore, transformation of flows is separated from transformation of actions and data. Flows decide the order of transformation of actions and data. This separation can flexibly transform one model to source code of multiple platforms. The transformation flow of executable UML is described in the following.

1) Transformation of flows: A flow of executable UML is shown by connecting nodes, which include actions and data, with an edge. However, neither a branch nor loop is transformed only by connecting nodes along the flow. On transforming a decision or merge node that are used for a branch or loop, the proposed method searches a part of the executable UML near the node and gives an appropriate keyword to a connecting node and edge. The method transforms them according to the keywords. The keywords given to model elements are shown below.

a) finish: It means a node or edge whose processing are finished.

b) loop: It means a decision node in the entrance or exit of a loop.

c) endif: It means a merge node in the end of a branch.

d) read: It means an edge under searching.

The flow of search is shown in Fig. 3 and its algorithm is shown below.

(1) Follow an edge that is not searched in the reverse direction of its arrow.
(2) If an edge and node that are not searched, give the keyword of 'read'.
(3) If a node has the keyword of 'read', replace the keyword to 'loop'. If the node is a decision node, give the keyword of 'loop' to a searched edge going out from the decision node.

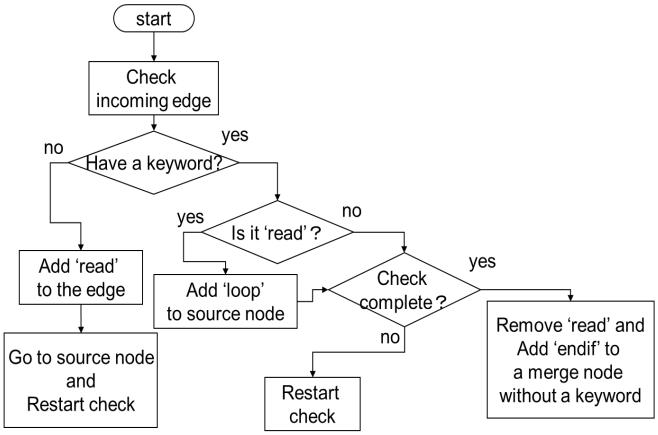

Figure 3. Search algorithm.

(4) Repeat the processing of (1) - (3) until there is no edge that can be searched.
(5) Remove 'read' at the last. If there is a merge node that does not have 'loop', 'endif' will be given to it.

2) Transformation of actions and data

Transformation rules of actions and data are prepared for every platform. In executable UML, an action is properly used for every type of processing, and a transformation rule may be defined per action. The flow of transformation processing is shown in the following.

(1) If a node is an action, it will be transformed and given keyword of finish. The processing will move to the next nodes. If the action has an input pin, its flow will be gone back and the objects and actions in the starting point of this flow will be transformed.
(2) If a node is a decision node and it has the keyword of loop, it will be transformed by rules for loop. If the decision node has no keyword, it will be transformed by rules for branch. In addition, the following nodes and conditional expressions are picked out from the connecting edges. Keywords of finish are given to these nodes and edges and the processing will move to the next nodes.

TABLE I. ACTIONS AND THEIR APPLICATIONS TO EACH LANGUAGE.

Action	Java	C#
CreateObjectAction	new *<object>*	new *<object>*
ReadSelfAction	this	this
ValueSpecification Action	*<value>*	*<value>*
ReadStructural FeatureAction	*<object>.<variable>*	*<object>.<variable>*
	(*<resultType>*) *<object>*	*<resultType>*.Parse (*<object>*)
CallOperationAction	*<target>.<operation>* (*<parameter>*)	*<target>.<operation>* (*<parameter>*)
AddVariableValue Action	*<variable>=<value>*	*<variable>=<value>*

TABLE II. MODEL LIBRARY ELEMENTS AND THEIR APPLICATIONS.

Model Library	Java	C#
ReadLine	(new BufferedReader(new InputStreamReader(System.in))).readLine()	Console.ReadLine()
WriteLine	System.out.println(*value*)	Console.WriteLine(*value*)
List.size	<*target*>.size()	<*target*>.Count
List.get	<*target*>.get(<*index*>)	<*target*>[<*index*>]
List.add	<*target*>.add(<*data*>)	<*target*>.Add(<*data*>)
Primitive Functions	<*x*><*function*><*y*>	<*x*><*function*><*y*>

(3) If a keyword is given to a merge node, it will be transformed according to it.

Correspondence relation of actions with Java and C# is shown in Table I. The upper row of ReadStructural FeatureAction in Table I is the case where <variable> is specified, and the lower row is the case where it is not specified.

In addition, correspondence relation of model libraries with Java and C# is shown in Table II. ReadLine and WriteLine are model libraries for input and output. List.size, List.get, and List.add are prepared by Collection Functions, and they are used for output of list capacity, extraction of list element, and addition of list element, respectively. Primitive Functions are operations prepared in the primitive type. Collection Functions are used by calling CallOperationAction, and other functions except them are used by calling CallBehaviorAction. Variables inputted by pins and operators defined in the library are assigned in italics surrounded by <>.

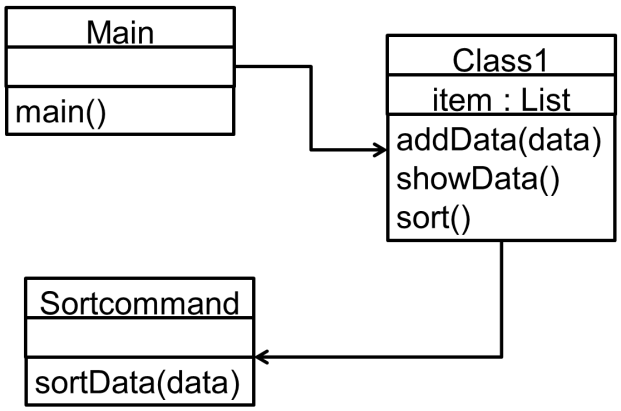

Figure 4. Class diagram of example system.

IV. APPLICATION EXPERIMENTS

As an experiment that verifies operation and development cost of created templates, a system shown below was developed by the proposed method. The system receives three commands of adding data to a list, sorting list elements, and outputting data. This system was described by executable UML and class diagrams. Source code of Java and C# was generated automatically. The same models were used for model transformation of both languages. Operations were checked by using the source code. Figures 4 and 5 show the class diagram and the behavior model of the system.

The generated source code of Java and C# is shown in Figures 6, 7, and 8, respectively. The development cost is

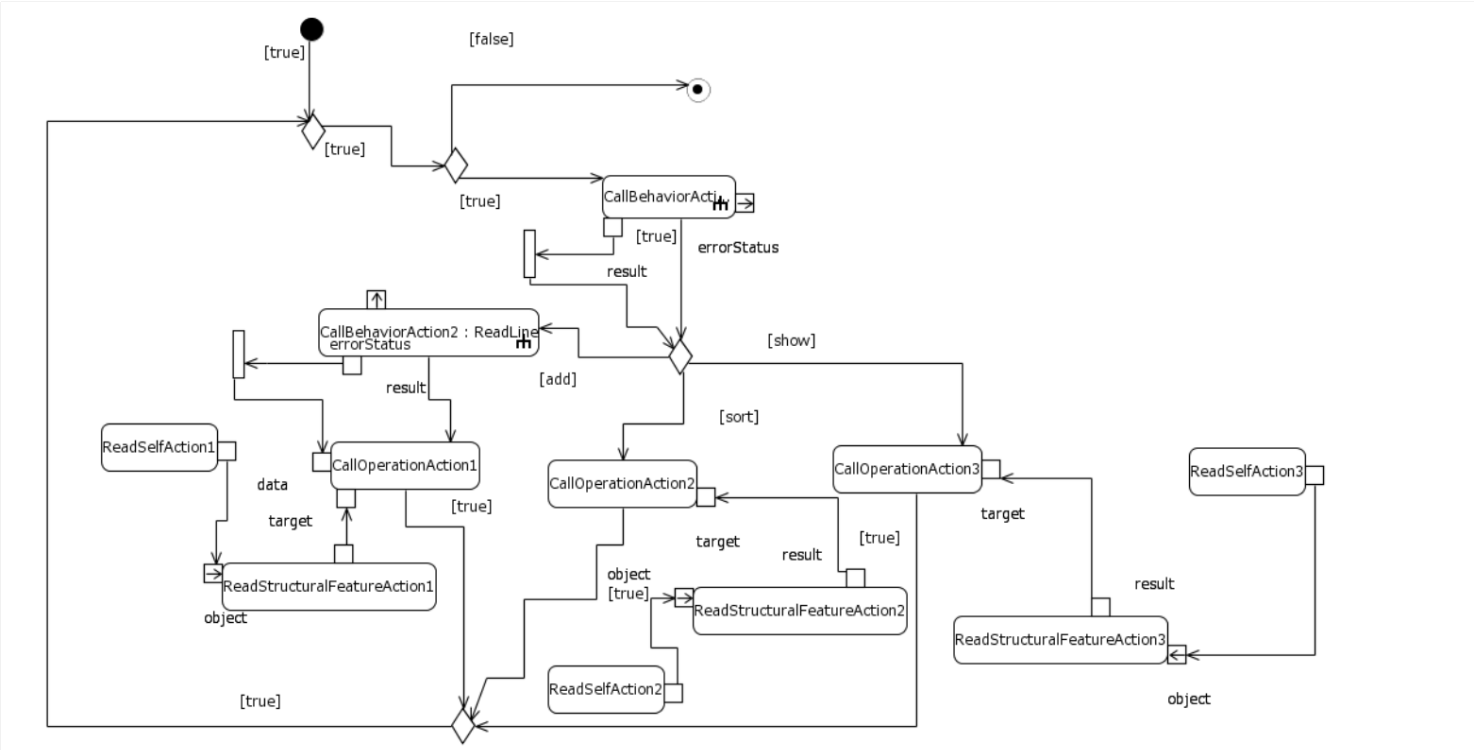

Figure 5. Total system behavior of example system.

evaluated according to [15]. It assumes that workload to add one node in UML diagrams equals to that to describe one line of source code. Table III shows the number of model nodes, the number of lines of added or modified lines, and finished source code of each language. The added and modified lines correspond to parts that cannot be expressed by executable UML like package, import, and so on. Table IV shows the rate of automatically generated code to the finished code. In addition, it shows the rate of cost of new development and reuse as compared with manual procedures from scratch to finish. The calculation formulas used in Table IV are shown below.

```
package Package1;
import java.util.ArrayList;
public class Class1 {
    private Sortcommand sortcommand=new Sortcommand();
    private List item=new List();
    public Class1() {
        super();
    }
    public void addData(String data) {
        this.item.add(data);
    }
    public void showData() {
        System.out.println(this.item);
    }
    public void sort() {
        this.sortcommand.sortData(this.item);
    }
    public Sortcommand getSortcommand() {
        return this.sortcommand;
    }
    public void setSortcommand(Sortcommand newSortcommand) {
        this.sortcommand = newSortcommand;
    }
    public List getItem() {
        return this.item;
    }
    public void setItem(List newItem) {
        this.item = newItem;
    }
}
```

Figure 6. Generated Java code of Class1.

```
using System;
using System.Collections.Generic;
namespace Package1
{
   public class Class1 {
      private Sortcommand sortcommand=new Sortcommand();
      private List item=new List();
      public  void sort() {
         this.sortcommand.sortData(this.item);
      }
      public  void showData() {
         Console.WriteLine(this.item);
      }
      public  void addData(String data) {
         this.item.Add(data);
      }
      public Sortcommand getSortcommand() {
         return this.sortcommand;
      }
      public void setSortcommand(Sortcommand newSortcommand) {
         this.sortcommand = newSortcommand;
      }
      public List getItem() {
         return this.item;
      }
      public void setItem(List newItem) {
         this.item = newItem;
      }
   }
}
```

Figure 7. Generated C# code of Class1.

$$\text{Production rate} = (\text{finished code lines} - \text{added and modified lines}) * 100 / \text{finished code lines} \quad (1)$$

$$\text{New development cost rate} = (\text{model nodes} + \text{added and modified lines}) * 100 / \text{finished code lines} \quad (2)$$

$$\text{Reuse cost rate} = (\text{added and modified lines}) * 100 / \text{finished code lines} \quad (3)$$

Cost by the proposed method is about 130 - 160% in developing new software, but it is held down to less than 10% in reusing them. According to the investigative report of IPA, about 60% of software development is reuse, modification, and migration of existing systems and new software development occupies about 40%. If a system is developed by the proposed method, the cost is

```
package Package1;
import java.util.List;
public class Sortcommand {
    public Sortcommand() {
        super();
    }
    public void sortData(List<Integer> data) {
        int dumy = 0;
        int min = 0;
        int j = 0;
        int s = 0;
        int k = 0;
        while (k < data.size() == true) {
            min = (int) data.get(k);
            s = k;
            j = k + 1;
            while (j < data.size() == true) {
                if (min < (int) data.get(j) == true) {
                    min = (int) data.get(j);
                    s = j;
                } else
                if (min < (int) data.get(j) == false) {
                }
                j = j + 1;
            }
            dumy = (int) data.get(k);
            data.set(k, data.get(s));
            data.set(s,dumy);
            k = k + 1;
        }
    }
}
```

Figure 8. Generated Java code of Sortcommand.

TABLE III. COMPARISON OF MODEL NODES AND GENERATED LINES.

Number of model nodes	Languages	Number of added or modified lines	Number of finished lines
94	Java	3	74
	C#	5	65

TABLE IV. COMPARISON OF DEVELOPMENT COST.

Languages	Production rate	New development cost	Development cost by reusing
Java	96%	131%	4%
C#	92%	152%	8%

 ISBN: 978-1-61208-576-0

$$10*0.6+160*0.4=70(\%) \quad (4).$$

Although the proposed method is more expensive than manual procedures in new development, it can be less expensive in when it comes to reusing. Created templates can be diverted to other projects and the cost declines further by repeating reuse. In present software development with much reuse, a large effect can be expected. The systems can be hierarchically divided into several classes for every function.

V. CONCLUSION

Based on the situation where the rate of reuse, modification, and migration of existing systems is increasing in software development, a MDA method that uses executable UML jointly with class diagrams was proposed in this paper. The proposed method associates class operations with executable UML. Source code of Java and C# was generated from the same models of the system, and development cost was verified.

If the platform of future systems is changed, software developers could not reuse existing source code, but they can reuse UML models to generate automatically new source code of the new programming language. As a result, it was confirmed that this method could reduce cost very much when models are reused. The proposed method can transform models into source code written in any kinds of programming languages if there is an appropriate template. However, how to make models in the method is very vague.

As future work, it is necessary to decide a standard of model partitioning and a notation system of objects that are not defined by executable UML. In addition, important future task is investigating what type of problems will occur when models are changed.

ACKNOWLEDGMENT

This work was supported in part by JSPS KAKENHI Grant Number JP16K06424.

REFERENCES

[1] Information-Technology Promotion Agency, “Actual condition survey on software industry in the 2011 fiscal year,” (in Japanese) [Online]. Available from: https://www.ipa.go.jp/files/000026799.pdf, 2017.06.06.

[2] S. J. Mellor, K. Scott, A.Uhl, and D.Weise, MDA distilled: Principle of model driven architecture. Addison-Wesley Longman Publishing Co., Inc. Redwood City, CA, 2004.

[3] T. Buchmann and A. Rimer, “Unifying modeling and programming with ALF,” Proc. of the Second International Conference on Advances and Trends in Software Engineering, pp. 10-15, 2016.

[4] M Usman, N. Aamer, and T. H. Kim, “UJECTOR: A tool for executable code generation from UML models. In Advanced Software Engineering and Its Applications, ASEA 2008, pp. 165-170, 2008.

[5] Papyrus User Guide, [Online]. Available from: http: //wiki.eclipse.org/Papyrus_User_Guide#Code_Generation_Support, 2017.06.06.

[6] Object Management Group, “Semantics of a foundational subset for executable UML models,” [Online]. Available from: http://www.omg.org/spec/FUML/1.1/, 2017.06.06.

[7] Object Management Group, “List of executable UML tools,” [Online]. Available from: http://modeling-languages.com/list-of-executable-uml-tools/, 2017.06.06.

[8] Object Management Group, “Unified modeling language superstructure specification V2.1.2, ” 2007.

[9] Acceleo: [Online]. Available from: http://www.eclipse.org/acceleo/, 2017.06.06.

[10] F. Budinsky, Eclipse modeling framework: A developer's guide. Addison-Wesley Professional, 2004.

[11] Object Management Group, “Metaobject facility,” [Online]. Available from: http://www.omg.org/mof/, 2017.06.06.

[12] Acceleo template: [Online]. Available from: https://www.eclipse.org/home/search.php?q=template, 2017.06.06.

[13] A. Lanusse et al. “Papyrus UML: An open source toolset for MDA,” Proc. of the Fifth European Conference on Model-Driven Architecture Foundations and Applications, pp. 1-4, 2009.

[14] UML to Java Generator [Online]. Available from: https://marketplace.eclipse.org/content/uml-java-generator, 2017.06.06.

[15] K. Matsumoto, T. Maruo, M. Murakami, and N. Mori, “A graphical development method for multiagent simulators,” Modeling, Simulation and Optimization - Focus on Applications, Shkelzen Cakaj, Eds., pp. 147-157, INTECH, 2010.

 ISBN: 978-1-61208-576-0